

Sisteme de operare

6 septembrie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Un sistem dispune de următoarele caracteristici

- magistrala de date pe 64 de biți
- overhead-ul impus de un page fault este de 1ms
- nu dispune de memorie cache
- durata unei operații cu memoria este de 100ns

Pe sistem rulează un sistem de operare în cadrul căruia biblioteca standard C folosește un buffer intern de 64K la nivelul fiecărui handle de fișier.

Care din operațiile marcate aldin (bold) de mai jos durează mai mult?

```
char buf[32*1024];
f = fopen("a.dat", "wb");
/* fill buffer */
fwrite(buf, 32*1024);
fflush(f);
fwrite(f, buf, 32*1024);
```

Operația write înseamnă copierea datelor din buf în bufferul intern al bibliotecii standard C. Întrucât bufferul intern oferă spațiu pentru tot buffer-ul nu va exista nici un apel de sistem și nici pagefault-uri.

Operația memcpy va presupune copierea datelor într-o zonă alocată cu mmap. Durata de copiere este identică celei de mai sus, dar au loc page faultsuri la fiecare pagină. Aceasta se întâmplă pentru că mmap alocă memorie virtuală pură (demand paging). Prinul acces la o pagină va conduce la page fault.

În concluzie, operația memcpy durează mai mult decât write.

2. În cadrul problemei celor 5 filozofi se folosește următoarea implementare (în pseudo-C) a funcției eat():

```
mutex_t global_mutex;
void eat(int fork1, int fork2)
{
    lock(global_mutex);
    take_fork(fork1);
    take_fork(fork2);
    do_eat();
    put_fork(fork1);
    put_fork(fork2);
    unlock(global_mutex);
}
```

Care este neajunsul acestiei implementări?

Folosirea mutexului global înseamnă că un singur filozof din cei 5 poate mânca la un moment dat. Fiind 5 furculițe, soluția eficientă permite ca doi filozofi să mănânce simultan.

3. Pe un sistem rulează 50 de procese. La un moment dat este pornită o mașină virtuală VMware Server pe care rulează 50 de procese. Câte procese vor rula pe sistemul gazdă? Dar în cazul pornirii unei mașini virtuale OpenVZ pe care rulează 50 de procese?

O mașină virtuală VMware Server este reprezentată pe sistemul gazdă de un singur proces. Vor exista, în total, 51 de procese.

O mașină virtuală (container) OpenVZ are, pe sistemul fizic, un corespondent pentru fiecare proces. Vor exista, în total, 100 de procese.

4. Pe un sistem de fișiere MINIX se execută operațiile

```
lseek(fd, SEEK_SET, 32*1024);
write(fd, buffer, 1024);
```

Descrieți operațiile asociate asupra strukturilor sistemului de fișiere (inode, bitfield, data block, dentry etc.)

Câte pagini de memorie fizică, respectiv virtuală vor fi ocupate în urma operațiilor de mai sus?

Fiecare proces accesează cele două pagini alocate. În urma accesului se realizează un page fault și se mapează paginile virtuale peste paginile fizice.

Se vor aloca 4 procese * 2 pagini virtuale = 8 pagini virtuale

Pentru fiecare pointer (a1 sau a2) se mapează aceeași pagină fizică. Maparea este partajată (MAP_SHARED) și toate procesele vor vedea același conținut. În cazul particular al mapării a2 (PROT_WRITE) scrierile din cadrul unui proces vor fi vizibile în celelalte procese.

Se vor aloca 1 pagină fizică (prin a2) + 1 pagină fizică (prin a1) = 2 pagini fizice

lseek nu modifică structurile interne ale sistemului de fișiere. Este pozitionat cursorul de fișier (corespondent unei structuri din memorie) la offsetul specificat.

Se calculează blocul aferent adresei 32*1024 prin parcursarea pointerelor de bloc din inode-ul MINIX. Întrucât se depășește spațul referibil prin referință directă se va căuta blocul aferent pentru referință indirectă.

Se parcurge bitfield-ul și să găsește primul bloc liber. Adresa acestui bloc este scrisă pe poziția aferentă din blocul de referință indirectă. Se citește blocul de pe disc în memorie și se scrie informația furnizată din user-space. Se actualizează câmpul size din inode. Ulterior se va face flush la bloc din memorie pe disc.

5. Un sistem folosește un planificator round-robin non-preemptiv. În cadrul sistemului rulează 5 procese care execută următoarea secvență:

[10ms rulare | 10ms așteptare | 10ms rulare]

Cât durează planificarea celor 5 procese?

Planificare round-robin înseamnă că fiecare proces este planificat pe rând. Planificarea se face astfel:

0-10ms: procesul 1

10ms-20ms: procesul 2 (processul 1 așteaptă)

20ms-30ms: procesul 3 (processul 2 așteaptă, procesul 1 este gata de execuție)

30ms-40ms: procesul 4 (processul 3 așteaptă, procesul 1 și procesul 2 sunt gata de execuție)

40ms-50ms: procesul 5 (processul 4 așteaptă, procesele 1, 2 și 3 sunt gata de execuție)

...

Durata de planificare este de 100ms

6. Cum se modifică spațiul de adresă al unui proces la schimbarea de context între două thread-uri?

Nu se modifică. Thread-urile partajează spațiul de adresă al procesului.

7. Are sens folosirea operațiilor asincrone în locul celor sincrone în situația de mai jos (pseudo-cod)?

```
AIO_TYPE aioarray[32];
InitializeAsyncIoArray(&aioArray);
for (i = 0; i < 32; i++)
    StartAsyncIo(&aioArray[i]);
WaitForAllObjects(&aioArray);
```

Cele 32 de operații asincrone sunt pornite în același timp. Durata de așteptare este durata de rulare/planificare a celei mai lente operații. În cazul unei operații sincrone (blocante, sevenciale). Durata de așteptare ar fi fost suma duratelor de rulare/planificare a tuturor operațiilor.

8. Descrieți o situație în care operația:

```
memcpy(a, "12345678901234567890", 20);
durează mai mult, respectiv mai puțin, decât operația
getpid();
```

Operația getpid rezultă într-un apel de sistem. Overhead-ul unui page fault este de 5ms, iar unui apel de sistem de 7ms.

Dacă zona indicată de a (20 de octeți) este pozitionată într-o singură pagină overhead-ul este de 5ms în cazul unui page fault (pagina nu este prezentă în memoria fizică) sau neglijabil în cazul în care pagina este prezentă în memorie. Durează, astfel, mai puțin decât getpid().

Dacă zona indicată de a (20 de octeți) este pozitionată pe două pagini (spre exemplu 8 octeți în prima, 12 în a doua), overhead-ul (în cazul absenței paginilor din memoria fizică) este de 10ms.

9. Pe o arhitectură x86, care registre generale (eax, ebx, ecx, edx, esi, edi, ebp, esp) sunt schimbată în cazul unei schimbări de context între două thread-uri? Dar în cazul unei schimbări de context între două procese?

În ambele cazuri se schimbă tot setul de registre, întrucât definesc un nou context.

10. Se presupune că se implementează la nivel hardware o tehnică ce împiedică accesarea zonelor de memorie nealocate la nivel de octet. Cum poate fi folosită această tehnică pentru prevenirea atacurilor de tip buffer overflow la nivelul stivei?

Nu previne. Atacul de tip buffer overflow înseamnă suprascrierea stivei sau a altrei regiuni deja alocate și a adresei de return (și aceasta alocată pe stivă). Protecția la accesarea unor zone nealocate nu împiedică acest tip de atac.

11. Într-un sistem de operare, 4 (patru) procese execută operațiile

```
fd1 = open("a.txt", O_RDONLY);
fd2 = open("b.txt", O_RDONLY);
a1 = mmap(NULL, 4*1024, PROT_READ, MAP_SHARED, fd1, 0);
a2 = mmap(NULL, 4*1024, PROT_READ | PROT_WRITE, MAP_SHARED, fd2, 0);
printf("sc", *a1);
*a2 = 'a';
```

Sisteme de operare

10 septembrie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Care din următoarele acțiuni consumă cel mai mult timp în cazul unei schimbări de context între două thread-uri ale aceluiași proces:

- schimbarea registrilor
- flush TLB
- schimbarea tabeliei de pagini
- schimbarea tabeliei de descriptori de fișier

În cazul schimbării de context între două thread-uri ale aceluiași proces nu se face flush la TLB, nu se schimbă tabela de pagini, nu se schimbă tabela de descriptori de fișier. În consecință, deși foarte rapidă, acțiunea care consumă cel mai mult timp este schimbarea registrilor.

2. Un sistem dispune de magistrală de date și registre pe 32 de biți (sizeof(unsigned long) = 32). Sistemul folosește paginare simplă (non-ierarhică) și nu are memorie cache, nici TLB. Stînd că un acces la memoria durează 50ns iar o pagină este de 4KB, cat va dura sevența de mai jos? Se presupune că vectorul buffer este alocat în RAM (memoria fizică) și că valoarea contorului i se păstrează într-un registru (nu folosește memoria).

```
unsigned long buffer[32*1024];
for (i = 0; i < 32 * 1024; i++)
    buffer[i] = 1;
```

În absența TLB fiecare acces la memoria fizică necesită accesarea tabeliei de pagini (aflată tot în memoria fizică). Astfel, pentru fiecare dintre cele 32*1024 de accese la buffer vor exista încă 32*1024 accese la tabele de pagini. Rezultă 64*1024 accese la memoria cu o durată totală de 64*1024*50ns.

3. Care dintre variantele chroot, respectiv OpenVZ oferă un grad mai mare de securitate?

chroot oferă "incapsulare" (securizare) doar la nivelul sistemului de fișiere. OpenVZ oferă securizare la nivelul proceselor, memoriei, procesorului, rețelei, utilizatorilor etc. OpenVZ oferă, aşadar un grad mai mare de securitate.

