**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

2 septembrie 2013

Timp de lucru: 100 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**1 punct**) De ce **NU** este posibil ca un planificator s˘a fie, simultan, echitabil (fair) s, i productiv (s˘a ofere un throughput mare)?

* Pentru că ai un overhead la schimbarea de task-uri. Ca să fii echitabil, fiecare proces trebuie să aibă un timp mai mult sau mai puțin egal pe procesor, însă schimbarea de pe un proces pe altul durează -> nu poți fi productiv.

**2.** (**1 punct**) De ce este afectat sistemul gazd˘a (host) ˆın cazul aparit, iei unei erori fatale la nivelul nucleului unui container OpenVZ?

**3.** (**1 punct**) Funct, ia malloc aloc˘a memorie. Funct, ia calloc aloc˘a memorie s, i completeaz˘a spat, iul cu zero-uri. De ce un apel calloc genereaz˘a, de obicei, mai multe page fault-uri decˆat malloc?

Pentru că malloc nu verifică dacă memoria este alocată sau nu, și nu îți dai seama de asta decât atunci când faci o scriere. Calloc scrie 0 în fiecare bit, deci dacă apar probleme de permisiuni, calloc o să țipe (not the most decent explanation)

**4.** (**1 punct**) De ce este preferat˘a folosirea apelurilor asincrone, acolo unde exist˘a, ˆın locul celor sincrone?

* Pentru că prin folosirea apelurilor sincrone, procesul este blocat, așteptând răspunsul apelului. La folosirea unui apel asincron, procesul poate executa alte operații (sau poate fi înlocuit pe procesor de alt proces) în timp ce așteaptă după răspunsul la apel)

**5.** (**1 punct**) Trei thread-uri afis, eaz˘a continuu respectiv mesajul "red", "green", "blue". Descriet, i, t, inˆand cont de sincronizare, cele trei funct, ii aferente thread-urilor astfel ˆıncˆat s˘a se afis, eze "red green blue red green blue red green blue ...".

**6.** (**1 punct**) De ce este indicat, din punct de vedere al securit˘at, ii, s˘a fie folosit apelul

printf("%s", argv[1]) ˆın locul apelului printf(argv[1])?

* Pentru că în primul apel printf o să afișeze un string. În al doilea apel, printf poate afișa orice (inclusiv cifre, ce ar cauza un overflow sau type mismatch)

**7.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *fis, ier mapat ˆın memorie* s, i *apel de sistem*.

**8.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *hard link, symbolic link* s, i *inode*.

**9.** (**1 punct**) Care sunt asocierile dintre clasele asincron, zero-copying, sincron blocant s, i apelurile/tehnologiile read, readv, ReadFile, mmap, sendfile, Completion Ports, io submit?

**10.** (**1 punct**) Care sunt asocierile dintre perechile de mai jos (unul la unul, unul la mai multe, mai multe la unul, mai multe la mai multe)?

(procesor, registru) - unul la mai multe

(spat, iu de adres˘a, stiv˘a) - unul la mai multe

(procesor, proces) - unul la mai multe

(hard link, inode) -mai multe la unul

(mutex, thread-uri ˆın as, teptare) - unul la mai multe

**11.** (**2.5 puncte**) Se d˘a un server TCP pe care trebuie s˘a ˆıl caracterizat, i din punct de vedere al performant, ei. ˆIn general, asta ˆınseamn˘a s˘a m˘asurat, i, de exemplu:

*•* num˘arul de cereri servite pe secund˘a

*•* num˘arul maxim de client, i servit, i simultan

*•* throughput de date

s, i cum variaz˘a aceste trei m˘arimi una ˆın funct, ie de cealalt˘a.

Descriet, i cum realizat, i o arhitectur˘a de sistem (topologie, leg˘aturi, scenarii de folosire) care s˘a poat˘a m˘asura aces, ti parametri. Elaborat, i avˆand ˆın vedere faptul c˘a software-ul de server ruleaz˘a pe un calculator server mult mai performant decˆat calculatoarele pe care le avet, i la dispozit, ie pentru testare.

**12.** (**2.5 puncte**) Pentru aceeas, i situat, ie ca la punctul anterior, descriet, i arhitectura software a software-ului de m˘asur˘a: single threaded, multi-threaded, multi-proces? Ce alt, i parametri at, i mai putea m˘asura?

10 iunie 2013

**1.** (**1 punct**) S, tiind c˘a overhead-ul unui apel de sistem este de 7 ms s, i overhead-ul trat˘arii unui page fault este de 2 ms, ˆın ce situat, ie un apel memcpy va dura mai mult decˆat un apel de sistem?

**2.** (**1 punct**) De ce este o eroare de tip *Segmentation fault* urmat˘a de o schimbare de context?

Pentru că după ce se rulează handler-ul întreruperii de segfault, fie se va pasa contextul sistemului de operare, în cazul în care programul nu are handler pentru segfault, fie contextul va deveni cel al programului în cauză, pentru a își putea reveni din situația creată

**3.** (**1 punct**) De ce putem crea un container OpenVZ ˆın cadrul unui mas, ini virtuale VMware Workstation, dar nu s, i invers?

**4.** (**1 punct**) Precizat, i s, i justificat, i valoarea de adev˘ar a urm˘atoarei afirmat, ii: *Un apel write blocant apelat de un proces multithreaded cu implementare ˆın user-space* ***NU*** *va cauza un TLB flush*.

Nu va cauza un TLB flush pentru că apelul este executat în user-space, deci se face buffered

**5.** (**1 punct**) Un proces execut˘a secvent,a:

for (i = 0; i < 42; i++)

a++;

ˆIn ce situat, ie poate genera aceast˘a secvent, a˘ un TLB flush?

**6.** (**1 punct**) Majoritatea planificatoarelor I/O (*I/O scheduler-ul* ) realizeaz˘a operat, ii de tip *sorting and merging* pe cererile de lucrul cu discul. De ce aceste planificatoare sunt utile pentru discuri IDE, dar nu sunt potrivite pentru dispozitive de stocare de tip SSD (*Solid State Drive* )?

Pentru ca SSD-urile sunt construite pe o cu totul alta tehnologie fata de HDD-uri, tehnologie in care nu se foloseste un cap de citire deci nu exista latenta la seek, algoritmii de disk-scheduling nu isi au rostul. In orice ordine s-ar realiza operatiile, overhead-ul e acelasi.

**7.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *ASLR* (*Address Space Layout Randomization* ) s, i *shellcode*.

Folosirea ASLR împiedică rularea de shellcode-uri de pe stivă prin poziționarea stivei în diverse zone din spațiul de adresă; atacatorul nu poate ști (ușor) care este adresa de start a shellcode-ului. Are eficiență în special pe sistemele pe 64 de biți. Pe cele pe 32 de biți se poate folosi brute force.

**8.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *stack overflow* s, i *thread*.

Folosirea unui număr mare de thread-uri în cadrul unui proces poate conduce mai rapid la stack overflow. Fiecare thread are stiva proprie, cu dimensiunea fixată la crearea. Dacă se creează prea multe thread-uri, se va ocupa foarte mult stiva. Stivele thread-urilor vor fi apropiate unele de altele astfel că, în cazul unui flux de apeluri mare (apeluri recursive, de exemplu), există riscul ca stiva unui thread să suprascrie stiva altui thread.

**9.** (**1 punct**) Fie urm˘atoarea comand˘a rulat˘a ˆıntr-un shell. Identificat, i procesele, relat, iile p˘arinte-copil, descriptorii de fis, ier s, i redirect˘arile existente.

cat examen | grep raspuns > corect

cat examen | grep raspuns > corect

Procese:

Bash

Cat

Grep

Relații

Bash – părinte la cat

Cat – părinte la grep

Descriptori de fișier:

Pipe va avea doi descriptori de fișier (output-ul de la cat examen și input-ul de la GREP

Mai avem un descriptor către fișierul corect, în care scrie grep

Redirectări

Stdout de la cat la stdin grep

Stdout grep la corect

**10.** (**1 punct**) Ce drepturi (citire, scriere, execut, ie) au urm˘atoarele zone din spat, iul de adres˘a al unui proces: text, data, rodata, bss, stiv˘a, heap, biblioteci mapate? Justificat, i.

Executable files include four canonical sections called, by convention, .text, .data, .rodata, and .bss. The .text section contains executable code and is packed into a segment which has the read and execute access rights. The .data and .bss sections contain initialized and uninitialized data respectively, and are packed into a segment which has the read and write access rights.

**11.** (**2.5 puncte**) Se cere s˘a construit, i un server care ascult˘a pe un socket TCP s, i serves, te cereri efectuate ˆıntr-un limbaj propriu ˆımpachetat in XML. Este nevoie s˘a ofere suport pentru maxim 1000 de client, i simultan s, i s˘a serveasc˘a 10.000 de cereri pe secund˘a ˆın total din partea acestor client, i. Alc˘atuit, i schema bloc a acestui server s, i detaliat, i blocul de comunicat, ie peste TCP ˆın pseudocod. Explicat, i alegerile f˘acute.

Blocurile/modulele implicate intro diagrama, despre care ar trebui discutat, sunt:

1. TCP communication
2. I/O model (async vs threading)
3. XML parser

Daca le luam de la coada la cap, parserul XML trebuie sa fie lightweight si sa implementeze un subset necesar comunicarii intre client si server.

Modelul de I/O, daca ne uitm la numarul maxim de clienti cerut de specificatii, de 1000, putem sa implementam folosind 1000 de threaduri care asculta pe acelasi socket. Daca insa ne uitam la faptul ca trebuie sa serveasca un numar relativ mare de tranzactii pe secunda, probabil ca un model async I/O ar fi mai potrivit, eliminand context switchurile dintre cele 1000 de threaduri din celalalt model. In cele din urma, probabil ca un model hibrind cu un thread care asculta pe un socket si foloseste async IO si un numar de threaduri care este o functie de numarul de core-uri este cel mai eficient.

Modulul de TCP are o particularitate interesanta. Din starea de listen, la aparitia unei conexiuni noi se creeaza un fd de date. TCPul fiind un protocol de tip stream, sunt doua posibilitati. Daca clientii trimit cereri care au toate nevoie de raspuns, atunci este ok, serverul citeste de pe socket pana cand se poate forma un mesaj XML corect, apoi il trimite mai departe catre procesare si trimite raspunsul inapoi pe aceeasi conexiune de date, dupa care inchid conexiunea. Cazul mai complicat este cand un client poate trimite notificari catre server fara sa astepte raspuns, caz in care este necesar sa se detecteze si sa se delimiteze mai multe requesturi in acelasi buffer. In acest caz este posibil sa apara si desincronizari intre client si server, daca clientul trimite notificari foarte rapid, bufferul de TCP poate sa contina requesturi incomplete, ceea ce complica detectia de mesaje bine formate.

**12.** (**2.5 puncte**)

Un program are nevoie s˘a stocheze 1.000.000 de fis, iere pe disc, fiecare de

10MB, care s˘a poat˘a fi accesate pe baz˘a de identificatori numerici unici. Alc˘atuit, i schema bloc a unui sistem de stocare care s˘a ofere suport pentru acest volum de informat, ii s, i detailat, i blocul de identificare a fis, ierului pe disc.

Aici problema consta in faptul ca nu este scalabil sa stochezi 1,000,000 de fisiere in acelasi director, si nici intro baza de date nu prea are sens sa stochezi ca bloburi toata povestea asta care insumeaza 10TB de date. Asadar, e clar ca fisierele vor trebui stocate pe disc in foldere separate, cel mai bine intro structura arborescenta, care sa se poata intinde pe oricate filesystemuri, deoarece e vorba de o capacitate mai mare decat discurile uzuale. Asadar, blocurile implicate in aceasta solutie sunt doua:

1. Bloc de identificare
2. Bloc de stocare

Blocul de identificare se poate implementa cu o tabela simpla in orice sistem de baze de date, care face o mapare de la un identificator unic la o cale de fisier pe disc.

Blocul de stocare are un API simplu, prin care i se cere, pentru un nou fisier de stocat, locul in care se va stoca. Pentru a face asta, trebuie sa stie ce discuri exista in sistem, ce capacitati au si ce filesystem au, pentru a determina un numar optim de intrari in director pentru fiecare dintre ele. Cand se cere un slot pt un fisier nou, sistemul cauta pe volumele existente cel mai bun loc in termeni de spatiu disponibil, intrari in director, etc. Se poate implementa si un load balancer care sa distribuie fisierele cele mai cerute pe discuri diferite, etc.

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, declar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

14 iunie 2013

Timp de lucru: 100 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**1 punct**) Fie T1 timpul de mutare a unui fis, ier pe aceeas, i partit, ie s, i T2 timpul de mutare a aceluias, i fis, ier pe o alt˘a partit, ie. Ce relat, ie exist˘a ˆıntre T1 s, i T2 s, i de ce?

*T1 < T2*

*In cazul in care se muta un fisier pe aceeasi partitie, tot ce se intampla este crearea unui nou hard link catre inode-ul referentiat de fisier si stergerea vechiului link. In situatia in care se muta fisiere intre partitii, datele chiar se muta. Astfel, T2 este mai mare deoarece are overhead de la mutarea datelor.*

**2.** (**1 punct**) Bibliotecile partajate cont, in zone de memorie mapate ˆın cadrul proceselor cu urm˘atoarele permisiuni: r-x (cod), r-- (read-only data) s, i rw- (date). Care din aceste zone **NU** trebuie s˘a fie partajate ˆıntre mai multe procese care folosesc aceeas, i bibliotec˘a s, i de ce?

**3.** (**1 punct**) Dou˘a sisteme identice din punct de vedere al hardware-ului s, i al sistemului de operare sunt folosite ˆın scopuri diferite. Primul este folosit pentru calculul de transformate Fourier, iar al doilea este folosi drept server (web, e-mail, ssh, dns, dhcp, etc.). Care dintre cele dou˘a va petrece mai mult timp ˆın schimb˘ari de context s, i de ce?

Al doilea!

There are three potential triggers for a context switch:

### Multitasking[[edit](http://en.wikipedia.org/w/index.php?title=Context_switch&action=edit&section=2)]

Most commonly, within some [scheduling](http://en.wikipedia.org/wiki/Scheduling_(computing)) scheme, one process needs to be switched out of the CPU so another process can run. This context switch can be triggered by the process making itself unrunnable, such as by waiting for an [I/O](http://en.wikipedia.org/wiki/Input/output) or [synchronization](http://en.wikipedia.org/wiki/Synchronization_(computer_science)) operation to complete. On a [pre-emptive multitasking](http://en.wikipedia.org/wiki/Pre-emptive_multitasking) system, the scheduler may also switch out processes which are still runnable. To prevent other processes from being starved of CPU time, preemptive schedulers often configure a timer interrupt to fire when a process exceeds its [time slice](http://en.wikipedia.org/wiki/Time_slice). This interrupt ensures that the scheduler will gain control to perform a context switch.

### Interrupt handling[[edit](http://en.wikipedia.org/w/index.php?title=Context_switch&action=edit&section=3)]

Modern architectures are [interrupt](http://en.wikipedia.org/wiki/Interrupt) driven. This means that if the CPU requests data from a disk, for example, it does not need to [busy-wait](http://en.wikipedia.org/wiki/Busy-wait) until the read is over; it can issue the request and continue with some other execution. When the read is over, the CPU can be *interrupted* and presented with the read. For interrupts, a program called an [*interrupt handler*](http://en.wikipedia.org/wiki/Interrupt_handler) is installed, and it is the interrupt handler that handles the interrupt from the disk.

When an interrupt occurs, the hardware automatically switches a part of the context (at least enough to allow the handler to return to the interrupted code). The handler may save additional context, depending on details of the particular hardware and software designs. Often only a minimal part of the context is changed in order to minimize the amount of time spent handling the interrupt. The [kernel](http://en.wikipedia.org/wiki/Kernel_(computing)) does not spawn or schedule a special process to handle interrupts, but instead the handler executes in the (often partial) context established at the beginning of interrupt handling. Once interrupt servicing is complete, the context in effect before the interrupt occurred is restored so that the interrupted process can resume execution in its proper state.

### User and kernel mode switching[[edit](http://en.wikipedia.org/w/index.php?title=Context_switch&action=edit&section=4)]

When a transition between [user mode](http://en.wikipedia.org/wiki/User_mode) and [kernel mode](http://en.wikipedia.org/wiki/Kernel_mode) is required in an operating system, a context switch is not necessary; a mode transition is *not* by itself a context switch. However, depending on the operating system, a context switch may also take place at this time.

**4.** (**1 punct**) ˆIn ce situat, ie pot dou˘a procese copil ale aceluias, i proces s˘a aib˘a acelas, i PID?

Neither fork() nor vfork() keep the same PID although clone() *can* in one scenario (\*a). They are all different ways to achieve roughly the same end, the creation of a *distinct* child.

clone() is like fork() but there are many things shared by the two processes and this is often used to enable threading.

vfork() is a variant of clone in which the parent is halted until the child process exits or executes another program. It's more efficient in those cases since it doesn't involve copying page tables and such. Basically, everything is shared between the two processes for as long as it takes the child to load another program.

Contrast that last option with the normal copy-on-write where memory itself is shared (until one of the processes writes to it) but the page tables that reference that memory are copied. In other words, vfork() is even *more* efficient than copy-on-write, at least for the fork-followed-by-immediate-exec use case.

But, in *most* cases, the child has a different process ID to the parent.

\*a Things become tricky when you clone() with CLONE\_THREAD. At that stage, the processes still have different identifiers but what constitutes the PID begins to blur. At the deepest level, the Linux scheduler doesn't care about processes, it schedules threads.

A thread has a thread ID (TID) and a thread group ID (TGID). The TGID is what you get from getpid().

When a thread is cloned *without* CLONE\_THREAD, it's given a new TID and it also has its TGID set to that value (i.e., a brand new PID).

*With* CLONE\_THREAD, it's given a new TID but the TGID (hence the reported process ID) remains the same as the parent so they really have the same PID. However, they can distinguish themselves by getting the TID from gettid().

There's quite a bit of trickery going on there with regard to parent process IDs and delivery of signals (both to the threads within a group and the SIGCHLD to the parent), all which can be examined from the [clone() man page](http://linux.die.net/man/2/clone).

**5.** (**1 punct**) Dou˘a procese scriu s, i citesc un fis, ier, ˆın mod sincronizat, prin urm˘atoarea secvent, a˘

de pseudo-cod:

acquire\_mutex (&m);

if (condition)

write\_to\_file (content);

else read\_from\_file(content);

release\_mutex(&m);

Care este num˘arul minim de apeluri de sistem care sunt generate ˆın secvent, a de pseudo-cod de mai sus?

**6.** (**1 punct**) La un moment dat un proces acceseaz˘a o adres˘a de memorie, f˘ar˘a a rezulta page fault. Dup˘a un timp, acceseaz˘a din nou acea adres˘a s, i rezult˘a page fault. S, tiind c˘a ˆıntre cele dou˘a accese descrise nu au existat alte accese la pagina aferent˘a, explicat, i de ce s-a produs acel page fault.

**7.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *handler de semnal* s, i *mutex*.

Nu este recomandat să se folosească mutex-uri într-un handler de semnal. Dacă se face lock pe mutex și, în programul principal, s-a făcut de asemenea, lock pe mutex, există riscul unui deadlock; handler-u de semnal întrerupe programul principal în timp ce acesta încă are ocupat mutex-ul.

**8.** (**1 punct**) Creat, i un paragraf adev˘arat, informativ s, i argumentat din domeniul sistemelor de operare care s˘a cuprind˘a ca subiecte principale not, iunile de *pagin˘a fizic˘a (frame)* s, i *fis, ier mapat ˆın memorie*.

Un fișier mapat în memorie ocupă pagini fizice (frame-uri) care sunt mapate apoi în spațiuil de adresă al procesului. Un fișier poate fi mapat de mai multe procese, caz în care paginile fizice (frame-urile) aferente pot fi partajate. Scrierile în spațiul de adresă vor ajunge în spațiul fizic și vor fi flushed doar la apeluri specifice sau la închiderea mapării.

**9.** (**1 punct**) Care sunt asocierile dintre sect, iunile de memorie de mai jos s, i programe/executabile, procese, respectiv thread-uri? (unul la unul, unul la mai multe, etc.)

text, rodata, data, bss, heap, stack

**10.** (**1 punct**) Asociat, i conceptele de mai jos cu solut, iile de virtualizare VMware Workstation, KVM, LXC: kernel development, bare-metal virtualization, modul de kernel, native virtualiza- tion, full virtualization, containers, disk image file.

**11.** (**2.5 puncte**) Un programator implementeaz˘a o baz˘a de date care va stoca obiecte de dimensiune fix˘a de 1MB. Fiecare obiect are un identificator numeric unic (ID), s, i este salvat pe disc (HDD) ca un fis, ier separat, folosind sistemul de fis, iere ext2. Num˘arul total de obiecte poate fi foarte mare, fiind limitat doar de dimensiunea HDD-ului (e.g. 10TB 10 milioane fis, iere).

Baza de date trebuie optimizat˘a astfel ˆıncˆat s˘a acceseze cˆat mai repede un fis, ier identificat cu un anumit ID.

Vi se cere s˘a sugerat, i optimiz˘ari posibile pentru a atinge acest scop. Explicat, i care sunt fac- torii care limiteaz˘a performant, a, s, i argumentat, i optimiz˘arile alese. Desenat, i o schem˘a bloc a sistemului propus.

Pentru deschiderea unui fisier e nevoie sa localizam fisierul in dentry - costul este liniar in nr. de fisiere in director in ext2. Idee de baza: ca sa reducem timpul de acces, trebuie sa ne asiguram ca subiectele sunt stocate in directoare care au un numar redus de fisiere.