4. În cadrul unui proces cu mai multe thread-uri, un thread execută următoarea sevență de cod (pseudo C):

```
int esp;
int stack_val;
/* se obține valoarea registrului esp (registrator de stivă) al thread-ului planificat anterior */
esp = get_former_esp();
stack_val = *(esp + 4);
```

Care va fi rezultatul execuției sevenței de mai sus pe un sistem în care stiva crește în jos?

Întrucât stiva crește în jos, valoarea esp+4 va puncta către o zonă alocată din stiva fostului thread. Execuția de mai jos va rezulta în obținerea aceliei valori (nu se va obține segmentation fault decât dacă thread-ul anterior și-a încheiat execuția).

5. Pe un sistem de fișiere MINIX se execută operațiunile

```
fd = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
```

Precizați ce se întâmplă la nivelul sistemului de fișiere (inode, inode bitmap, zone bitmap, data block, dentry etc.) în cazul în care fișierul există sau nu există pe disc.

Dacă fișierul există se găsește dentry-ul acestuia și se obține inode-ul aferent și se citește inode-ul în memorie.

Dacă fișierul nu există se creează un inode nou. Pentru această parcurge bitmapul de inode-uri și se alocă un inode. Se completează cu 1 pozitie liberă găsită. Se creează un dentry cu numele "a.txt".

Dacă fișierul există trunchiat. Se parcurge pointer-ii de blocuri ai inode-ului și se marchează cu NULL (sau ceva echivalent). Se parcurge bitmapul de blocuri și marchează cu 0 poziții aferente acelor blocuri. Dacă fișierul avea mai mult de 7 blocuri se citește și completează cu NULL blocul pentru dereferențieri simple.

6. Precizați o soluție de sincronizare pentru problema formării moleculei de oxid de fier (Fe₂O₃) (ca alternativă la problema formării apei).

```
void fe_fun()
{
    m.enter();
    fe_count++;
    if (o_count >= 3) {
        if (fe_count < 2)
            m.fe_cond.wait();
        else {
            m.o_cond.signal();
            m.o_cond.signal();
            m.o_cond.signal();
            m.fe_cond.signal();
            fe_count = 2;
        }
    }
    else
        m.fe_cond.wait();
    m.leave();
}

void o_fun()
{
    m.enter();
    o_count++;
    if (o_count >= 3 && fe_count >= 2) {
        m.o_cond.signal();
        m.o_cond.signal();
        m.fe_cond.signal();
        m.fe_cond.signal();
        if (o_count > 3) {
            m.o_cond.signal();
            m.o_cond.wait();
        }
    }
    else
        m.o_cond.wait();
    m.leave();
}
```

7. Biblioteca standard C oferă programatorului funcția `calloc` (alloca cu zeroing). De ce este nevoie și de oferirea funcției `malloc`?

Funcția malloc este mai rapidă – nu face zeroing și permite demand paging.

8. Pe un sistem care dispune de 3 pagini fizice (frames) și folosește un algoritm de înlocuire a paginii de tip NRU se execută următoarea secvență:

1r, 2w, 4r, 1w, 3r, 2w, 1w, 4w, 3r, 3w, 1r, 2r, 5r, 2w, 6r, 3w, 1w, 2r

Câte page fault-uri au loc? Iată cum arată operația de citire în cadrul paginii virtuale 1: 2w înseamnă operație de scrisere în cadrul paginii virtuale 2.

Continutul celor 3 pagini fizice, înpreună cu evenimentul aferent este prezentat, evolutiv, în tabelul de mai jos; la fiecare două pagefault-uri se resetează bitul referențial:

frame	1r (R)	1r (R)	1r (W)	3r (R)	3r (R)	3r (W)	3w (R)	3w (W)	2r (R)	2r (R)	2w (W)	6r (R)	6r (R)	6r (W)	2r (R)
frame 1															
frame 2	2w (W)	4w (W)	4w (W)	4w (W)	4w (W)	5r (R)	5r (R)	5r (W)	3w (W)	3w (W)	3w (W)				
frame 3			4r (R)	4r (R)	4r (W)	1w (R)	1w (R)	1w (W)	1w (W)	1w (W)	1w (W)				
	PF														

9. Fie secvența de program de mai jos:

```
int main(void)
{
    char *a;
    int i;

    for (i = 0; i < 10; i++)
        a = malloc(1);

    return 0;
}
```

La rulare se observă (prin folosirea unui profiler) că primul apel `malloc` durează semnificativ mai mult decât celelalte 9. Care este motivul? Toate apelurile reușesc (întorc o adresa validă) și nu există nici o modificare adusă apelului `malloc`.

Primal apel malloc generează un page fault, urmată fiind alocarea unei pagini fizice întregi (chiar dacă se solicită alocarea unei singure octeți). Următoarele apeluri vor aloca octeți din cadrul aceleiași pagini – nu mai este generat un page fault și nu se aloca alte pagini.

10. Are sens folosirea operațiilor de tip Overlapped I/O pe un sistem care dispune de un singur hard-disk?

Da. Operațiile overlapped I/O permit să planificare mai eficientă operațiile de I/O la nivelul nucleului și permit aplicației să ruleze

11. Descrieți o situație în care două procese partajează o pagină virtuală (din spațiu virtual de adrese).

Fiecare proces are propriul spațiu de adrese. Nu există noțiunea de partajare a unei pagini virtuale.

6. Are sens folosirea unui sistem de protejare a stivei (stack smashing protection, canary value) pe un sistem care dispune de și folosește bitul NX?

Da, are sens. În general, sistemele de tip stack overflow suprascriu adresa de return a unei funcții cu o adresă de pe stivă. Bitul NX previne execuția de cod pe stivă. Dar adresa de return poate fi suprascrisă cu adresa unei funcții din zona de text (return_to libc attack) sau o adresă din altă zonă care poate fi executată (biblioteci, heap).

7. Pe un sistem quad-core și 4GB RAM rulează un proces care planifică 3 thread-uri executând următoarele funcții:

```
thread1_func(initial_data)
{
    for (i = 0; i < 100; i++) {
        work_on_data1();
        wake_thread2();
        wait_for_data_from_thread3();
    }
}

thread2_func()
{
    for (i = 0; i < 100; i++) {
        wait_for_data_from_thread1();
        work_on_data2();
        wake_thread3();
    }
}

thread3_func()
{
    for (i = 0; i < 100; i++) {
        wait_for_data_from_thread2();
        work_on_data3();
        wake_thread1();
    }
}
```

Care este dezavantajul acestui abordare? Propuneți o alternativă.

Codul de mai sus este un cod serial. Folosirea celor trei thread-uri este neficientă pentru că se execută mai ușor în cadrul unui singur thread (apar overhead-uri de creare, sincronizare și schimbările de context între thread-uri). Soluția este folosirea unui singur thread sau reglarea algoritmului folosit pentru a putea fi cedată de către un alt thread.

8. În spațiu de adresă al unui proces, zona de cod (*text*) este mapată read-only. Acest lucru este avantajos din punct de vedere al securității, intrucât împiedică suprascrierea codului executat. Ce alt avantaj important oferă?

Find read-only zona poate fi partajată între mai multe procese limitând spațiul ocupat în RAM.

9. Folosind o soluție de virtualizare, se dorește simularea unei rețele formată din: două sisteme Windows, un gateway/firewall OpenBSD și un server Linux. Opțiunile sunt VMware Workstation, OpenVZ și Xen. Care variantă de virtualizare permite rularea unui număr cât mai mare de instanțe de rețele pe un sistem dat?

OpenVZ nu poate fi folosit pentru că este OS-level virtualization: toate mașinile virtuale folosesc același nucleu deci pot fi folosite mașini virtuale care rulează același sistem de operare ca sistemul gazdă. Xen este o soluție rapidă dar rularea unui sistem nemodificat (gen Windows) este posibilă doar în situația în care hardware-ul peste care rulează oferă suport (Intel VT sau AMD-V). VMware Workstation este o soluție mai lentă, în general, decât Xen dar permite rularea oricărui tip de sistem de operare guest.

10. Un sistem de operare dat poate să folosească un split user/kernel 2GB/2GB al spațiului de adresă al unui proces sau un split 3GB/1GB. Sistemul fizic dispune de 1GB RAM. Un proces rulează secvența:

```
for (i = 0; i < N; i++)
    malloc(1024*1024);
```

Pentru ce valori (aproximative) ale lui *N* `malloc` va întoarce NULL în cele două cazuri de split?

În exemplul de cod de mai sus, malloc aloca memorie pur virtuală (fără suport de memorie fizică). Alocarea de memorie fizică se va realiza la cerere (sună paging). malloc va întoarce NULL în momentul în care procesul rămâne fără memorie virtuală în user-space. N va avea, aşadar, valori aproximative de 2048 și 3072. Dimensiunea memoriei RAM a sistemului este nerăbdătoră în această situație.

11. Un proces dispune de o tabelă de descriptori de fișiere cu 1024 de intrări. În codul său, procesul deschide un număr mare de fișiere folosind `open`. Totuși, al 1010-lea apel `open` se întoarce cu eroare, iar `errno` are valoarea `EMFILE` (maximum number of files open). Care este o posibilă explicație?

Procesul realizează 1009 apeluri `open` cu succes, rezultând în 1009 file descriptori deschisi. `stderr`, `stdout`, `stdin` sunt 3 descriptori inițiali, rezultând 1012 descriptori. Restul au fost creați prin alte metode. File descriptori pot fi creați și astfel: `creat` (creare fișiere), `dup`, `socket`, `pipe`. O altă situație este aceea în care procesul moștenește un număr de file descriptori de la procesul părinte.

Sisteme de operare

25 iunie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie **justifycate**

1. "Sistemele de operare moderne nu au probleme de fragmentare externă a memoriei fizice alocate din user-space." Indicați și motivea valoarea de adevără a propoziției anterioare.