Pentru a implementa baza de date, avem nevoie de doua lucruri:

* o tabela de hash care mapeaza ID-ul fisierului in directorul care il contine. Aceasta tabela va fi stocata in memorie. Presupunand ca numarul de directoare este relativ mic, dimensiunea tabelei este data de nr de fisiere \* (dim\_id + pointer\_nume\_director) ~ 8 sau 16 octeti. 10 milioane de fisiere ar ocupa numai 160MB de RAM.
* o ierarhie de directoare in care fiecare director are un numar maxim de intrari X (prestabilit).

Se pot folosi multiple structuri ierarhice cu adancime prestabilita. Se creaza astfel mai multe directoare:

* un director “direct” care contine un numar de fisiere X
* un director “indirect” care contine X alte directoare fiecare cu X fisiere
* un director dublu indirect
* un director triplu indirect etc.

Se vor folosi directoarele “directe” dupa care cele indirecte, dublu-indirecte, etc.

Cum alegem X? Timpul de acces la un fisier depinde de X si de adancimea structurii de directoare, citirea unui director dureaza: X(logXN + 1). Stiind N (e.g. 10 milioane), se poate optimiza X alegand valoarea care minimizeaza formula de mai sus.

**12.** (**2.5 puncte**) Client, ii Youtube se conecteaz˘a la servere via TCP s, i transmit o cerere care cont, ine numele fis, ierului dorit, offset-ul de ˆınceput s, i dimensiunea dorit˘a (tipic, cˆateva zeci de KB, pentru a evita download-ul cont, inutului ˆın mod inutil, dac˘a utilizatorul ˆınchide clipul). Dac˘a serverul ˆınchide conexiunea, clientul se va reconecta atunci cˆand are nevoie de urm˘atoarea secvent, a˘ de clip s, i va relua download-ul. Altfel, clientul t, ine conexiunea deschis˘a s, i doar va emite o nou˘a cerere.

Vi se cere s˘a implementat, i un server Youtube care s˘a permit˘a unui num˘ar cˆat mai mare de utilizatori s˘a priveasc˘a clipuri simultan, astfel:

*•* desenat, i o schem˘a bloc cu sistemul propus;

*•* explicat, i cum va fi tratat fiecare client (proces nou / thread / etc.) s, i motivat, i alegerea f˘acut˘a;

*•* explicat, i ce API vet, i folosi pentru a citi / scrie din sockets TCP (nonblocking, event-based, blocking?) s, i de ce;

*•* atunci cˆand cres, te num˘arul de client, i, ce resurse vor limita scalabilitatea sistemului pe care l-at, i construit? (e.g. nr. de descriptori, nr. de procese/thread-uri, ret, eaua)?

Ideea principala este de a trata o cerere de continut cat mai repede si de a inchide conexiunea – astfel numarul de clienti conectati simultan este mic si acest lucru reduce stresul asupra resurselor sistemtului (thread-uri, descriptori).

Se poate folosi un pool de thread-uri; atunci cand vine o cerere noua se aloca cererea unuia din thread-urile din pool. Folosirea pool-ului de thread-uri minimizeaza costurile de startup, si reduc costurile de switching (no tlb flush), insa toti clientii vor folosi acelasi spatiu de adresa – totusi potentialele probleme de securitate par minore (read-only video data).

Se va folosi blocking API pt. citire din sockets – e cea mai usor de folosit. Non-blocking nu prea are sens cu thread-uri. Event based la fel. Cel mai probabil reteaua va deveni un bottleneck. Din cauza ca folosim un pool de thread-uri nr de thread-uri nu e o problema, si nici cel de descriptori (presupunand ca nu se executa accept atunci cand nu se poate repartiza cererea clientului unui thread).

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, declar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

27 mai 2013

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** Comanda ”ulimit -s unlimited” stabiles, te ca dimensiunea stivei s˘a fie maxim˘a posibil˘a. De ce aceast˘a comand˘a poate fi folosit˘a pentru a ˆınlesni atacuri de tip return-to-libc pe sisteme cu suport de ASLR?

*Cu cat stiva este mai mare cu atat posibilitatile de pozitionare a acesteia in spatiul de adresa scad. Astfel, se poate face brute force mai usor.*

**2.** Care este asocierea dintre not, iunea de spat, iu de adres˘a s, i proces? Dar ˆıntre spat, iu de adres˘a s, i thread-uri?

*Fiecare proces are propriul spatiu de adresa -->( spatiu de adresa, proces ) - unu la unu*

*Threadurile unui proces partajeaza acelasi spatiu de adresa (spatiu de adresa, thread) - unul la mai multe*

**3.** Folosind mecanismul de memorie virtual˘a, se pot partaja zone din spat, iul de adres˘a din user space al unui proces cu zone din spat, iul kernel. Dat, i exemplu de situat, ie ˆın care o astfel de abordare este util˘a.

A processor in a computer running Windows has two different modes: user mode and kernel mode. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode. Many drivers run in kernel mode, but some drivers run in user mode.

When you start a user-mode application, Windows creates a process for the application. The process provides the application with a private virtual address space and a private handle table. Because an application's virtual address space is private, one application cannot alter data that belongs to another application. Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

In addition to being private, the virtual address space of a user-mode application is limited. A processor running in user mode cannot access virtual addresses that are reserved for the operating system. Limiting the virtual address space of a user-mode application prevents the application from altering, and possibly damaging, critical operating system data.

All code that runs in kernel mode shares a single virtual address space. This means that a kernel-mode driver is not isolated from other drivers and the operating system itself. If a kernel-mode driver accidentally writes to the wrong virtual address, data that belongs to the operating system or another driver could be compromised. If a kernel-mode driver crashes, the entire operating system crashes.

This diagram illustrates communication between user-mode and kernel-mode components.

**4.** De ce num˘arul de intr˘ari ˆın TLB este de ordinul sutelor? De ce nu se aloc˘a dimensiuni mai mari pentru TLB (num˘ar de intr˘ari de ordinul miilor sau zecilor de mii)?

I'd be interested to experiment with different TLB sizes, to see what effect

that has on performance. But I suspect that lack of TLB contexts mean that we

wind up flushing the TLB more often than real hardware does, and therefore a

larger TLB merely takes longer to flush.

Hardware TLBs are limited in size primarily due to the fact that increasing their sizes increases their access latency as well. but software tlb does not

suffer from that problem. so i think the size of the softtlb should be not influenced by the size of the hardware tlb.

Flushing the TLB is minimal unless we have a really really large TLB, e.g. a TLB with 1M entries. I vaguely remember that i see ~8% of the time is spent in

the cpu\_x86\_mmu\_fault function in one of the speccpu2006 workload some time ago. so if we increase the size of the TLB significantly and potential getting

rid of most of the TLB misses, we can get rid of most of the 8%. ( there are still compulsory misses and a few conflict misses, but i think compulsory

misses is not the major player here).

**5.** De ce se prefer˘a folosirea spinlock-urilor pentru regiuni critice de dimensiuni mici iar mutex- urile pentru regiuni critice de dimensiuni mari?

Deoarece spinlock-ul foloseste busy-waiting este indicat sa fie folosit pe portiuni mici unde timpul de asteptare este mic si unde folosirea unui mutex ar aduce un overhead mult prea mare. Mutex-urile sunt folosite pe portiuni critice mari deoarece costul unui busy waiting pe un timp lung e mult mai mare decat asteptarea intr-o coada pana la un unlock pe mutex.

**6.** Care este leg˘atura s, i diferent, a dintre not, iunile de ”drepturi de creare a unui fis, ier” s, i ”drep- turi de deschidere a unui fis, ier”?

**7.** Care este un avantaj al fiec˘areia dintre formele de virtualizare: full virtualization s, i paravir- tualization?

Full virtualization:

\*AVANTAJE

One of the most common reasons for implementing a full virtualization solution is for operational efficiency. It allows organizations to use existing hardware more efficiently by placing a greater load on each computer. This means that servers using full virtualization can use more of the computer’s processing and memory resources than servers running a single OS instance and a single set of services.

Another reason to use full virtualization is to facilitate desktop virtualization, in which a single PC runs more than one OS instance. There are a number of reasons to do so. This can offer support for applications that only run on a particular OS. It can also allow changes to be made to an OS and later on, revert to the original if necessary. Desktop virtualization has also proven to support better control of OSs, in order to ensure that they meet basic security requirements

\*DEZAVANTAJE

* Adds layers of technology, which increase the security management burden as it requires additional security controls.
* Combining a number of systems onto a single physical computer causes a larger impact, should a security compromise occur.
* It’s relatively easy to share information between virtualization systems, which can facilitate attack vectors, if not carefully controlled or regulated.
* The dynamic aspect of virtualized environments renders creating and maintaining the necessary security boundaries more complex.

.PARAVIRTUALIZATION

\*AVANTAJE

Performance is the most well known advantage that paravirtualization has, however with paravirtualized device drivers in a fully virtualized OS this advantage is actually getting smaller over time.

However, compared to traditional full virtualization, where the virtualization software emulates a complete computer and a completely unmodified guest operating system is run, paravirtualization has very significant performance advantages.

**8.** De ce un proces care reprezint˘a un server web are prioritate mai mare decˆat un proces care este folosit pentru ˆınmult, iri de matrice?

CPU INTENSIVE VS I/O

**9.** Procesul P foloses, te thread-uri hibride, avˆand 9 thread-uri user-level, mapate pe 3 thread- uri kernel-level (cˆate trei user-level threads pe un kernel-level thread). Fie thread-ul user-level TU1 mapat pe thread-ul kernel-level TK1. ˆIn ce mod vor fi afectate celelalte thread-uri mapate pe TK1, ˆın cazul ˆın care TU1 realizeaz˘a un acces nevalid la memorie? Dar thread-urile mapate pe celelalte thread-uri kernel?

**10.** De ce este necesar˘a operat, ia de ”Safe Remove” a dispozitivelor USB, ˆın urma copierii unor fis, iere noi pe acestea?

Obviously, yanking out a drive while it's being written to could corrupt the data. However, even if the drive isn't *actively* being written to, you could still corrupt the data. By default, most operating systems use what's called *write caching* to get better performance out of your computer. When you write a file to another drive—like a flash drive—the OS waits to actually perform those actions until it has a number of requests to fulfill, and then it fulfills them all at once (this is more common when writing small files). When you hit that eject button, it tells your OS to flush the cache—that is, make sure all pending actions have been performed—so you can safely unplug the drive without any data corruption.

ˆIn conformitate cu ghidul de etic˘a al Catedrei de Calculatoare, declar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

24 mai 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Precizat, i o diferent, a˘ ˆıntre un thread s, i un proces.

Procesele au propriul spatiu de adresa pe cand threadurile unui proces sharuiesc acelasi spatiu de adresa.

**Processes are the abstraction of running programs**: A binary image, virtualized memory, various kernel resources, an associated security context, and so on. **Threads are the unit of execution in a process**: A virtualized processor, a stack, and program state. Put another way, processes are running binaries and threads are the smallest unit of execution schedulable by an operating system's process scheduler.  
  