Sistemele de operare moderne folosesc suportul de paginare pus la dispozitivele de sistemul de calcul. Folosirea paginării înseamnă că se pot aloca ușor pagini de memorie fizică acolo unde sunt libere. Mecanismul de memorie virtuală asigură faptul că a aloca rămâne virtual contigu. În felul acesta dispările de fragmentare externe – adică de găsire a unui spațiu continuu pentru alocare (rămân însă problemele de fragmentare interne).

Excepție fac acoloce din kernel-space care pot solicita alocare de memorie fizică contiguă sau alocările impuse de hardware (de exemplu DMA).

2. Un sistem de operare dispune de un planificator de procese care folosește o cantitate de 100ms. Durata unei schimbări de context este 1ms. Este posibil ca planificatorul să petreacă jumătate din timp în schimbări de context? Motivați.

Da, este posibil în situație în care procesele planificate execută acțiuni scurte și apoi se blochează determinând schimbări de context. Acest lucru se poate întâmpla în cazul sincronizării între procese (un proces P1 execută o acțiune, apoi trezește procesul P2 și apoi se blochează, procesul P2 execută o acțiune, apoi trezește procesul P1, etc.), sau în cazul comunicării cu dispozitive de I/O rapide (procesul P1 planifică o operație I/O și se blochează, operația se încheie rapid și trezește procesul P2).

O altă situație este schimbarea rapidă a priorității proceselor care determină schimbarea de context pentru rularea procesului cu prioritatea cea mai bună.

3. Dați exemplu de funcție care este reentrantă, dar nu este thread-safe. Dați exemplu de funcție care este thread-safe, dar nu este reentrantă.

Toate funcțiile reentrantă sunt thread-safe. Exemplu de funcție care este thread-safe dar nu reentrantă este `malloc`. Un exemplu generic este o funcție care folosește un mutex pentru sincronizarea accesului la variabilele partajate între thread-uri: funcția este thread-safe, dar nu este reentrantă (nu pot fi executate simultan două instanțe ale acestei funcții). Proprietatea de reentrantă sau thread-safety se referă la implementarea și interfața funcției, nu la contextul în care este folosită (o funcție reentrantă poate fi folosită într-un context unsafe din punct de vedere al sincronizării, dar nu înseamnă că este non-thread-safe).

4. Într-un sistem de fisiere FAT un fișier ocupă 5 blocuri: 10, 59, 169, 598, 1078. Știind că:

- un bloc ocupă 1024 de octeți
- o intrare în tabelă FAT ocupă 32 de biți
- tabela FAT NU se găsește în memorie
- copierea unui bloc în memorie durează 1ms

cât timp va dura copierea completă a fișierului în memorie?

Un bloc ocupă 1024 de octeți, o intrare în tabelă FAT ocupă 32 de biți și tabela FAT NU se găsește în memorie, deci sumă 256 intrări FAT într-un bloc. În tabela FAT intrările 10, 59, 169 se găsesc în primul bloc, intrările 598 și 1078 în al treilea bloc și 1078 în al cincilea bloc. Vor trebui, astfel, cîte 3 blocuri asociate tabelei FAT. Fișierul ocupă 5 blocuri, deci vor fi cîtite, în total, 8 blocuri. Timpul total de copiere este 8ms.

5. Două procese P1, respectiv P2 ale aceluiași utilizator sunt planificate după cum urmează:

fd = open("/tmp/a.txt", O_CREAT O_RDWR, 0644); write(fd, "p1", 2); --- schedule ---	
	--- schedule --- fd = open("/tmp/a.txt", O_CREAT O_RDWR, 0644); write(fd, "p2", 2);

Ce va conține, în final, fișierul /tmp/a.txt? Ce va conține fișierul în cazul în care se folosesc thread-uri în loc de procese?

Două apeluri open întorc descriptori către structuri distincte de fișier deschis. Acest lucru înseamnă că fiecare descriptor va folosi un cursor de fișier propriu. Al doilea apel open va poziționa cursorul de fișier la începutul fișierului și va suprascrie mesajul primului proces. În final în fișier se va scrie P2. În cazul folosirii thread-urilor situația este neschimbată pentru că se vor folosi, din nou, curseau de fișier diferite.

Sisteme de operare

26 iunie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie **justifycate**

1. Știind că operațiile de lucru cu pipe-uri sunt atomice, implementați în pseudocod un mutex cu ajutorul pipe-urilor.

```
lock: read(pipefd[0], &a, 1);
unlock: write(pipefd[1], &a, 1);
```

a este un char; pipefd este un pipe

2. Durata unei schimbări de context este de 1ms iar overhead-ul unui apel de sistem de 100μs. Totuși, la un moment dat, apelul durează 400μs; durează doar 1μs. Apelul se realizează cu succes. Care este explicația?

Apelul down este implementat în user-space (de exemplu o implementare de tip `utex`). Dacă valoarea semaforului este strict pozitivă, atunci apelul down va decrementa valoarea semaforului și va continua execuția (fără apel de sistem și fără schimbare de context). În cazul în care valoarea este egală cu 0 va avea loc un context switch. În cazul unel implementării de thread-uri kernel-level, acest lucru va presupune și un apel de sistem (planificator este implementat în kernel-space).

3. De ce este mai avantajos ca, pe un sistem uniprocessor, după un apel `fork` să fie planificat primul procesul fiu?

Pentru a evita posibilele copieri inutile datorate `copy-on-write`. De multe ori, după fork procesul copil execută exec, rezultând în schimbarea completă a spațiului de adresă. Dacă procesul părinte va planifica primul, atunci apelurile de scris ale acestuia vor rezulta în duplicarea paginilor (overhead temporal și consum memorie) datorită `copy-on-write`. Dacă procesul face exec, atunci acele duplicate au însemnat un consum inutil de resurse.

4. Există o diferență între implementarea simbolului `errno` în contextul unui proces single-threaded față de un proces multi-threaded? Argumentați.

Da, există diferență. Fiecare thread trebuie să aibă acces la o variabilă `errno` proprie, astfel că `errno` va fi de obicei implementat ca variabilă per-thread. Acest lucru se poate realiza cu ajutorul `TLS/TSD`. O variabilă comună `errno` pentru toate thread-urile ar conduce la incoerența informațiilor referitoare la erorile apărute.

5. Un sistem uniprocessor (single-core) dispune de 64KB L1 cache, 512KB L2 cache și un TLB cu 256 intrări. Pe un sistem de operare cu suport în kernel pentru thread-uri, ce durează mai mult: schimbarea de context între două thread-uri sau între două procese?

Schimbarea de context între două procese va dura tot timpul mai mult decât schimbarea de context între două procese. În momentul schimbării de context între două procese se schimbă întreg spațiul de adresă și resursele asociate. Se schimbă astfel tabela de pagini, se face flush la TLB etc. În cazul thread-urilor se presupune că schimbarea de context presupune doar schimbarea registrilor și a informațiilor specifice unui thread.

6. Comanda `pmap` afișează informații despre spațiul de adrese al unui proces. În urma rulării de mai jos a comenzi `pmap` se observă următoarele informații despre biblioteca standard C:

```
# pmap 1
base address      size   rights   name
[...]
00007fc48060000 1320K  r----
```

Preșupunând că în sistem rulează 50 de procese care folosesc biblioteca standard C, care este spațiu total de memorie RAM ocupat de bibliotecă?

*Ultima zona este o zonă read-write și nu poate fi partajată între două procese. Celelalte două zone sunt read-only și vor fi partajate. Biblioteca va ocupa, aşadar, 50*8K + 12K + 1320K.*

7. Descrieți și explătiți în pseudo-asamblare cum acționează suportul de SSP (Stack Smashing Protection) pe un sistem în care stiva crește în sus (de la adrese mici la adrese mari).

Pe un sistem pe care stiva crește în sus nu se poate realiza stack overflow din stack-frame-ul curent ci din stack frame-ul apelantului. Astfel, dacă o funcție apelează strcpy și un argument este un buffer al funcției, acest buffer poate fi folosit pentru a suprascrie (prin overflow) adresa de return a funcției strcpy. Valoarea de tip canary trebuie stocată la o adresă mai mică decât adresa de return a funcției strcpy (practic, la fel ca la o stivă care crește în jos). Întrucât apelantul este cel care construiește stack-frame-ul apelantului, acesta va trebui să marcheze valoarea de tip canary. În schimb apelatul (aici strcpy), înainte de întoarcere va verifica suprascrierea adresei de tip canary.

Stack frame-ul este cel de mai jos:

```
[ strcpy local ]
[ ... ]
[ ret address ]
[ old_ebp ] callee (strcpy) stack frame
[ canary value ] ----
[ strcpy param1 ]
[ strcpy param2 ]
[ ... ]
[ local buffer ] caller stack frame
[ ... ]
[ local buffer ]
```

8. Pe un sistem de fișiere dat un dentry are următoarea structură:

- 1 octet - lungimea numelui
- 251 octet - numele
- 4 octeți - numărul inode-ului

O instanță a unui astfel de sistem de fișiere definește un director rădăcină, 5 subdirective, iar fiecare subdirector conține 5 fișiere. Căte dentry-uri definește sistemul de fișiere?

Fiecare intrare în sistemul de fișiere (director sau fișier) conține cel puțin un dentry. Ignorând intrările speciale, și rezultă $(1 + 5 + 5^5 = 30)$ (31) intrări. Directorul rădăcină poate să nu aibă dentry. Considerând intrările speciale, rezultă un plus de $1 + 5^5$ intrări = 11 intrări (directorul rădăcină nu are referință)..

9. Un sistem pe 64 de biți folosește pagini de 8KB și 43 de biți pentru adresele într-o schemă de adresare ierarhică pe trei niveluri cu împărțirea $(10 + 10 + 10 + 13)$. Un proces care rulează în cadrul acestui sistem are, la un moment dat, următoarea compoziție a spațiului de adrese (se începe de la adresa 0):

```
[ text ] - 16 pagini
[ data ] - 8 pagini
[ spațiu nealocat ] - 8168 pagini
[ stivă ] - 8 pagini
```

Ştiind că o intrare în tabela de pagini ocupă 64 de biți, câte pagini ocupă tabelile de pagini pentru procesul dat?

O intrare în tabela de pagini ocupă 64 de biți = 8 octeți. Există, astfel, 1024 de intrări într-o pagină. Zona text și data ocupă 24 de pagini deci vor exista 24 de intrări valide în prima pagină de tabelă de pe nivelul 3. Următoarele 8168 pagini vor completa intrările din prima tabelă de pe nivelul 3 și vor mai folosi 7 pagini de tabele. Întrucât este spațiu nealocat, cele 7 pagini de tabele nu vor fi nici ele alocate. Un 9-a pagină de tabelă va folosi primele 8 intrări pentru a referi paginile de stivă.

Prima pagină de tabelă de pe nivelul 2 va avea validă doar prima și a 9-a intrare (care vor referi prima și a 9-a pagină de tabelă). Pagina de tabelă de pe nivelul 1 va avea validă doar prima intrare către pagina de tabelă de pe nivelul 2. Vor fi, astfel, folosite, doar 4 pagini.

Schematic, reprezentarea este următoarea:

```
[ level 1 page table ] ---> [#1 level 2 page table] ---> [#1 level 3 page table] ---> text
| ...
| +--> data
...
+---> [#9 level 3 page table] ---> stack
...
```

10. Dati exemplu de situație în care, pentru comunicația cu dispozitivele de I/E, se preferă folosirea polling în loc de interruperi.

Pollingul se preferă în situații în care interrupterile previn funcționarea eficientă a sistemului. Acest lucru se întâmplă în cazul în care interrupterile sunt transmise foarte des și procesorul petrece mult timp în rutinele de tratare a interrupterilor. Soluția este dezactivarea temporară a interrupterilor și folosirea polling. Acest lucru se întâmplă la dispozitivele de rețea foarte rapide, spre exemplu plăcile de rețea.

1. Care din următoarele instrucțiuni ar putea suprascrie adresa de return a unei funcții? (my_func este o funcție) Motivați și precizați contextul în care se poate întâmpla. (Poate fi un singur răspuns, răspunsuri multiple, nici unul, toate răspunsurile)

```
long *a = malloc(30); /* definire și alocare */
```

```
/* instrucțiuni */
a) a = my_func;
b) *(a + 4) = my_func;
c) *(a + 0x40000000) = my_func;
d) memcpy(my_func, a, sizeof(void *));
```

Cuvânt cheie: ar putea. Unele situații sunt improbable dar posibile.

Înainte de toate:

- a este o variabilă (de tip pointer)
- a rezidă pe stivă
- &a este adresa pe stivă a variabilei a
- a (valoarea a) este o adresa de heap (punând către zona de 30 de octeți alocate folosind malloc)
- în general, heap-ul crește în sus și stiva crește în jos
- my_func este o funcție deci rezidă în .text (zona de cod)

a) nu are un efect: se suprascrie valoarea lui a cu adresa funcției my_func
b) posibil: dacă există unde variabile între [retebp] și [a] atunci &a + 4 poate puncta către adresa unde se găsește valoarea de return și *&a + 4) și poate suprascrie
c) posibil: a + 0x40000000 înseamnă adunarea a 0x40000000 la o adresă de heap (valoarea lui a); se poate ajunge (greu probabil, dar posibil) la o adresă de return de pe stivă
d) ciudată, se suprascrie o informație din zona de cod (zona read only); pot apărea erori de acces sau comportamente nedeterminist (în nici un caz nu suprascrierea adresei de return dinto-ru funcție)

2. Un proces este folosit pentru calcularea de transformate Fourier iar un altul este folosit pentru căutarea de informații într-o ierarhie de fișiere. Care dintre cele două procese va avea prioritate mai mare? De ce?

Proces care calculează transformate Fourier – CPU intensive. Proces care căută informații într-o ierarhie de fișiere – I/O intensive. În general, procesele I/O intensive au prioritate mai mare. Motivele sunt:

- creșterea interactivității
- împiedicarea starvation (fairness): dacă nu ar fi astfel prioritizate, procesele CPU intensive s-ar transforma în "processor hogs" și ar folosi resursele sistematici
- procesele I/O intensive ocupă timp puțin pe procesor deci întârzierea provocată altor procese este mică

3. Un sistem dispune de un TLB cu 128 de intrări; care este capacitatea maximă a memoriei fizice și a memoriei virtuale pe acel sistem?

Nu există nici o legătură. TLB-ul menține mapările de pagini fizice și pagini virtuale. Conține un subset al tabeliei de pagini. Memoria fizică și memoria virtuală pot fi oricără de mici/mari. Nu sunt afectate de dimensiunea TLB-ului.

4. Completează zona punctată de mai jos cu (pseudo)cod Linux (POSIX) sau Windows (WIN32) (la alegere) care va conduce la afișarea mesajului "alfa" la ieșirea standard (standard output) și mesajul "beta" la ieșirea de eroare standard (standard error):

```
/* de completat */
[...]
fputs("alfa", stderr);
fputs("beta", stdout);
Nu alterați simbolurile standard fputs (funcție), stderr și stdout (FILE *).
```

Problema este, de fapt, o problemă a păharelor ascunsă. Se dorește ca ieșirea standard să folosească descriptorul 2, iar ieșirea de eroare standard să folosească descriptorul 1.

În pseudocod Linux, lucrurile stau astfel:

```
int aux_fd;
/* aux_fd punctează către ieșirea standard */
dup2(STDOUT_FILENO, aux_fd);
/* descriptorul de ieșire standard este închis și apoi punctează către ieșirea de eroare standard */
dup2(STDERR_FILENO, STDOUT_FILENO);
/* descriptorul de ieșire de eroare standard este închis și apoi punctează către ieșirea standard (indicată de aux_fd) */
dup2(aux_fd, STDERR_FILENO);
fputs ....
```

5. Fie următoarea secvență de (pseudo)cod:

11. Un program execută secvența de cod din coloana din stânga tabelului de mai jos. În coloana din dreapta este prezentat rezultatul rulării programului:

```
/* init array to 2, 0, 0, 0 ... */
static int data[1024*1024] = {2, };
static void print_time(char *msg)
// ...

static void init_array(int *a, size_t len)
{
    size_t i;
    for (i = 0; i < len; i += 1024)
        a[i] = 2097;
}

int main(void)
{
    int *data2 = malloc(1024*1024 * sizeof(int));
    print_time("before init data1");
    init_array(data, 1024*1024);
    print_time("after init data1");

    print_time("before init data2");
    init_array(data2, 1024*1024);
    print_time("after init data2");

    return 0;
}
```

before init data1: 12455829628, 753431us
after init data1: 12455829628, 767496us
before init data2: 12455829628, 767522us
after init data2: 12455829628, 776012us

Cum explică faptul că initializarea vectorului *data1* durează mai mult decât initializarea vectorului *data2*?

Data1 se găsește în *data* și este stocat în executabil. Zona *data* a executabilului va fi mapată în memorie folosind demand paging. Drepă consecință, un page-fault în momentul inițializării vectorului *data1* va forța citirea de pe disc (din executabil). De aceea ceeață, *data2* va fi alocat direct în RAM la cerere (tot prin demand-paging).

int *a;
a = mmap(NULL, 4100, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
n = read_int_from_user();
a[n] = 42;
n = read_int_from_user();
a[n] = 42;

Ce efect au valorile introduse de utilizator asupra programului? (page faults, erori, scrieri în memorie) Discuție. (o pagină ocupă 4 KB; read_int_from_user() citește o valoare întregă de la intrarea standard)

Discuția este ceva mai amplă. Prerequisites:

- mmap lucrează la nivel de pagină
- mmap nu alocă memorie fizică "din prima"; se alocă la acces (demand paging) în urma unui page fault

Nu am considerat necesar să se observe că a este int și că referirea primei pagini se face cu 0 <= n <= 1024. Au fost considerată validă și observată 0 <= n <= 4096 pentru prima pagină.

Se solicită alocarea a 4100 de octeți (> 4096, < 8192) deci se vor aloca 2 pagini.

Primul read:

- n < 0, probabilitate eroare (SIGSEGV) în cazul în care pagina anterioră este nevalidă (destul de probabil)
- n >= 2048 (peste cele două pagini), probabilitate eroare (SIGSEGV)
- 0 <= n < 1024 page fault și alocare spațiu fizic și validare pagină pentru prima pagină; fără erori
- 1024 < n < 2048 la fel ca mai sus pentru a doua pagină; fără erori (chiar și pentru n >= 1025 (4100/4))

Al doilea read:

- n < 0 idem
- n >= 2048 idem
- n este în aceeași pagină ca mai sus nu se întâmplă nimic
- n în cealaltă pagină atunci page fault, alocare spațiu fizic și validare pagină

Practic mmap(..., 4100, ...) este echivalent cu mmap(..., 8192, ...).

6. Care este avantajul configurării întreruperii de ceas la valoarea de 1ms? Dar la valoarea de 100ms?

1ms

- timp de răspuns scurt, interactivitate sporită, fairness, sisteme desktop (interruperi dese, se diminuază timpul de așteptare 100ms
- productivitate (throughput) sporită, sisteme server (mai puține context switch-uri, mai mult timp pentru "lucru efectiv")

1. Un proces execută la un moment dat:
`sigaction(SIGUSR1, &sa, NULL);`
 iar la un moment ulterior
`write(fd, buffer, 4);`

În care din situație este mai probabilă înlocuirea majorității intrărilor din TLB? Motivați. (Argumentele și modul de folosire a funcțiilor se presupun corecte.)

Problema se tratează cel mai bine de la coadă la cap. Care sunt situațiile în care se înlocuiesc majoritatea intrărilor din TLB (eventual un flush – golire)? Răspuns: În cazul unei schimbări de context. Se schimbă tabelele de pagini între procesul preemptat și cel planificat, mai puțin partea kernel. TLB-ul se golește (dacă arhitectura și/sau sistemul de operare permite) atunci unele intrări rămân active (zone de memorie partajată comun proceselor, zone din spațiul kernel). De aici cuvântul "majorității".