A process *contains* one or more threads.

**Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.**

**2.** (**7 puncte**) Care este leg˘atura ˆıntre memorie virtual˘a s, i spat, iu de swap?

Memoria virtuală este memoria pe care o vede un proces care rulează în sistem. Pentru că în RAM nu încape întreg spațiul adresat de memoria virtuală (~4GB pentru procesoarele pe 32 de biți) extindem memoria cu un cache pe hard disk, pe care îl numim swap.

**Virtual memory is a combination of RAM and disk space that running processes can use.**

**Swap space is the portion of virtual memory that is on the hard disk, used when RAM is full.**

**3.** (**7 puncte**) Dat, i exemplu de dispozitiv de tip caracter. De ce este acesta un dispozitiv de tip caracter?

Tastatura este un dispozitiv de tip caracter pentru ca trasferul de date se realizeaza caracter cu caracter.

**A character device is any device that can have streams of characters read from or written to it. A character device has a character device driver associated with it that can be used for a device such as a line printer that handles one character at a time. However, character drivers are not limited to performing I/O a single character at a time (despite the name ``character'' driver). For example, tape drivers frequently perform I/O in 10K chunks. A character device driver can also be used where it is necessary to copy data directly to or from a user process. Because of their flexibility in handling I/O, many drivers are character drivers. Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.**

(si despre block character, sa-l avem acolo)

exemplu: hard disk

A device, such as a magnetic tape drive or disk drive, that conveys data in blocks through the buffer management code.

A block device is one that is designed to operate in terms of the block I/O supported by Digital UNIX. It is accessed through the buffer cache. A block device has an associated block device driver that performs I/O by using file system block-sized buffers from a buffer cache supplied by the kernel. Block device drivers are particularly well-suited for disk drives, the most common block devices.

**4.** (**7 puncte**) Descriet, i, ˆın pseudocod sau literal, un scenariu ˆın care are loc un atac de tipul buffer overflow. Precizat, i ˆın ce condit, ii se realizeaz˘a acest atac.

exemplU, avem urmatorul program:

#include <stdio.h>

#include <string.h>

int main(void) {

char buff[15];

int pass = 0;

printf("\n Enter the password : \n");

gets(buff);

if(strcmp(buff, "thegeekstuff")) {

printf ("\n Wrong Password \n");

} else {

printf ("\n Correct Password \n");

pass = 1; }

if(pass) { /\* Now Give root or admin rights to user\*/ printf ("\n Root privileges given to the user \n"); } return 0; }

Daca rulam programul cu parola: thegeekstuff, se produce ceea ce ne asteptam, si primim drepturi de root.

Acest program insa are posibilitatea de a produce buffer overflow. Functia gets() nu verifica marginile unde scrie si poate scrie string-uri de lungime mai mare decat marimea celui in care scrie el de fapt. Asadar, daca atacatorul da o parola mai lunga decat marimea buffer-ului, acesta poate sa ajunga sa scrie peste zona de memorie a variabilei "pass", devenind ceva diferit de 0 => drepturi de root pentru atacator.

**5.** (**7 puncte**) Un fis, ier are dou˘a link-uri hard (a.txt s, i b.txt). Ce se ˆıntˆampl˘a dac˘a s, tergem unul dintre link-uri (rm a.txt)?

When you create a second, third, fourth, etc link, the counter is incremented (increased ) each time by one. When you delete (rm) a link the counter is decremented ( reduced ) by one. If the link counter reaches 0 the filesystem removes the inode and marks the space as available for use.

In short, as long as you do not delete the last link the file will remain.

Edit: The file will remain *even if the last link is removed*. This is one of the ways to ensure security of data contained in a file is not accessible to any other process. Removing the data from the filesystem completely is done only if the data has 0 links to it as given in its metadata and is not being used by any process.

raspunzi cam la orice intrebare soft links vs hard links:

**Underneath the file system files are represented by inodes (or is it multiple inodes not sure)**

**A file in the file system is basically a link to an inode.**  
**A hard link then just creates another file with a link to the same underlying inode.**

**When you delete a file it removes one link to the underlying inode. The inode is only deleted (or deletable/over-writable) when all links to the inode have been deleted.**

**A symbolic link is a link to another name in the file system.**

**Once a hard link has been made the link is to the inode. deleting renaming or moving the original file will not affect the hard link as it links to the underlying inode. Any changes to the data on the inode is reflected in all files that refer to that inode.**

**Note: Hard links are only valid within the same File System. Symbolic links can span file systems as they are simply the name of another file.**

**6.** (**10 puncte**) De ce este necesar˘a prezent, a bitului setuid (suid) pe executabilul /usr/bin/passwd?

*Pentru a executa passwd user-ul nu are neaparat nevoie de privilegii de root (ex: vrea sa isi schimbe parola). Cu toate astea, procesul are nevoie de privilegii pentru a modifica fisierul /etc/passwd. Aici intervine bitul setuid care seteaza effective user id-ul la owner-ul executabilului - root.*

**7.** (**10 puncte**) ˆIntr-un sistem ruleaz˘a la un moment dat 100 de procese. Cˆate tabele de pagini sunt alocate? Justificat, i.

By giving each process its own page table, every process can pretend that it has access to the entire address space available from the processor. It doesn't matter that two processes might use the same address, since different page-tables for each process will map it to a different frame of physical memory. Every modern operating system provides each process with its own address space like this.

*Over time, physical memory becomes fragmented, meaning that there are "holes" of free space in the physical memory. Having to work around these holes would be at best annoying and would become a serious limit to programmers. For example, if you malloc 8 KiB of memory; requiring the backing of two 4 KiB frames, it would be a huge inconvience if those frames had to be contiguous (i.e., physically next to each other). Using virutal-addresses it does not matter; as far as the process is concerned it has 8 KiB of contiguous memory, even if those pages are backed by frames very far apart. By assigning a virtual address space to each process the programmer can leave working around fragmentation up to the operating system.*

**8.** (**10 puncte**) ˆIn urma a dou˘a apeluri accept() ˆın codul unui server sunt creat, i doi socket, i care au aceeas, i adres˘a IP s, i port. Cum diferent, iaz˘a sistemul de operare socketul c˘aruia ˆıi va fi livrat un pachet dat?

*Socketii sunt identificati printre altele de urmatoarele atribute: ip sursa, port sursa, ip destinatie, port destinatie. In cazul de fata, desi doua dintre atribute vor fi identice pentru ambii socketi celelalte doua vor putea identifica peer-ul asociat (port sursa, ip sursa).*

**9.** (**10 punct**) De ce este de preferat folosira unei cuante de timp mai mari pentru planificatorul de procese al unui sistem de tip **server**, s, i a unei cuante de timp mai mici pentru planificatorul de procese al unui sistem de tip **laptop**?

**10.** (**10 puncte**) Apelul lseek() actualizeaz˘a cursorul de fis, ier. De ce aceast˘a actualizare nu produce nici o modificare a inode-ului fis, ierului?

**If it were associated with the inode, then you would not be able to have multiple processes accessing a file in a sensible manner, since all accesses to that file by one process would affect other processes.**

Thus, a single process could have track many different file positions as it has file descriptors for a given file.

Per the [lseek docs](http://www.kernel.org/doc/man-pages/online/pages/man2/lseek.2.html), the file position is associated with the open file pointed to by a file descriptor, i.e. the thing that is handed to your by [open](http://www.kernel.org/doc/man-pages/online/pages/man2/open.2.html). Because of functions like dup and fork, multiple*descriptors* can point to a single *description*, but it's the description that holds the location cursor.

|  |
| --- |
| File position is associated with an *open file description*, not a file descriptor. Many different file descriptors can refer to the same open file description, due to fork, dup, etc. |

**11.** (**15 puncte**) Avet, i la dispozit, ie un sistem cu mai multe core-uri s, i vret, i s˘a dezvoltat, i o bibliotec˘a de video transcoding (CPU-bound) pentru stream-uri video dintr-un fis, ier. Presupu- nem c˘a avet, i detaliile unui algoritm de transcoding paralel. Acest algoritm permite efectuarea operat, iei de transcoding separat pe blocuri diferite din stream (transcode\_block()). Este

ˆıns˘a nevoie, la finalul trascodingului unui bloc de o operat, ie de unificare la final pentru dou˘a blocuri adiacente (merge\_adjacent\_stream\_blocks()); ˆın aceast˘a parte de unificare se ,,li- pesc”, respectiv, p˘art, ile de ˆınceput s, i sfˆars, it ale celor dou˘a blocuri. Blocurile sunt citite s, i scrise dintr-un/ˆıntr-un fis, ier. Dimensiunea blocului este prestabilit˘a.

Care vor fi principiile de proiectare a bibliotecii? Vet, i folosi thread-uri sau procese? Cˆate? Ce mecanisme de comunicare s, i sincronizare vet, i folosi ˆın cadrul bibliotecii? Care sunt factorii de overhead din implementare?

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

3 iunie 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Precizat, i dou˘a entit˘at, i diferite pe care le poate referi un descriptor de fis, ier ˆın

Linux.

**Entati diferite ce le poate referi: standard input/output/error, fisier, pipe, resursa input/output**

if there are 100 files opened in your OS then there will be 100 entries in OS (somewhere in kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. So it is just an integer number that uniquely represents an opened file in operating system. If your process opens 10 files then your Process table will have 10 entries for file descriptors.

Similarly when you open a network socket, it is also represented by an integer and it is called Socket Descriptor.

**2.** (**7 puncte**) ˆIn urma unui buffer overflow adresa de retur este suprascrisa˘. C˘atre ce zon˘a poate pointa adresa suprascris˘a pentru a genera un atac, ˆın cazul ˆın care sistemul are DEP (*Data Execution Prevention* )? Aleget, i dintre *text*, *stack*, *data*. Justificat, i.

**3.** (**7 puncte**) De ce este util˘a paginarea ierarhic˘a?

**The inverted page table keeps a listing of mappings installed for all frames in physical memory. However, this could be quite wasteful. Instead of doing so, we could create a page table structure that contains mappings for virtual pages. It is done by keeping several page tables that cover a certain block of virtual memory. For example, we can create smaller 1024-entry 4K pages that cover 4M of virtual memory.**

**This is useful since often the top-most parts and bottom-most parts of virtual memory are used in running a process - the top is often used for text and data segments while the bottom for stack, with free memory in between. The multilevel page table may keep a few of the smaller page tables to cover just the top and bottom parts of memory and create new ones only when strictly necessary.**

**Now, each of these smaller page tables are linked together by a master page table, effectively creating a tree data structure. There need not be only two levels, but possibly multiple ones.**

**A virtual address in this schema could be split into three parts: the index in the root page table, the index in the sub-page table, and the offset in that page.**

**Multilevel page tables are also referred to as hierarchical page tables.**

**Alt raspuns:**

You will appreciate the space optimization of multi-level page tables when we go into the 64-bit address space.

Assume you have a 64-bit computer ( which means **64 bit virtual address space** ), which has **4KB pages** and **4 GB** of physical memory. If we have a single level page table as you suggest, then it should contain one entry per virtual page per process.