Când se realizează un context switch?

- la terminarea unui proces
- la expirarea cuantel de rulare a unui proces
- în momentul în care prioritățea procesului curent (cel ce rulează) este sub prioritățea unui proces READY
- în cazul blocării procesului curent

În cazul celor două apeluri doar ultima variantă are sens (bloarea procesului curent). Acest lucru se poate întâmpla doar în cazul apelului write, care este un apel blocant.

2. Care este limita de spațiu de swap pentru un sistem cu magistrala de adrese de 32 de biți cu spațiul de adrese împărțit 2GB/2GB (user/kernel). Dar pentru un sistem cu magistrală de adrese de 64 de biți?

Nu există nici o limitare. Singura limitare este cea impusă de hardware.

3. O funcție signal-safe este o funcție care poate fi apelată fără restricții dintă-o rutină de tratare a unui semnal (signal handler). De ce nu este maloc o funcție signal-safe? Ofereți o secvență de cod pentru exemplificare.

După cum s-a menționat în cetea rezolvări (fără a aduce o contribuție în cadrul răspunsului, însă) funcțiile signal-safe sunt practic echivalente cu funcțiile reentrant. O funcție non-signal-safe este o funcție care folosește variabile statice, astfel că, dacă un semnal interupe funcția în programul principal și funcția este rulată în handler este posibil să apară inconsistentă (exact cum se întâmplă în momentul în care un thread este întrerupt și rulează alt thread fără asigurarea accesului exclusiv și consistent la date).

Dacă un semnal interupe funcția malloc și, în handler, rulează funcția malloc structurile interne folosite de libc pentru gestionarea alocării memoriei procesului vor fi datea peste cap. Funcția printf este, în mod similar, o funcție non-signal-safe pentru că folosește buffer-ele interne ale libc. Mai multe informații aici (<https://www.securecoding.cert.org/confluence/x/34A1>)

Scenariul de exemplificare este de forma:

```
void sig_handler(int signo)
{
    void *p = malloc(100);
}

int main(void)
{
    ...
    void *a = malloc(BUFSIZ); /* aici se șosește semnalul */
    ...
}
```

Răspunsul "malloc poate genera SIGSEGV când deja rulează un signal handler" nu este valid. Malloc nu generează SIGSEGV; cel mult rămâne fără memorie și întoarce NULL. SIGSEGV este generat în momentul accesării unei regiuni invalide a memoriei.

4. Un program execută următoarea secvență de cod:

```
for (i = 0; i < BUFSIZE; i++)
    printf("%c", buf[i]);
iar altul
for (i = 0; i < BUFSIZE; i++)
    write(i, buf+i, 1);
Care se va întâmpla? De ce?
```

Funcția printf folosește buffering-ul din libc. Acest lucru înseamnă că, până la înălținirea uneia dintre cele trei acțiuni de mai jos, nu se face apel de sistem:

- se umplu buffer-ele
- se apelăază flush(stdout)

• se transmite newline (\n)
 Apelul de sistem write face apel de sistem de fiecare dată rezultând un overhead important.

Pentru convinsere puteți rula testul de aici (http://anaconda.cs.pub.ro/~ravan/so/test_printf_write.c). Mai jos este un exemplu de rulare, primul folosind printf al doilea folosind write (se altereză macro-ul USE_PRINTF). Rezultatele sunt, în opinia mea, edificatoare.

```
ravan@valhalla:~/school/2008-2009/so/examen$ time ./test_printf_write > out.txt
real 0m0.076s
user 0m0.060s
sys 0m0.020s

ravan@valhalla:~/school/2008-2009/so/examen$ time ./test_printf_write > out.txt
real 0m5.930s
user 0m0.052s
sys 0m5.868s
```

5. Un proces P se găsește în stare READY. Precizați și motivați două acțiuni care determină trecerea acestuia în starea RUNNING.

Fie Q procesul care rulează în acest moment pe procesor. Situații de trecere a lui P din READY în RUNNING:

- Q se încheie și P este primul în coada de procese READY
- lui P îi este crescută prioritatea peste a lui Q
- lui Q îi exprimă cuantă și P este primul în coada de procese READY
- Q execuează o operație blocantă (trece în blocking) și P este primul proces în coada de procese READY

6. De ce obținerea ordinată/ierarhică a lock-urilor previne apariția deadlock-urilor, respectiv apariția fenomenului de starvation?

Ordonarea modului de obținere (achiziție) a lock-urilor în particular și a resurselor în general previne apariția de cicluri în graful de alocare a resurselor și deci apariția deadlock-urilor.

În absența ordinii de obținere un proces P1 poate face Lock(1) și apoi Lock(2). Înainte de Lock(2) este preemptat și procesul P2 face Lock(2) și apoi încearcă Lock(1). Ambele procese rămân blocate în aşteptare mutuală (deadly embrace) = deadlock.

Nu există nici o legătură directă în lock-uri și fenomenul de starvation. Fenomenul de starvation caracterizează o durată de așteptare foarte mare pentru un proces gata de rulare. Alte procese îi iau tot timpul "fata" și procesul nu ajunge pe procesor. Se spune că sistemul nu este "fair" (echitabil). Principala formă de asigurare a echității este folosirea notiunii de cuantă și, în lumea Linux, de epochă și folosirea priorității dinamice a proceselor. Orice formă de locking duce la creșterea nivelului de starvation. Un proces care așteaptă la un semnal într-o regiune critică dar alte procese intră înaintea sa. Asigurarea fairness-ului poate fi asigurată prin strategii de tip FIFO. Dar, obținerea ierarhică a lock-urilor nu are un efect vizibil diferit față de folosirea în orice fel a lock-urilor din perspectiva starvation.

Sisteme de operare

20 iunie 2010

Timp de lucru: 90 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Câte inode-uri va folosi un hard-link către un fișier aflat pe un sistem de fișiere diferit?

Nu se pot crea hard-link-uri către un fișier aflat pe un alt sistem de fișiere. Un hard-link conține un nume și un număr de inode. Inode-uril referitor corespund sistemului local de fișiere, nu altuia sistem de fișiere (nu există un identificator al sistemului de fișiere, se presupune că locul).

2. Fie următoarea secvență de comenzi:

```
touch a.txt
ln a.txt b.txt
ln -s b.txt c.txt
```

Comanda în fără opțiuni creează hard-link-uri, iar comanda ln cu opțiunea -s creează symbolic link-uri. Câte inode-uri, respectiv dentry-uri vor fi create în urma rulării comenzilor de mai sus?

Un hard-link se asociază cu un dentry. La fel un nume de fișier. Un symbolic link are asociat un inode, inode ce conține numele fișierului referit. Se vor crea astfel următoarele:
 touch a.txt → 1 dentry (a.txt) și 1 inode (afferent fișierului proaspăt creat)
 ln a.txt b.txt → 1 dentry (b.txt) și 1 inode (afferent link-ului a.txt)
 ln -s b.txt c.txt → 1 dentry (c.txt) și 1 inode (afferent symbolic link-ului proaspăt creat)

Se creează 3 dentry-uri și 2 inode-uri.

3. Un proces execută secvență următoare în două situații diferite:

```
a = malloc(5000);
memset(a, 0, 5000);
```

Într-ună din situații rezultă două page fault-uri, iar în alta trei page fault-uri. Explicați acest comportament.

Pentru alocarea celor 5000 de octeți se folosește demand-paging. Accesele la acea zonă conduc la generarea de page fault-uri. Se generază un page fault pentru fiecare pagină. Depinzând de alocarea celor 5000 de octeți pot rezulta două sau trei page fault-uri după ce se accesează octetul cu indexul 0, octetul cu indexul 500, octetul cu indexul 4596.

În cazul în care se alocă [4096, 904] pe parcursul a două pagini vor rezulta două page fault-uri după ce se accesează octetul cu indexul 0 și octetul cu indexul 4096.

4. Un program citește un fișier de pe disc, operatie care durează T1. Imediat după prima rulare, se execută din nou programul și durează T2. T2 este semnificativ mai mic decât T1. Cum explicăți?

Citirea unui fișier de pe disc presupune, pe sistemele de operare moderne, interacțiunea cu un subsistem de caching în memorie (buffer cache) al datelor de pe disc. La prima rulare, nu există date în cache-ul din memoria fizică (buffer cache) și toate datele sunt citite de pe disc. La a doua rulare, datele se regăsesc în cache și impuls de citire va fi redus – diferența de acces la memorie făță de disc este mare.

5. Care proces este părintele proceselor zombie?

Un proces zombie este un proces care și-a încheiat execuția dar a căruia stare nu a fost "analiizată" de procesul părinte – adică procesul părinte nu a apelat wait pentru culegerea de informații despre procesul copil. Drepturn, procesul zombie are același proces părinte ca procesul obișnuit înainte să-și încheie execuția – nu există un proces specializat care să fie părintele proceselor zombie.

6. Precizați o situație în care accesarea unei adrese virtuale valide produce segmentation fault.

În cazul în care pagina referată de adresă este marcată de tip read-only (fără a fi vorba de copy-on write), un acces de scriere la acea pagină va genera un page fault. Sistemul de operare analizează tipul de page fault; fiind vorba de un acces de scriere la o adresă dintr-o zonă marcată read-only (non copy-on-write), conchide că este vorba de un acces de tipul "read-only". Rezultă transmiterea unui semnal SIGSEGV (pe un sistem Unix) către procesul care a generat accesul, adică afișarea unui mesaj de tipul "Segmentation fault".

7. Este utilă folosirea "canary value" (stack smashing protection) în cadrul funcției de mai jos? Justificați.

```
void f(char *msg)
{
    char *buffer = malloc(10);
    strcpy(buffer, msg);
}

f("supercalifragilisticexpialidocious");
```

Stack smashing protection se referă la protejarea stivei prin detectarea situațiilor în care adresa de return a unei funcții este probabil să fie suprascrișă. În cazul particular al sevenției de cod de mai sus, se realizează un buffer overflow la nivelul heap-ului, adică la nivelul variabilei buffer (aloctată pe heap folosind malloc). Drepturn, folosirea stack smashing protection nu are nici o utilitate.

8. Un sistem S1 folosește segmentare. Timpul de translatăre a unei adrese virtuale într-o adresă fizică este T1. Un sistem S2 folosește paginare, iar timpul de translatare este T2. Care dintre timpii T1 și T2 este mai mare?

În cazul paginării, translatarea unei adrese virtuale în adrese fizice duce la interrogarea tabelii de pagini, care rezidă în memorie; diminuarea overhead-ului de acces la memorie se realizează prin folosirea TLB. În cazul unei paginări ierarhice timpul de acces este mai mare.

Dacă descriptorii/selectori de segment sunt menținuți în registre ale procesorului atunci timpul T1 este mai mic decât timpul T2.

Dacă descriptorii/selectori de segment sunt menținuți în memorie, atunci T1 este aproximativ egal cu T2 în cazul folosirii unui sistem cu adresare neierarhică și mai mic decât T2 în cazul folosirii unui sistem cu adresare ierarhică.

9. Ce se întâmplă cu sistemul de bază (host) în cazul în care apare o eroare fatală la nivelul nucleului:

- unei mașini virtuale VMware Workstation;
- unui container OpenVZ.

Dacă apare o eroare la nivelul unei mașini virtuale VMware, mașina virtuală trebuie repornită (este într-o stare inconsistentă). Sistemul de bază nu este afectat în vremul fel.

OpenVZ este o soluție de operating system level virtualization. Drepturn, containerele OpenVZ partajează același nucleu de sistem și memoria de operare (Linux) cu sistemul de bază (denumit și container-ul 0). Astfel o eroare de nucleu apărută în nucleul unui container OpenVZ se manifestă la nivelul tuturor container-elor și a sistemului de bază – este, de fapt, improprie exprimarea "nucleul unui container OpenVZ" – nucleul este comun tuturor container-elor și sistemului de bază. O astfel de eroare va fi, deci, fatală și sistemul de bază și acesta trebuie repornit.

10. Fie următoarea două secvențe de programe

<pre>/* S1 */ fd = open("a.txt", O_RDWR O_CREAT, 0644); pid = fork(); if (pid == 0) { write(fd, "a", 1); close(fd); exit(EXIT_SUCCESS); } wait(&status); write(fd, "b", 1);</pre>	<pre>/* S2 */ void *thread_handler(void *arg) { write(fd, "a", 1); close(fd); return NULL; } fd = open("a.txt", O_RDWR O_CREAT, 0644); pthread_create(stid, thread_handler, NULL); pthread_join(&tid, NULL); write(fd, "b", 1);</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

În cazul secvenței S2 apelul write(fd, "b", 1); se întoarce cu eroarea EBADF. Care este explicația? De ce în primul caz nu se întâmplă același lucru?

În cazul sevenței S1, după apelul fork, procesul copil folosește un descriptor propriu (duplicat al descriptorului fd al procesului părinte). Operația close(fd) conduce la închiderea descriptorului doar în procesul copil.

În cazul sevenței S2, thread-ul principal și thread-ul nou creat partajează resursele procesului și, deci, tabela de descriptori a procesului. În consecință, operația close(fd) are sens la nivelul întregului proces și va închide descriptorul. Operația write(fd, "b", 1) executată în thread-ul principal după ce thread-ul creat a închis fișierul folosește un descriptor invalid. Operația va întoarce EBADF (Bad file descriptor).

11. Fie următoarea sevență de operații:

```
for (i = 0; i < N; i++)
    a[i] = 1;
```

Sevența este rulată pe două sisteme diferite care nu dispun de TLB sau memorie cache. Pe un sistem au loc N accese la memoria iar pe un alt sistem 2^N accese. Sevența este identică și rulată în aceleași condiții (același program) pe ambele sisteme. Cu ce differă cele două sisteme?

Sevența de mai sus conduce la N accese la elementele ale vectorului a[i], aflat în memorie. Într-un caz se produc N accese, deci fiecare acces la un element al vectorului înseamnă 1 acces la memoria. În al doilea caz se produc 2^N accese, deci fiecare acces la un element al vectorului înseamnă 2 accese la memoria.

Pentru primul caz (N accese) sistemul nu dispune de memorie virtuală – în acest caz un acces în limbajul C se traduce printr-un acces la memoria fizică.

Pentru al doilea caz (2^N accese) sistemul dispune de memorie virtuală cu adrese neierarhică. Sistemul nu dispune de TLB astfel că fiecare acces la un element al vectorului va însemna un prim acces la tabela de pagini și apoi unul la zona de memorie aferentă elementului, ambele localizate în memorie fizică (RAM) a sistemului.

Sisteme de operare

22 iunie 2010

Timp de lucru: 90 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Care dintre secțiunile de memorie de mai jos sunt proprii unui proces dar nu unui program/executabil? Justificați.
`text, rodata, data, bss, heap, stack`

Un executabil definește secțiunile text, rodata, data și, fără a aloca spațiu, bss. Secțiunea bss este populată cu zero-uri în momentul creării procesului (load-time). Zonele heap și stack (stivă) sunt zone pur dinamice - fin de evoluția procesului – alocație memoriei; alocația pe heap se realizează prin malloc iar alocația pe stack se realizează în contextul apelurilor de funcții.

2. Precizați o situație în care accesarea unei adrese virtuale valide produce page fault, fără a produce segmentation fault.

Dacă adresa este validă, dar pagina fizică nu este prezentă în RAM (este în swap sau a fost alocață folosind demand-paging), va rezulta page fault, și apoi pagina va fi adusă în RAM sau alocată. Dacă pagina este marcată read-only dar de tip copy-on-write (după un fork) atunci un acces de scriere la pagină va conduce la obținerea unei page fault; page fault-ul va conduce la alocația unei pagini fizice noi și marcară acesteia cu drepturi de scriere.

3. Presupunem că avem 3 page frames la dispoziție, toate inițial goale. Se realizează următorul sir de accese (numerele reprezintă pagini virtuale): 3 2 1 0 3 2 4 3 2 1 0 4. Câte page faulturi vor rezulta în urma folosirii algoritmului FIFO? Dar dacă se mărește numărul de page frames la 4?

În tabelul de mai jos, cele 3 linii conțin, respectiv, pagina virtuală aferentă fiecărei pagini fizice (se folosesc 3 frame-uri).

frame1	3	3	3	0	0	0	4	4	4	4	4	4
frame2	-	2	2	2	3	3	3	3	3	1	1	1
frame3	-	-	1	1	1	2	2	2	2	0	0	0

În tabelul de mai jos, cele 4 linii conțin, respectiv, pagina virtuală aferentă fiecărei pagini fizice (se folosesc 4 frame-uri).

frame1	3	3	3	3	3	3	4	4	4	4	0	0
frame2	-	2	2	2	2	2	2	3	3	3	3	4
frame3	-	-	1	1	1	1	1	1	2	2	2	2
frame4	-	-	-	0	0	0	0	0	0	1	1	1

Cu font aldin (bold) au fost marcate paginile virtuale accesate, iar cu font roșu dacă acel acces a generat un page fault. În cazul folosirii a 3 page frame-uri, se obțin 9 page fault-uri, iar în cazul folosirii a 4 page frame-uri se obțin 10 page fault-uri. Acest fenomen poartă numele de anomalie lui Belady (http://en.wikipedia.org/wiki/Belady's_anomaly).

4. Se consideră următoarea schemă de segmentare:

Segment	Base	Length
0	100	1000
1	1200	250
2	1800	300
3	2200	500
4	3000	800

Care dintre următoarele reprezintă adrese logice valide? Adresele sunt de forma (segment, offset) .

- a. (0, 820)
b. (1, 430)
c. (2, 13)

În cazul opțiunilor prezентate contează dacă offsetul în cadrul segmentului depășește dimensiunea segmentului (length). Se observă că doar a doua opțiune (b) conduce la depășirea lungimii segmentului (430 > 250), deci nu reprezintă o adresă logică validă.

logică validă.

5. Dați exemplu de situație în care operația lock(&mutex) conduce la invocarea scheduler-ului și un exemplu de situație în care nu conduce la invocarea scheduler-ului.

Dacă apelul este blocant va conduce la invocarea scheduler-ului. Dacă apelul nu este blocant în momentul în care mutex-ul este deja achiziționat. Apelul este neblocant dacă mutexul nu este achiziționat (adică este liber). În caz particular, dacă mutex-ul este implementat în user space (de tip futex) și este liber, nu va genera apel de sistem și nu există "riscul" de planificare a acestuia din cauza priorității proceselor sau a altor euristici de planificare ale nucleului.

6. Un sistem de fișiere dispune de un bitmap pentru inode-uri (un bit specifică folosirea sau nu a unui inode) de 32KB. Câtă symlink-uri pot fi create? (este suficient ordinul de mărime și justificarea răspunsului)

*$32KB = 32 * 2^{10} * 8biti = 256 * 2^{10}$ intrări în bitmap. Pot fi create $256 * 2^{10}$ inode-uri. Întrucât un symbolic link ocupă un inode pot fi create $256 * 2^{10}$ inode-uri. Se pot scădea cîteva inode-uri aferente directorului rădăcină și fișierelor reale către care punctează symbolic link-urile.*

7. Se dă următoarea sevență de execuție :

Thread A	Thread B
-----	-----
work_a1	work_b1
work_a2	work_b2

Realizăți sincronizarea celor 2 fire de execuție folosind semafoare astfel încât work_a1 să se execute înainte de work_b2 și work_b1 să se execute înainte de work_a2. Folosiți primitivele: `sem_init(sem_t *sem, int count)`, `sem_up(sem_t *sem)`, `sem_down(sem_t *sem)`.