One entry per virtual page – 264 addressable bytes / 212 bytes per page = **252 page table entries**

One page table entry contains: Access control bits ( Bits like Page present, RW etc ) + Physical page number

4 GB of Physical Memory = **232 bytes**.

232 bytes of memory/212 bytes per page = **220 physical pages**

**20 bits required for physical page number.**

So each page table entry is approx **4 bytes**. ( 20 bits physical page number is approx 3 bytes and access control contributes 1 byte )

Now, Page table Size = 252 page table entries \* 4 bytes = **254 bytes** ( **16 petabytes** ) !

**16 petabytes per process** is a very very huge amount of memory.

Now, if we page the pagetable too, ie if we use multi level page tables we can magically bring down the memory required to as low a single page. ie **just 4 KB**.

Now, we shall calculate how many levels are required to squeeze the page table into just 4 KB. 4 KB page / 4 bytes per page table entry = **1024 entries**. **10 bits of address space required.** i.e **52/10**ceiled is 6. ie **6 levels of page table can bring down the page table size to just 4KB.**

6 level accesses are definitely slower. But I wanted to illustrate the space savings out of multi level page tables.

**4.** (**7 puncte**) Cu ce difer˘a apelul fork() din Linux fat, a˘ Windows?

de apelul CreateProcess() din

In Windows when you call CreateProcess() it does just that. It creates a new "clean" process ready to run what you say.

In Linux fork() clones the process's page table and sets the child process's pages to copy-on-write--so no, there is no measurable "waste of time and resources". And if you're *really* that concerned about performance, you can use vfork() instead, which doesn't even bother to clone the page table. In fact, if you'd done your homework at all you would know that spawning new processes on Linux is quite significantly *faster* than on Windows.

**5.** (**7 puncte**) Cˆand folosim mutex-uri ˆın loc de operat, ii atomice pentru asigurarea accesului serial la date?

**6.** (**10 puncte**) Fie instruct, iunea:

a = b;

ˆIn ce situat, ie instruct, iunea genereaz˘a dou˘a page fault-uri f˘ar˘a a conduce la terminarea proce- sului curent?

DEFINITIE:

**A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory.**

That's not entirely correct, as explained later in the same article (*Minor page fault*). There are soft page faults, where all the kernel needs to do is add a page to the working set of the process. Here's a table from the Windows Internals book (I've excluded the ones that result in an access violation):

* Reason for Fault - *Result*
* Accessing a page that isn’t resident in memory but is on disk in a page file or a mapped file -*Allocate a physical page, and read the desired page from disk and into the relevant working set*
* Accessing a page that is on the standby or modified list - *Transition the page to the relevant process, session, or system working set*
* Accessing a demand-zero page - *Add a zero-filled page to the relevant working set*
* Writing to a copy-on-write page - *Make process-private (or session-private) copy of page, and replace original in process or system working set*

Page faults can occur for a variety of reasons, as you can see above. Only one of them has to do with reading from the disk. If you try to allocate a block from the heap and the heap manager allocates new pages, then accesses those pages, you'll get a demand-zero page fault. If you try to hook a function in kernel32 by writing to kernel32's pages, you'll get a copy-on-write fault because those pages are silently being copied so your changes don't affect other processes.

Now to answer your question more specifically: Process Hacker only seems to have page faults when updating its service information - that is, when it calls [EnumServicesStatusEx](http://msdn.microsoft.com/en-us/library/ms682640.aspx), which RPCs to the SCM (services.exe). My guess is that in the process, a lot of memory is being allocated, leading to demand-zero page faults (the service information requires several pages to store, IIRC).

**Several reasons:**  
1. Processes are created by memory mapping the code sections from the file. Reading sections of a memory mapped file that aren't yet in memory causes page faults, so each process will at least have the page faults of reading in its own executable and any DLLs whose code wasn't yet in memory. Other memory mapped files used by a process will also cause page faults.  
2. When a process requests memory with VirtualAlloc, no physical frames are actually committed to the process until the allocated pages are 'touched' for the first time. This also causes page faults.  
3. Even when memory is not full, Windows will trim infrequently used pages from the process' working set and lazily page them out to disk. This enables Windows to better respond to sudden demands for large amounts of memory. When a process attempts to access such a page, it causes a page fault. In situations where memory consumption is low enough, the page will probably still be in memory so no disk read is necessary, but a page fault is still triggered. This is called a soft page fault.

**7.** (**10 puncte**) ˆIn ce situat, ie folosit, i apeluri de tip read/write ˆın loc de maparea fis, ierului ˆın memorie?

It really depends on what you're trying to do. If all you need to do is hop to a known offset and read out a small tag, read() may be faster (mmap() has to do some rather complex internal accounting). If you are planning on copying out all 200mb of the MP3, however, or scanning it for some tag that may appear at an unknown offset, then mmap() is likely a faster approach.

**read() on the other hand involves an extra memory-to-memory copy, and can thus be inefficient for large I/O operations, but is simple, and so the fixed overhead is relatively low. In short, use mmap()for large bulk I/O, and read() or pread() for one-off, small I/Os.**

**8.** (**10 puncte**) ˆIn cadrul unei conexiuni TCP, transmit, a˘torul realizeaz˘a apelul:

send(s, send\_buffer, 5000, 0); /\* send 5000 bytes \*/

iar receptorul realizeaz˘a apelul

recv(s, recv\_buffer, 7000, 0); /\* receive 7000 bytes \*/

Cˆat, i octet, i va primi receptorul (la ˆıntoarcerea din apelul recv)?

*Receptorul poate primi oricati octeti intre 1 si 5000, presupunand ca niciun alt send nu s-a realizat sau se va realiza din partea tansmitatorului. In functie de congesia retelei, a incarcarii bufferelor din kernel si cele de pe placa de retea aceasta valoare variaza intre 1 si 5000. Daca transmitatorul inchide conexiunea inainte ca datele sa fie transmise, apelul recv se va intoarce cu -1.*

**9.** (**10 punct**) Un thread foloses, te malloc pentru a aloca memorie. Precizat, i un set de pas, i (ˆın pseudocod) ˆın care alt thread al aceluias, i proces acceseaz˘a zona de memorie alocat˘a de primul thread.

* malloc() and free() are not thread-safe functions. You need to protect the calls to those functions with a mutex.
* You need to protect all shared variables with a mutex as well. You can use the same one as you use for malloc/free, one per variable.
* You need to declare variables shared between several threads as volatile, to prevent dangerous optimizer bugs on some compilers. Note that this is no replacement for mutex guards.
* Are the buffers arrays, or two-dimensional arrays (like arrays of C strings)? You have declared all buffers as potential two-dimensional arrays, but you never allocate the inner-most dimension.
* Never typecast the result of malloc in C. Read [this](http://c-faq.com/malloc/mallocnocast.html) and [this](http://stackoverflow.com/questions/1565496/specifically-whats-dangerous-about-casting-the-result-of-malloc).
* free(bufferaction), not free(&bufferaction).
* Initialize all pointers to NULL explicitly. After free(), make sure to set the pointer to NULL. Before the memory is accessed by either thread, make sure to check the pointer against NULL.

**10.** (**10 puncte**) Un sistem de fis, iere ext2 poate folosi fis, iere cu dimensiune de pˆan˘a la 16GB. Ce limiteaz˘a dimensiunea maxim˘a a fis, ierelor?

There are various limits imposed by the on-disk layout of ext2. Other limits are imposed by the current implementation of the kernel code. Many of the limits are determined at the time the filesystem is first created, and depend upon the block size chosen. The ratio of inodes to data blocks is fixed at filesystem creation time, so the only way to increase the number of inodes is to increase the size of the filesystem.

The 2TiB file size is limited by the i\_blocks value in the inode which indicates the number of 512-bytes sector rather than the actual number of ext2 blocks allocated.

This limit was also overcome ages ago by the use of a flag in the inode that indicates that the i\_blocks value is, in fact, in units of block size rather than 512 bytes. The triple indirect block structure though, can only address just over 4 TiB using a 4k block size.

**11.** (**15 puncte**) Dorim s˘a implement˘am un proxy server pentru conexiuni web. Un proxy ser- ver serves, te cereri din cache-ul local dac˘a paginile web se g˘asesc ˆın cache; altfel face cereri c˘atre serverul web destinat, ie, obt, ine pagina s, i apoi o cache-uies, te. Facem urm˘atoarele presupuneri:

*•* Paginile cerute sunt, ˆın mare parte, de dimensiuni mici (ordinul kilooctet, ilor).

*•* Paginile se modific˘a greu; nu este nevoie s˘a v˘a gˆandit, i la expirarea paginilor ˆın cache.

*•* Se poate folosi pentru caching atˆat memorie cˆat s, i spat, iu pe disc, ambele limitate.

*•* Exist˘a un num˘ar mare de cereri pe secund˘a pe care le primes, te proxy serverul.

Ce tehnologii vet, i folosi ˆın proiectarea proxy serverului? (operat, ii asincrone, multiplexare, multithreading, multiproces, etc.). Justificat, i alegerea.

Ce politic˘a de ˆınlocuire a paginilor ˆın cache vet, i folosi?

Ce pagini vet, i plasa ˆın cache-ul de memorie s, i ce pagini vet, i plasa ˆın cache-ul de pe disc? Cum vet, i asigura accesul sincronizat/consecvent/coerent la datele din cache?

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

5 iunie 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Cˆand are loc un flush de TLB (*Translation Lookaside Buffer* )?

Are loc la orice schimbare asupra structurii paginilor.

If another processor could also be affected by a page table write (because of shared memory, or multiple threads from the same process), you must also flush the TLBs on those 1processors.

**2.** (**7 puncte**) Ce cont, in blocurile de date aferente unui director?

**3.** (**7 puncte**) De ce consider˘am tastatura un dispozitiv de tip caracter?

a mai fost

**4.** (**7 puncte**) De ce este timpul de creare a unui thread al aceluias, i proces mai mic decˆat timpul de creare a unui proces?

* Less time to create a new thread than a process, because the newly created thread uses the current process address space.

**5.** (**7 puncte**) Cˆate tabele de descriptori de fis, iere are un proces cu 10 thread-uri?

una!

there is only one file descriptor table per process, and it's shared among all the threads.

The file descriptors are shared between the threads. If you want "thread specific" offsets, why not have each thread use a different file descriptor (open(2) multiple times) ?

**6.** (**10 puncte**) Care este o leg˘atur˘a ˆıntre planificatorul de procese s, i sistemul de ˆıntreruperi?

*Sistemul de intreruperi furnizeaza intreruperea de ceas, ce determina planificatorul sa ia decizii privind procesul ce va rula in urmatoarea cuanta de timp.*

**7.** (**10 puncte**) ˆIn cadrul unei conexiuni TCP un client trimite c˘atre un server mesaje ˆıntr-o bucl˘a:

while (1) {

send(s, buffer, 8192, 0); /\* send 8192 bytes \*/

}

La un moment dat, apelul send se blocheaz˘a (pentru o perioad˘a de timp). Care este o cauz˘a posibil˘a pentru aceast˘a blocare?