Presupunem folosirea a două semafoare (s_a și s_b) cu următoarele roluri:

- * s_a, thread-ul A îajuns la punctul de întâlnire
* s_b, thread-ul B îajuns la punctul de întâlnire.

Soluția de sincronizare este cea de mai jos:

Thread A	Thread B
-----	-----
work_a1	work_b1
sem_up(&s_a);	sem_up(&s_b);
sem_down(&s_b);	sem_down(&s_a);
work_a2	work_b2

8. Într-o aplicație multi-threaded există două threaduri. Primul execută o funcție CPU intensive, iar celălalt execută preponderent operații I/O. De ce folosirea implementării threadurilor la nivel user este de dorit?

În cazul unei implementări de thread-uri la nivel user, bloarea unui thread conduce la blocarea întregului proces. Thread-ul care execută operații I/O va avea parte de situații deosebite de blocaje (operațiile I/O sunt, în general, blocante). Blocarea acestui thread va conduce la blocarea întregului proces. În acest caz, thread-ul CPU intensive, deși ar putea rula și executa acțiuni utile, este blocat. Această lucru duce la folosirea necorespunzătoare a procesorului: un thread este pregătit pentru execuție (READY) dar nu poate rula. Pe un sistem cu implementare de thread-uri la nivel kernel, acest lucru nu are avă loc.

9. De ce, înainte de a realiza un apel exec(), e recomandat să se închidă toate fișierele de care nu are nevoie procesul copil?

*Un proces copil mosteneste descriptorii procesului părinte. Acest lucru atrage două dezavantaje importante:
* securitate: un proces poate citi, parcurge sau copia date din fișierele unui alt proces
* resurse: menținerea descriptorilor deschisi duce la ocuparea unui număr mare de descriptori de fișier; în cazul în care se creează procese în continuare, tabela de descriptori de fișier este ocupată în mare măsură de fișiere deschise de alte procese*

10. Care este numărul minim de apeluri de sistem generate de următoarea sevență de pseudocod? (toate apelurile de funcții se

Sisteme de operare

22 iunie 2010

Timp de lucru: 90 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Care dintre secțiunile de memorie de mai jos sunt proprii unui proces dar nu unui program/executabil? Justificați.

`text, rodata, data, bss, heap, stack`

Un executabil definește secțiunile text, rodata, data și, fără a aloca spațiu, bss. Secțiunea bss este populată cu zero-uri în momentul creării procesului (load-time). Zonele heap și stack (stivă) sunt zone pur dinamice - fin de evoluția procesului – alocație memoriei; alocația pe heap se realizează prin malloc iar alocația pe stack se realizează în contextul apelurilor de funcții.

2. Precizați o situație în care accesarea unei adrese virtuale valide produce page fault, fără a produce segmentation fault.

Dacă adresa este validă, dar pagina fizică nu este prezentă în RAM (este în swap sau a fost alocată folosind demand-paging), va rezulta page fault, și apoi pagina va fi adusă în RAM sau alocată. Dacă pagina este marcată read-only dar de tip copy-on-write (după un fork) atunci un acces de scriere la pagină va conduce la obținerea unei page fault; page fault-ul va conduce la alocația unei pagini fizice noi și marcară acesteia cu drepturi de scriere.

3. Presupunem că avem 3 page frames la dispoziție, toate inițial goale. Se realizează următorul sir de accese (numerele reprezintă pagini virtuale): 3 2 1 0 3 2 4 3 2 1 0 4. Câte page faulturi vor rezulta în urma folosirii algoritmului FIFO? Dar dacă se mărește numărul de page frames la 4?

frame1	3	3	3	0	0	0	4	4	4	4	4	4
frame2	-	2	2	2	2	2	2	2	3	3	3	3
frame3	-	-	1	1	1	1	1	1	1	2	2	2

În tabelul de mai jos, cele 4 linii conțin, respectiv, pagina virtuală aferentă fiecărei pagini fizice (se folosesc 4 frame-uri).

frame1	3	3	3	3	3	3	4	4	4	4	0	0
frame2	-	2	2	2	2	2	2	2	3	3	3	3
frame3	-	-	1	1	1	1	1	1	1	2	2	2
frame4	-	-	-	0	0	0	0	0	0	1	1	1

Cu font aldin (bold) au fost marcate paginile virtuale accesate, iar cu font roșu dacă acel acces a generat un page fault. În cazul folosirii a 3 page frame-uri, se obțin 9 page fault-uri, iar în cazul folosirii a 4 page frame-uri se obțin 10 page fault-uri. Acest fenomen poartă numele de anomalie lui Belady (http://en.wikipedia.org/wiki/Belady's_anomaly).

4. Se consideră următoarea schemă de segmentare:

Segment	Base	Length
0	100	1000
1	1200	250
2	1800	300
3	2200	500
4	3000	800

Care dintre următoarele reprezintă adrese logice valide? Adresele sunt de forma (segment, offset) .

- a. (0, 820)
b. (1, 430)
c. (2, 13)

În cazul opțiunilor prezентate contează dacă offsetul în cadrul segmentului depășește dimensiunea segmentului (length). Se observă că doar a doua opțiune (b) conduce la depășirea lungimii segmentului (430 > 250), deci nu reprezintă o adresă logică validă.

5. Apelul de bibliotecă write conduce la invocarea apelului de sistem aferent (sys_write per Linux).

În consecință, numărul minim de apeluri de sistem generate este 1 (unu), generat de apelul write.

6. De ce nu se poate implementa un mecanism de memorie partajată pentru un sistem cu paginare inversată?

Într-un sistem cu paginare inversată, intrările din tabela de pagini conțin PID-ul procesului și pagina virtuală aferentă. Indeksul intrării în tabelă reprezintă frame-ul aferent. Partajarea unei pagini se poate realiza în măsură în care se poate asocia unui frame (unei pagini fizice) mai multe pagini virtuale. Într-un sistem cu paginare inversată, o singură pagină virtuală poate corespunde unei pagini fizice, și nu se poate implementa partajarea memoriei.

Dacă sistemul permite alocația unei liste de elemente de tip (PID, pagină virtuală) în cadrul unei intrările din tabela de pagini atunci partajarea memoriei se poate implementa, cu dezavantajul unui timp de căutare ridicat (problemă care se poate rezolva prin folosirea de tabele hash).

7. De ce, înainte de a realiza un apel exec(), e recomandat să se închidă toate fișierele de care nu are nevoie procesul copil?

Un proces copil mosteneste descriptorii procesului părinte. Acest lucru atrage două dezavantaje importante:

** securitate: un proces poate citi, parcurge sau copia date din fișierele unui alt proces*

** resurse: menținerea descriptorilor deschisi duce la ocuparea unui număr mare de descriptori de fișier; în cazul în care se creează procese în continuare, tabela de descriptori de fișier este ocupată în mare măsură de fișiere deschise de alte procese*

8. Care este numărul minim de apeluri de sistem generate de următoarea sevență de pseudocod? (toate apelurile de funcții se

Timp de lucru: 90 de minute

NOTĂ: toate răspunsurile trebuie justificate

1. Câte procese copil, respectiv părinte poate avea un proces la un moment dat?

Un proces poate avea, la un moment dat, un singur proces părinte și oricătre procese copil, în limita resurselor sistemului. Procesul inițial poate fi considerat un proces particular care nu are un proces părinte.

2. Un proces execută sevență:

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

În timpul execuției sevenței, procesul este preemptat și este planificat alt proces. Dați exemplu de o situație care poate genera preemptarea.

Intrucăt secțiunea de mai sus nu este blocantă, procesul poate fi preemptat dacă îl exprimă cantitatea de timp sau dacă în sistem există un proces cu prioritate superioară pregătit pentru execuție. Această situație este declanșată de apariția întreruperii de ceas.

3. Dați exemplu de o funcție thread safe dar non-reentrantă. Explicați.

O funcție thread safe dar non-reentrantă permite rularea acesteia în context multithreaded dar nu permite existența simultană a două fluxuri de execuție în contextul același thread/proces. Un exemplu este o funcție care folosește locking. De forma:

```
int my_function(void)
{
    lock(&mutex);
    /* TODO */
    unlock(&mutex);
}
```

4. Se consideră următorul cod:

```
void f()
{
    int *z = malloc(sizeof(int));
    [...]
    printf("z = %p\n", z);
    printf("&z = %p\n", &z);
}
```

După rularea secțiunii se afișează mesajul:

```
z = 0x12345678
&z = 0x87654321
```

Asociați adresele z, &z cu secțiunile spațiului de adresă al unui proces: .text, .data, .bss, heap și stack.

z este o variabilă de tip pointer – conținutul acesteia este o adresă. Adresa punctează către o zonă din heap (fiind rezultatul întors de apelul malloc).

&z reprezintă adresa variabilei v. Variabila este o variabilă locală unei funcții deci este alocată pe stivă.

Avea o variabilă alocată pe stivă (adresa ei este o adresă din stivă), iar conținutul acelei variabile (pointer) este o adresă întoarsă de apelul malloc, adică o adresă din heap.

5. Explicați modul în care se poate produce starvation pe un sistem cu planificare SRTF (Shortest Remaining Time First).

Dacă în cadrul sistemului apar în coada ready procese cu timp de rulare redus, acestea vor fi planificate prime. Presupunând un flux continuu de procese cu timp de rulare redus, procesele cu timp de rulare mare vor ajunge să se execute foarte rare sau deloc, adică să se producă fenomenul de starvation.

6. După schimbarea contextului între două thread-uri, care clase registre au valori diferite (înainte și după schimbarea de context): registrele generale, registrul de stivă, registrele de segment.

La schimbarea de context între două thread-uri, majoritatea registrilor se schimbă. Fiecare thread dispune de valori proprii ale registrilor. Astfel, registrele generale și registrul de stivă se schimbă. Sistemele de operare moderne folosesc rău registrele de segment, astfel că în general acestea nu vor fi schimbate.