*Toate bufferele intre transmitator si destinatar s-au umplut ori reteaua este congestionata. Prin buffere ma refer la atat la bufferele din kernel cat si la cele de pe placile de retea.*

**8.** (**10 puncte**) Descriet, i o situat, ie ˆın care un buffer overflow pe un array aflat ˆın zona de date globale conduce la un exploit.

The principle of exploiting a buffer overflow is to overwrite parts of memory which aren't supposed to be overwritten by arbitrary input and making the process execute this code. To see how and where an overflow takes place, lets take a look at how memory is organized. A page is a part of memory that uses its own relative addressing, meaning the kernel allocates initial memory for the process, which it can then access without having to know where the memory is physically located in RAM. The processes memory consists of three sections:

- code segment, data in this segment are assembler instructions that the processor executes. The code execution is non-linear, it can skip code, jump, and call functions on certain conditions. Therefore, we have a pointer called EIP, or instruction pointer. The address where EIP points to always contains the code that will be executed next.

- data segment, space for variables and dynamic buffers

- stack segment, which is used to pass data (arguments) to functions and as a space for variables of functions. The bottom (start) of the stack usually resides at the very end of the virtual memory of a page, and grows down. The assembler command PUSHL will add to the top of the stack, and POPL will remove one item from the top of the stack and put it in a register. For accessing the stack memory directly, there is the stack pointer ESP that points at the top (lowest memory address) of the stack.

’Lets assume that we exploit a function like this:

void lame (void) { char small[30]; gets (small); printf("%s\n", small); } main() { lame (); return 0; } Compile and disassemble it: # cc -ggdb blah.c -o blah /tmp/cca017401.o: In function `lame': /root/blah.c:1: the `gets' function is dangerous and should not be used. # gdb blah /\* short explanation: gdb, the GNU debugger is used here to read the binary file and disassemble it (translate bytes to assembler code) \*/ (gdb) disas main Dump of assembler code for function main: 0x80484c8 : pushl %ebp 0x80484c9 : movl %esp,%ebp 0x80484cb : call 0x80484a0 0x80484d0 : leave 0x80484d1 : ret (gdb) disas lame Dump of assembler code for function lame: /\* saving the frame pointer onto the stack right before the ret address \*/ 0x80484a0 : pushl %ebp 0x80484a1 : movl %esp,%ebp /\* enlarge the stack by 0x20 or 32. our buffer is 30 characters, but the memory is allocated 4byte-wise (because the processor uses 32bit words) this is the equivalent to: char small[30]; \*/ 0x80484a3 : subl $0x20,%esp /\* load a pointer to small[30] (the space on the stack, which is located at virtual address 0xffffffe0(%ebp)) on the stack, and call the gets function: gets(small); \*/ 0x80484a6 : leal 0xffffffe0(%ebp),%eax 0x80484a9 : pushl %eax 0x80484aa : call 0x80483ec 0x80484af : addl $0x4,%esp /\* load the address of small and the address of "%s\n" string on stack and call the print function: printf("%s\n", small); \*/ 0x80484b2 : leal 0xffffffe0(%ebp),%eax 0x80484b5 : pushl %eax 0x80484b6 : pushl $0x804852c 0x80484bb : call 0x80483dc 0x80484c0 : addl $0x8,%esp /\* get the return address, 0x80484d0, from stack and return to that address. you don't see that explicitly here because it is done by the CPU as 'ret' \*/ 0x80484c3 : leave 0x80484c4 : ret End of assembler dump. 3a. Overflowing the program # ./blah xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # ./blah xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Segmentation fault (core dumped) # gdb blah core (gdb) info registers eax: 0x24 36 ecx: 0x804852f 134513967 edx: 0x1 1 ebx: 0x11a3c8 1156040 esp: 0xbffffdb8 -1073742408 ebp: 0x787878 7895160

EBP is 0x787878, this means that we have written more data on the stack than the input buffer could handle. 0x78 is the hex representation of 'x'. The process had a buffer of 32 bytes maximum size. We have written more data into memory than allocated for user input and therefore overwritten EBP and the return address with 'xxxx', and the process tried to resume execution at address 0x787878, which caused it to get a segmentation fault.

**9.** (**10 punct**) Fie secvent, a de instruct, iuni de mai jos:

printf("%d", \*a);

\*a = 42;

unde a este un pointer la un ˆıntreg (int \*). Dat, i exemplu de situat, ie ˆın care prima instruct, iune

(printf) NU cauzeaz˘a page fault, dar a doua (\*a = 42) cauzeaz˘a page fault.

**10.** (**10 puncte**) Un set de thread-uri lucreaz˘a cu o structur˘a de tip list˘a dublu ˆınl˘ant, uit˘a. Unele thread-uri modific˘a lista (adaug˘a, s, terg elemente), altele doar parcurg lista. De ce trebuie asigurat accesul exclusiv la list˘a pentru ambele tipuri de thread-uri, nu doar pentru cele care modific˘a lista?

**11.** (**15 puncte**) Dorim s˘a implement˘am o bibliotec˘a de tip engine de baze de date. Aceast˘a bibliotec˘a va oferi un API de ad˘augare, s, tergere, inserare, modificare elemente ˆın baza de date s, i va realiza s, i stocarea fiec˘arei baze de date ˆıntr-un fis, ier pe disc. Un program care va folosi biblioteca va putea s˘a stocheze informat, ii ˆıntr-o baz˘a de date dintr-un fis, ier pe disc ˆıntr-un format intern. Biblioteca trebuie s˘a fie thread safe. Trebuie ca operat, iile executate ˆın thread- uri diferite ale procesului s˘a ment, in˘a datele coerente.

Definit, i schematic API-ul pe care ˆıl va expune biblioteca: structuri de date s, i funct, ionalit˘at, i expuse ca interfat, a˘ pentru programul ce va folosi biblioteca. Gˆandit, i-v˘a doar la interfat, a expus˘a nu la internele implement˘arii.

Cum vet, i asigura ˆın cadrul implement˘arii bibliotecii, ˆın mod eficient, partea de thread safety? Cum propunet, i s˘a asigurat, i o vitez˘a bun˘a de lucru cu fis, ierul de baz˘a de date s, i ˆın acelas, i timp

s˘a oferit, i o asigurare cˆat mai bun˘a c˘a datele ajung pe disc?

Cum vet, i implementa partea de tranzact, ie? Adic˘a un set de operat, ii s˘a fie executate atomic ˆın cadrul bibliotecii.

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

8 iunie 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Un apel mmap() rezerv˘a 16 pagini de memorie virtual˘a. Cˆate pagini de memorie fizic˘a aloc˘a apelul?

**2.** (**7 puncte**) De ce este dezavantajoas˘a folosirea unei cuante de timp prea mari pentru planificarea proceselor?

**3.** (**7 puncte**) Cu ce difer˘a un spinlock de un mutex?

## Mutex Vs Spinlock

|  |  |  |
| --- | --- | --- |
| Criteria | Mutex | Spinlock |
| Mechanism | Test for lock.  If available, use the resource  If not, go to wait queue | Test for lock.  If available, use the resource.  If not, loop again and test the lock till you get the lock |
| When to use | Used when putting process is not harmful like user space programs.  Use when there will be considerable time before process gets the lock. | Used when process should not be put to sleep like Interrupt service routines.  Use when lock will be granted in reasonably short time. |
| Drawbacks | Incur process context switch and scheduling cost. | Processor is busy doing nothing till lock is granted, wasting CPU cycles. |

**The Theory**

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

**The Problem**

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time. If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept by far and it might even exceed the time the thread would have wasted by constantly polling on a spinlock. On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

**The Solution**

Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. IOW, a spinlock wastes only CPU time on those systems for no real benefit. If the thread was put to sleep instead, another thread could have ran at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.

On a multi-core/multi-CPU systems, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput.

**The Practice**

Since very often programmers cannot know in advance if mutexes or spinlocks will be better (e.g. because the number of CPU cores of the target architecture is unknown), nor can operating systems know if a certain piece of code has been optimized for single-core or multi-core environments, most systems don't strictly distinguish between mutexes and spinlocks. In fact, most modern operating systems have hybrid mutexes and hybrid spinlocks. What does that actually mean?

A hybrid mutex behaves like a spinlock at first on a multi-core system. If a thread cannot lock the mutex, it won't be put to sleep immediately, since the mutex might get unlocked pretty soon, so instead the mutex will first behave exactly like a spinlock. Only if the lock has still not been obtained after a certain amount of time (or retries or any other measuring factor), the thread is really put to sleep. If the same code runs on a system with only a single core, the mutex will not spinlock, though, as, see above, that would not be beneficial.

A hybrid spinlock behaves like a normal spinlock at first, but to avoid wasting too much CPU time, it may have a back-off strategy. It will usually not put the thread to sleep (since you don't want that to happen when using a spinlock), but it may decide to stop the thread (either immediately or after a certain amount of time) and allow another thread to run, thus increasing chances that the spinlock is unlocked (a pure thread switch is usually less expensive than one that involves putting a thread to sleep and waking it up again later on, though not by far).

**Summary**

If in doubt, use mutexes, they are usually the better choice and most modern systems will allow them to spinlock for a very short amount of time, if this seems beneficial. Using spinlocks can sometimes improve performance, but only under certain conditions and the fact that you are in doubt rather tells me, that you are not working on any project currently where a spinlock might be beneficial. You might consider using your own "lock object", that can either use a spinlock or a mutex internally (e.g. this behavior could be configurable when creating such an object), initially use mutexes everywhere and if you think that using a spinlock somewhere might really help, give it a try and compare the results (e.g. using a profiler), but be sure to test both cases, a single-core and a multi-core system before you jump to conclusions (and possibly different operating systems, if your code will be cross-platform).

**4.** (**7 puncte**) Dat, i exemplu de apel de sistem care poate conduce la o schimbare de context

ˆıntre procese. Explicat, i inclusiv ˆın ce condit, ii se produce schimbarea de context.

* **Interrupts** - When the CPU is interrupted to return data from a disk read.

schimbare de context la apel de sistem read file

**5.** (**7 puncte**) Apelurile accept() s, i recv() au o sintax˘a de forma:

accept(sockfd1, ...)

recv(sockfd2, ...)

unde sockfd1 s, i sockfd2 sunt descriptori de socket. Cu ce difer˘a cei doi socket, i?

*sockfd1 - socket de tip listen asupra caruia se poate efectua doar operatia de accept deoarece campurile port sursa si ip sursa nu sunt completate*

*sockfd2 - socket normal cu ajutorul caruia se realizeaza comunicatia. Caracterizat de atributele: ip + port sursa, ip + port destinatie*

**6.** (**10 puncte**) ˆIntr-un executabil sunt definite sect, iunile text (codul programului), data (variabile globale) s, i rodata (variabile read-only). Care sect, iuni vor fi partajate de dou˘a procese pornite separat din acest executabil?

**7.** (**10 puncte**) Ignorˆand cˆampurile de tip timestamp din cadrul unui inode, dat, i un exemplu de apel de sistem de lucru cu fis, iere (din forma open(), read(), write(), seek(), close(), chmod(), stat() etc.) care modific˘a un inode s, i altul care nu modific˘a un inode.

*write() - modifica inode-ul aferent fisierului deoarece se poate modifica dimensiunea sau, mai exact, numarul de block-uri din care este alcatuit fisierul.*

*seek() - nu modifica inode-ul deoarece cursorul de fisier, ca si entitate, apartine de catre structura ce defineste un fisier deschis si nu de file control block - FCB - inode.*

**8.** (**10 puncte**) Un sistem are suport DEP (*Data Execution Prevention* ) dar nu are suport ASLR (*Address Space Layout Randomization* ). Precizat, i cum se face un atac de tipul *return- to-libc*. Cum se obt, ine adresa/adresele necesare?

[Address space layout randomization](http://en.wikipedia.org/wiki/Address_space_layout_randomization) (ASLR) makes this type of attack extremely unlikely to succeed on 64-bit machines as the memory locations of functions are random. For 32-bit systems ASLR provides little benefit since there are only 16 bits available for randomization, and they can be defeated by brute force in a matter of minutes

# DEP effectiveness (without ASLR)

In a previous blog post series we went into detail on what DEP is and how it works[[part 1](http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx), [part 2](http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-2.aspx)]. In summary, the purpose of DEP is to prevent attackers from being able to execute data as if it were code. This stops an attacker from being able to directly execute code from the stack, heap, and other non-code memory regions. As such, exploitation techniques like heap spraying (of shellcode) or returning into the stack are not immediately possible.

The effectiveness of DEP hinges on the attacker not being able to 1) leverage code that is already executable or 2) make the attacker's data become executable (and thus appear to be code). On platforms without ASLR (that is, versions of Windows prior to Windows Vista), it is often straightforward for an attacker to find and leverage code that exists in modules (DLLs and EXEs) that have been loaded at predictable locations in the address space of a process. Return-oriented programming (ROP) is perhaps the most extensive example of how an attacker can use code from loaded modules in place of (or as a stepping stone to) their shellcode [3,1]. In addition to loaded modules, certain facilities (such as Just-In-Time compilers) can allow an attacker to generate executable code with partially controlled content which enables them to embed shellcode in otherwise legitimate instruction streams ("JIT spraying")[2].

The fact that modules load at predictable addresses without ASLR also makes it possible to turn the attacker's data into executable code. There are a variety of ways in which this can be accomplished, but the basic approach is to use code from loaded modules to invoke system functions like VirtualAlloc or VirtualProtect which can be used to make the attacker's data become executable.

**Summary**: DEP breaks exploitation techniques that attackers have traditionally relied upon, but DEP without ASLR is not robust enough to prevent arbitrary code execution in most cases.

# ASLR effectiveness (without DEP)

Attackers often make assumptions about the address space layout of a process when developing an exploit. For example, attackers will generally assume that a module will be loaded at a predictable address or that readable/writable memory will exist at a specific address on all PCs. ASLR is designed to break these assumptions by making the address space layout of a process unknown to an attacker who does not have local access to the machine. This prevents an attacker from being able to directly and reliably leverage code in loaded modules.

The effectiveness of ASLR hinges on the entirety of the address space layout remaining unknown to the attacker. In some cases memory may be mapped at predictable addresses across PCs despite ASLR. This can happen when DLLs or EXEs load at predictable addresses because they have not opted into ASLR via the /DYNAMICBASE linker flag. Prior to Internet Explorer 8.0 it was also possible for attackers to force certain types of .NET modules to load at a predictable address in the context of the browser[6]. Attackers can also use various address space spraying techniques (such as heap spraying or JIT spraying) to place code or data at a predictable location in the address space.

In cases where the address space is initially unpredictable an attacker can attempt to discover the location of certain memory regions through the use of an *address space information disclosure* or through *brute forcing*[5]. An address space information disclosure occurs when an attacker is able to coerce an application into leaking one or more address (such as the address of a function inside a DLL). For example, this can occur if an attacker is able to overwrite the NUL terminator of a string and then force the application to read from the string and provide the output back to the attacker [4]. The act of reading from the string will result in adjacent memory being returned up until a NUL terminator is encountered. This is just one example; there are many other forms that address space information disclosures can take.

Brute forcing, on the other hand, can allow an attacker to try their exploit multiple times against all of the possible addresses where useful code or data may exist until they succeed. Brute forcing attacks, while possible in some cases, are traditionally not practical when attacking applications on Windows because an incorrect guess will cause the application to terminate. Applications that may be subjected to brute force attacks (such as Windows services and Internet Explorer) generally employ a restart policy that is designed to prevent the process from automatically restarting after a certain number of crashes have occurred. It is however important to note that there are some circumstances where brute force attacks can be carried out on Windows, such as when targeting an application where the vulnerable code path is contained within a catch-all exception block.

Certain types of vulnerabilities can also make it possible to bypass ASLR using what is referred to as a*partial overwrite*. This technique relies on an attacker being able to overwrite the low order bits of an address (which are not subject to randomization by ASLR) without perturbing the higher order bits (which are randomized by ASLR).

**Summary**: ASLR breaks an attacker's assumptions about where code and data are located in the address space of a process. ASLR can be bypassed if the attacker can predict, discover, or control the location of certain memory regions (particularly DLL mappings). The absence of DEP can allow an attacker to use heap spraying to place code at a predictable location in the address space.

**9.** (**10 punct**) Fie secvent, a de instruct, iuni de mai jos:

printf(\%d\n", \*a);

printf(\%d\n", \*(a+1));

unde a este un pointer la un ˆıntreg (int \*). Dat, i exemplu de situat, ie ˆın care prima instruct, iune

**NU** cauzeaz˘a page fault, dar a doua cauzeaz˘a page fault.

**10.** (**10 puncte**) Care este un avantaj, respectiv un dezavantaj al folosirii suportului de *huge pages* ? Adic˘a pagini de 2MB (*2 megabytes* ) ˆın locul paginilor de 4KB (*4 kilobytes* ).

Avantaje:

* Increased performance through increased TLB hits.
* Pages are locked in memory and are never swapped out which guarantees that shared memory like SGA remains in RAM.
* Contiguous pages are preallocated and cannot be used for anything else but for System V shared memory (e.g. SGA)
* Less bookkeeping work for the kernel for that part of virtual memory due to larger page sizes

Dezavantaje:

The amount of wasted memory will increase as a result of internal fragmentation;

extra data dragged around with sparsely-accessed memory can also be costly.

Larger pages take longer to transfer from secondary storage, increasing page fault latency (while decreasing page fault counts).

The time required to simply clear very large pages can create significant kernel latencies.

**11.** (**15 puncte**) Dorim s˘a implement˘am un alocator ˆımbun˘at˘at, it de memorie. Alocatorul va expune funct, iile malloc(), calloc(), realloc() s, i free(), apeluri standard ˆın lucrul cu memoria. ˆIn back end va folosi apelurile de sistem de tip mmap() sau brk() expuse de sistemul de operare pentru rezervarea de memorie virtual˘a. Cerint, ele alocatorului sunt vitez˘a foarte bun˘a s, i thread safety.

Ce structuri interne vet, i folosi ˆın cadrul alocatorului pentru gestiunea aloc˘arilor? Ce probleme posibile (de vitez˘a/eficient, a˘) pot ap˘area la nivelul alocatorului? Cum vet, i asigura vitez˘a bun˘a de alocare/dezalocare?

Ce fel de aplicat, ii/scenarii de test vet, i folosi pentru a testa alocatorul?

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

4 septembrie 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Care este un avantaj al folosirii tabelei de pagini ierarhice fat, a˘ de tabela de pagini simpl˘a?

A mai fost inca o data.

**2.** (**7 puncte**) De ce un sistem este cu atˆat mai ˆınc˘arcat cu cˆat num˘arul de procese din cozile

READY cres, te (adic˘a mai multe procese ˆın cozile READY ˆınseamn˘a ˆınc˘arcare mai mare)?

In a real time system, admitting too many processes to the "ready" state may lead to oversaturation and [overcontention](http://en.wikipedia.org/wiki/Bus_contention) for the systems resources, leading to an inability to meet process deadlines.

overcontention: **Bus contention**, in [computer design](http://en.wikipedia.org/wiki/Computer_design), is an undesirable state of the [bus](http://en.wikipedia.org/wiki/Computer_bus) in which more than one device on the bus attempts to place values on the bus at the same time.

**3.** (**7 puncte**) ˆIn ce situat, ie o operat, ie de tip lock() pe un mutex blocheaz˘a thread-ul curent s, i ˆın ce situat, ie nu ˆıl blocheaz˘a?

And for default mutexes, attempting to lock a mutex that has been locked by the calling thread leads to undefined behaviour:

If the mutex type is PTHREAD\_MUTEX\_DEFAULT, attempting to recursively lock the mutex results in undefined behaviour.

lock()

Locks the mutex. If another thread has locked the mutex then this call will block until that thread has unlocked it.

Calling this function multiple times on the same mutex from the same thread is allowed if this mutex is a[recursive mutex](http://doc.qt.io/qt-4.8/qmutex.html#RecursionMode-enum). If this mutex is a [non-recursive mutex](http://doc.qt.io/qt-4.8/qmutex.html#RecursionMode-enum), this function will *dead-lock* when the mutex is locked recursively.

**4.** (**7 puncte**) De ce au dispozitivele de tip bloc nevoie de operat, ii mai rapide (cu throughput mai mare) decˆat dispozitivele de tip caracter?

**5.** (**7 puncte**) Cu ce difer˘a un apel de bibliotec˘a de un apel de sistem?

System calls are operating system functions, like on UNIX, the [malloc()](http://linux.die.net/man/3/malloc) function is built on top of the [sbrk()](http://linux.die.net/man/2/sbrk) system call (for resizing process memory space).

Libraries are just application code that's not part of the operating system and will often be available on more than one OS. They're basically the same as function calls within your own program.

The line can be a little blurry but just view system calls as kernel-level functionality.

**6.** (**10 puncte**) ˆIn ce situat, ie se poate rula cod pe stiv˘a, chiar ˆın cazul folosirii unui mecanism de *stack smashing protection* (*canary value* )?

Stack-smashing protection is unable to protect against certain forms of attack. For example, it cannot protect against buffer overflows in the heap. There is no sane way to alter the layout of data within a [structure](http://en.wikipedia.org/wiki/Data_structure); structures are expected to be the same between modules, especially with shared libraries. Any data in a structure after a buffer is impossible to protect with canaries; thus, programmers must be very careful about how they organize their variables and use their structures.

**7.** (**10 puncte**) La montarea unui sistem de fis, iere se poate folosi opt, iunea noatime. Opt, iunea

ˆınseamn˘a c˘a **nu** va fi actualizat cˆampul atime (timestamp de acces) al unui inode ˆın momentul acces˘arii (citirii sau scrierii inode-ului). De ce este avantajoas˘a aceast˘a opt, iune ˆın cadrul unui sistem de fis, iere ˆınc˘arcat (cu accese dese la fis, iere)?

This basically means that the number of writes to a disk for relatime mount is close to double relative to a noatime mount other thing being equal. It is a serious concern for partitions on flash memory devices.

**8.** (**10 puncte**) Fie secvent, a de instruct, iuni de mai jos:

\*a = 42; /\* first dereferencing \*/

sleep(5); /\* sleep for 5 seconds \*/

\*a = 42; /\* second dereferencing \*/

unde a este un pointer la un ˆıntreg (int \*). Dat, i exemplu de situat, ie ˆın care prima dereferent, iere

**nu** cauzeaz˘a page fault, dar a doua dereferent, iere cauzeaz˘a page fault.

**9.** (**10 punct**) Un proces ˆın Linux are, ˆın mod obis, nuit, tabela de descriptori de fis, iere limitat˘a la 1024 de intr˘ari. De ce ˆın cazul unui server TCP ˆınc˘arcat, care primes, te multe conexiuni, este nevoie de cres, terea acestei limite?

**10.** (**10 puncte**) De ce este mai probabil˘a aparit, ia unui *stack overflow* ˆın cazul unui proces multi-threaded fat, a˘ de un proces single-threaded? (*stack overflow* = dep˘as, irea limitei stivei ˆın cadrul spat, iului de adrese al unui proces)

Folosirea unui număr mare de thread-uri în cadrul unui proces poate conduce mai rapid la stack overflow. Fiecare thread are stiva proprie, cu dimensiunea fixată la crearea. Dacă se creează prea multe thread-uri, se va ocupa foarte mult stiva. Stivele thread-urilor vor fi apropiate unele de altele astfel că, în cazul unui flux de apeluri mare (apeluri recursive, de exemplu), există riscul ca stiva unui thread să suprascrie stiva altui thread.

**11.** (**15 puncte**) Ne propunem implementarea unui framework de messaging (message queue framework). Cu ajutorul acestui framework, aplicat, ii diferite pot comunica unele cu celelalte. Exist˘a patru concepte importante: Publishers (cei care produc mesaje), Consumers (cei care consum˘a mesaje din cozi), Exchanges (cei care primesc mesajele ˆın cozi de la Publishers s, i apoi le transmit c˘atre Consumers), Queues (cozi de mesaje, create de Consumers s, i care stocheaz˘a mesajele). Framework-ul trebuie s˘a asigure scalabilitate s, i performant, a˘. ˆIn general, Consumers, Exchanges s, i Publishers se g˘asesc pe sisteme fizic diferite; sunt conectat, i prin Internet/ret, ea.

Definit, i, la nivel de pseudocod, metodele framework-ului folosite de Publishers s, i Consumers. Ce probleme de scalabilitate pot ap˘area la nivelul framework-ului? (adic˘a de la un nivel ˆın sus

se vor resimt, i probleme de performant, a˘)

Ce solut, ii de rezolvare a problemelor de scalabilitate exist˘a? (atˆat la nivelul framework-ului, cˆat s, i la nivelul infrastrcturii folosite ˆın instalare)

Cum asigurat, i o performant, a˘ ridicat˘a a framework-ului s, i pentru o utilizare simpl˘a (f˘ar˘a pro- bleme de scalabilitate)?

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**

**Sisteme de Operare**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

13 septembrie 2014

Timp de lucru: 60 de minute

**Not˘a**: Toate r˘aspunsurile trebuie justificate

**1.** (**7 puncte**) Cˆate tabele de pagini se g˘asesc la un moment dat ˆıntr-un sistem de operare?

Intr-un sistem se gasesc la un moment dat atatea tabele de pagini catre procese exista.

**2.** (**7 puncte**) De ce overhead-ul unui apel de sistem (oricˆat de simplu ar fi apelul de sistem) este, ˆın majoritatea cazurilor, semnificativ mai mare decˆat overhead-ul unui apel de bibliotec˘a? Presupunem c˘a vorbim de un apel de bibliotec˘a care nu face ˆın spate apel de sistem

A system call is a call to the kernel for something and acts as an entry point into the operating system. A system call executes in kernel address space and counts as part of the system time. System calls have a high overhead because of the switch to kernel and back, they are specific to each operating system and generally are not portable.  
  
A library call is a call to a routine in a library, such as printf, and is linked with the program. It executes in the user address space that is passed out by the operating system for user programs and has a much lower overhead than a system call. Library calls can be bundled up with a program so that they are portable.

**3.** (**7 puncte**) ˆIn ce situat, ie o operat, ie de tip down() pe un semafor blocheaz˘a thread-ul curent s, i ˆın ce situat, ie nu ˆıl blocheaz˘a?

If a semaphore has the value 0, a down operation on it will block until someone releases a resource and increments the semaphore.

A non-blocking semaphore does not block on a down operation if the resource is unavailable, but rather yields an error. This can be useful if the program needs that resource immediately or without suspending execution, and if the resource isn't available, the program logic can rather do something else.

**4.** (**7 puncte**) De ce fiecare thread al unui proces dispune de o stiv˘a proprie?

*Thread-ul reprezinta unitatea minimala ce descrie un fir de executie. Fara o stiva nu s-ar putea mentine si memora contextul unui thread atunci cand se face context switching.*

**5.** (**7 puncte**) Un proces CPU intensive este pornit s, i ruleaz˘a timp de 30 de minute. La

ˆıncheierea sa, se observ˘a c˘a acesta a consumat doar 20 de secunde timp de procesor. Cum se explic˘a acest lucru?

**6.** (**10 puncte**) ˆIn general, nu se pot crea hard link-uri la directoare. Cu toate acestea num˘arul de link-uri aferente unui director difer˘a ˆıntre directoare diferite; putem observa acest lucru prin rularea comenzii stat pe diverse directoare: /, /home, /usr/lib. De ce difer˘a num˘arul de link-uri ˆıntre directoare?

*Numarul de link-uri al unui director este reprezentat de numarul total de subdirectoare pentru ca fiecare dintre ele au un link catre .. + cele doua intrari default: . si ..*

*Astfel, putem sa zicem ca numarul de link-uri difera intre directoare deoarece difera numarul de subdirectoare continute.*

**7.** (**10 puncte**) De ce codul dintr-un shellcode se ˆıncheie, ˆın general, cu o instruct, iune pentru realizarea unui apel de sistem (de forma int 0x80)?

**8.** (**10 puncte**) Un utilizator ruleaz˘a, ˆın terminal, **de mai multe ori** cele dou˘a comenzi de mai jos (rul˘am de mai multe ori ca s˘a fie informat, iile citit cache-uite s, i s˘a nu afecteze rezultatul):

find /usr > find.out find /usr

Utilizatorul observ˘a c˘a prima comand˘a (cu redirectare ˆın fis, ier) dureaz˘a semnificativ mai put, in decˆat a doua (f˘ar˘a redirectare, care afis, eaz˘a pe terminal). De ce?

**9.** (**10 punct**) Care este avantajul unui server web care foloses, te mai multe procese pentru servirea cererilor fat, a˘ de unul care foloses, te mai multe thread-uri?

1. Threads will use up much less resident memory than Processes. Yes, with dynamically linked libraries a lot of memory is shared between the Apache Control Process and it's child Processes, however each new Process will need to instantiate all of the modules you have enabled.
2. This is easily testable by comparing the memory usage of each Process where you have, for example, either 5 Processes and 1 Thread each or 5 Processes and 25 Threads each. In my case here, each child Process takes about 7 MBs regardless of the amount of Threads.
3. +For Threads
4. It takes longer to initiate in terms of time and cpu cycles to load a new Process than it does a Thread. This can be tested by verifying avg amount of pages served via 'ab'.
5. +For Threads
6. A Processes Threads all depend on the Process .. The biggest concern here, is that if something happens to the Process it will affect all the Threads that are associated with it. If you're running with a single Process with a bunch of Threads, then when the Process dies so will the Threads. More Processes would therefore cause a better separation, and thus greater "fault" tolerance if you will.
7. +For Processes
8. Related to (3), for modules such as PHP, their memory is loaded by the Process and shared across all of the Threads. This means that if you have php with memory\_limit set to 100Mbs with 25 Threads below, then at max load technically each Thread would be able to allocate a maximum of 4MBs each ( course it won't happen this way, some will hog, some will starve ).

So in the end, it really depends on your use case .. That being said, you'll want to maximize the amount of Threads used so as to diminish memory usage and increase responsiveness. However, you'll have to balance that with a proper amount of Processes for better fault tolerance.

Course I'm no expert here as I've only recently have had to become concerned with this, so I look forward to see what other answers might pop up here !

**10.** (**10 puncte**) Un executabil are zona de cod de 1MB. Cu toate acestea, un proces creat din acest executabil ocup˘a, pe parcursul rul˘arii, maxim 100KB de memorie RAM. Cum explicat, i?

**11.** (**15 puncte**) Dorim s˘a implement˘am o ret, ea peer-to-peer. Ret, eaua va pune pret, mai mult decˆat orice pe disponibilitatea cont, inutului (*availability* ). Fiecare peer va rezerva un spat, iu dat pe hard disk-ul propriu pentru fis, iere care nu sunt ale sale dar care ne propunem s˘a fie disponibile.

Perioada de timp cˆat un peer este activ s, i l˘at, imea sa de band˘a sunt factori ˆın stabilirea nivelului de implicare (*involvement* ) al acestui peer. Un peer implicat va putea desc˘arca mai rapid date de la alt, i peeri; fiecare peer ˆıs, i controleaz˘a banda de upload s, i acord˘a mai mult peerilor implicat, i.

Ret, eaua este folosit˘a pentru transferuri de fis, iere mici (muzic˘a, documentat, ie, mici fis, iere video). Transferul se realizeaz˘a doar ˆıntre un peer s, i alt peer.

Care vor fi primitivele protocolului de comunicare ˆıntre peeri?

Cum at, i ret, ine, ˆın cadrul ret, elei peer-to-peer, informat, ile despre implicarea unui peer (involve- ment), pentru a fi accesate de peeri? Motivat, i alegerea.

Ce facilit˘at, i vet, i folosi pentru o performant, a˘

cˆat mai bun˘a a aplicat, iei specific˘a unui peer

(pentru transferul fis, ierelor pe ret, ea, stocarea fis, ierelor)?

Ce facilit˘at, i ofer˘a ret, eaua pentru a asigura disponibilitatea cont, inutului? Adic˘a fiecare fis, ier s˘a fie stocat ˆın cˆat mai multe locuri.

ˆIn conformitate cu ghidul de etic˘a al Departamentului de Calculatoare, de- clar c˘a nu am copiat s, i nu voi copia la aceast˘a lucrare. De asemenea, nu am ajutat s, i nu voi ajuta pe nimeni s˘a copieze la aceast˘a lucrare.

**Nume s, i grup˘a: Semn˘atur˘a:..............................**