În plus, anumite registre interne procesorului, inaccesibile din user space (ring3 pe arhitectură x86) își pot păstra valorile în cazul thread-urilor diferite (registre precum cr2, cr3 pe arhitectură x86).

7. De ce anume zone din bibliotecile partajate sunt mapate read-write? Date un exemplu.

Bibliotecile partajate conțin zone r-x (cod), r- (read-only data) și rw- (date). Zonele read-write conțin variabile care pot fi scrisă de procesul ce folosește biblioteca, precum variabila errno în cazul bibliotecii standard C.

8. Exceptând apelurile de sistem, dați exemplu de situație în care procesorul comută în kernel space.

Procesorul execută cod kernel în cazul unor solicitări din user space (apel de sistem) sau de la hardware (întreruperi). Procesorul execută, astfel, cod kernel, exceptând apelurile de sistem, în momentul sosirii unei întreruperi.

9. Un sistem dispune de N procese. Fiecare proces dispune de M pagini virtuale neallocat. Sistemul dispune de o singură pagină fizică disponibilă. Care este numărul maxim de pagini virtuale care pot fi asociate cu pagina fizică?

*Oricătre pagini virtuale pot fi asociate cu o pagină fizică. Implementările de tip mmap permit maparea unei zone de memorie virtuale peste o zonă de memorie fizică. În situația de mai sus, un proces poate mapea toate cele M pagini virtuale peste aceea pagină fizică. În total, pentru cele N procese, se pot mape N*M pagini virtuale.*

10. Explicați de ce nu se poate implementa mechanismul de swapping pe un procesor fără unitate de management al memoriei.

Unitatea de management a memoriei (MMU) este responsabilă cu translatarea paginilor virtuale în pagini fizice. Absența MMU conduce la absența mecanismului de memorie virtuală. Mechanismul de memorie virtuală permite existența unui spațiu virtual de adrese care depășește spațiul fizic – spațiu suplimentar poate fi furnizat de disc, prin intermediul swap-ului. În cazul absenței mecanismului de memorie virtuală, nu se poate refera la memoria virtuală, deci nu se poate implementa mechanismul de swapping (evacuare pe disc și recuperarea paginilor de pe disc).

11. Un inode dispune de 10 pointeri de indirectare simplă a blocurilor de date. Un bloc ocupă 4096 de octeți. Știind că un director ocupă 64 de octeți, câte întrări poate avea maxim un director?

10 pointeri de indirectare simplă punctează către blocuri care conțin, la rândul lor, pointeri. Considerând că un pointer ocupă 4 de octeți, rezultă că un bloc de pointeri conține $4096/4 = 1024$ de pointeri.

Cei 10 pointeri de indirectare simplă vor referi 10 blocuri care conțin, la rândul lor, 10×1024 pointeri adică 10240.

Fiecare dintre cei 10240 pointeri punctează către un bloc de date. Fiecare bloc de date ocupă 4K. Rezultă astăzi, că un inode poate referi $10240 \times 4KB$ de date.

În cazul unui director, datele sale sunt un vector (array) de dentry-uri. Numărul maxim de dentry-uri se obține împărțind spațiul maxim ce poate fi referit de un inode la dimensiunea unui dentry. În consecință, un director poate conține $10240 \times 4KB / 64$ dentry-uri, adică $10240 \times 64 = 655360$.

1. Două procese (P1, P2) folosesc o zonă de memorie partajată, rulează într-un sistem preemptiv și execută sevența de cod de mai jos. Știind că valoarea inițială indicată de pointerul counter este 0 și că acesta indică în zona de memorie partajată, care este valoarea indicată de pointerul counter la finalul execuției celor două procese?

P1	P2
if (*counter == 0) (*counter)++;	if (*counter == 0) (*counter)++;

Dacă cele două procese se execută sevențial atunci primul proces va incrementa variabila pe 1, iar al doilea nu va executa instrucțiunea de incrementare din if. Valoarea finală va fi 1.

În varianta în care procesul P1 este preemptat de procesul P2 după verificarea if (*counter == 0), dar înainte de (*counter)++ atunci procesul P2 va testa, la rândul său, ca fiind adeverată condiția (*counter == 0). În consecință, va incrementa valoarea contorului. P1 va incrementa, de asemenea, valoarea contorului, rezultând valoarea finală 2.

2. Pe un sistem dual processor cu 128MB de RAM și swap de 256MB rulează un sistem Linux. Câte procese se pot găsi la un moment dat în starea RUNNING, READY respectiv WAITING?

Caracteristica ce influențează numărul de procese din starea RUNNING este numărul de unități de execuție. Fiind vorba de un sistem dual processor, pot exista maxim 2 procese în starea RUNNING. În coada READY și WAITING se pot găsi oricătre procese, ilimitând numărul de stivele celor două procese.

3. Un sistem folosește TLB în care fiecare intrare conține trei câmpuri (pid, frame, pagină virtuală). Care este avantajul acestor implementări față de o implementare care conține doar două câmpuri (frame, pagină virtuală)?

Prezența câmpului pid înseamnă că se poate realiza o selecție după proces a intrărilor din TLB. Acest lucru este util în cazul schimbării de context, în cazul unei schimbări de context cea mai mare parte a intrărilor din TLB sunt anulate. Prezența unui câmp pid înseamnă că, în cazul unei schimbări de context, doar intrările specifice procesului vor fi eliminate rezultând într-un număr mai mic de acces directe la tabela de pagini.

4. Descrieți în pseudocod cum se poate determina dacă stiva crește de la adrese mari la adrese mici sau invers.

O soluție fină constă în construirea stack frame-urilor pentru funcții:

```
int *p_fcaller;
int *p_fcalled;
void f2(void)
{
    int f2_local;
    p_fcalled = &f2_local;

    if (p_fcaller > p_fcalled)
        printf("stack grows up\n");
    else
        printf("stack grows down\n");
}
void f1(void)
{
    int f1_local;
    p_fcaller = &f1_local;
    f2();
}

Exemplu complet aici: http://swarm.cs.pub.ro/git/?p=razvan-code.git;a=blob;t=tests/stack\_grow.c;h=3ebc0408a4c30cb8317714fb5c473c35ca7bb3d;hb=21d876d26d3db2145f6be96423cb14a7ddaf21a
```

5. Câte pagini fizice vor ocupa stivele celor două procese rezultante în urma apelului fork exact înainte de apelul exit? Apelul fork se întoarce cu succes.

```
int main(void)
{
    char buf[3 * PAGE_SIZE];
    buf[0] = 'a';
    buf[PAGE_SIZE] = 'z';
    fork();
    buf[0];
    exit(EXIT_SUCCESS);
}
```

Bufferul buf este alocat folosind demand paging. Acest lucru înseamnă că nu vor fi alocate pagini fizice până în momentul accesului. După buf[0] = 'a' și buf[PAGE_SIZE] = 'z' rezultă două page fault-uri care vor genera alocare a două pagini fizice.

După fork paginile sunt marcate copy-on-write.

După apelul fork() atât procesul părinte cât și procesul copil accesează buf[0]. Acest lucru va rezulta într-un page fault (se dupliquează pagina și pagina nouă și pagina veche primesc drept de scriere). Se va aloca astfel o pagină fizică nouă.

În final stivele celor două procese vor ocupa 3 pagini fizice.

6. În urma rulării sevenței de cod de mai jos fișierul a.txt conține sirul 222313. După fiecare apel write actualizați următorul vector (cursor fd1, cursor fd2, cursor fd3, conținutul fișier). Exemplu: înainte de primul write vectorul va fi (0, 0, 0, _)

```
fd1 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
fd2 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
fd3 = dup(fd1);

write(fd1, "1", 1);
write(fd2, "2", 1);
write(fd3, "3", 1);
break;

pid = fork();

switch (pid) {
    case 0:
        write(fd1, "1", 1);
        write(fd2, "2", 1);
        write(fd3, "3", 1);
        break;
    default:
        wait(&status);
        write(fd1, "1", 1);
        write(fd2, "2", 1);
        write(fd3, "3", 1);
        break;
}

/* continuă pe coloana a doua */
```

	initial	w1	w2	w3	w4	w5	w6	w7	w8	w9
f1.cursor	0	1	1	2	3	3	4	5	5	6
f2.cursor	0	0	1	1	1	2	2	3	3	3
f3.cursor	0	1	1	2	3	3	4	5	5	6
continut	--	"1"	"2"	"23"	"231"	"221"	"2213"	"22231"	"222313"	"222313"

Procesul părinte și fiu partajează descriptorii de fișier și cursorul de fișier. Orice modificare a cursorului în procesul copil va fi vizibilă în procesul părinte și invers.

fd3 partajează cursorul de fișier cu fd1 ca urmare a apelului dup. Orice incrementare a cursorului folosind descriptorul fd1 va fi vizibilă descriptorului fd3 și invers.

<p>S throughput de date Nume și grupă: și cum variază aceste trei mărimi una în funcție de celalătă.</p> <p>Descrieți cum realizati o arhitectură de sistem (topologie, legături, scenarii de folosire) care să poată măsura acesti parametri. Elaborați având în vedere faptul că software-ul de server rulează pe un calculator sever mult mai performant decât calculatoarele pe care le aveți la dispozitiv pentru testare.</p> <p>12. (2.5 puncte) Pentru aceeași situație ca la punctul anterior, descrieți arhitectura software a software-ului de măsură: single threaded, multi-threaded, multi-proces? Ce alti parametri ati mai putea măsura?</p>	<p>G</p>
	3

S Nume și grupă:
10 luni 2013

1. (1 punct) Stând că **overhead-ul unui apel de sistem este de 7 ms și overhead-ul tratarii unui page fault este de 2 ms**, în ce situație un apel memcpy va dura mai mult decât un apel de sistem?

2. (1 punct) De ce este o eroare de tip *Segmentation fault* urmată de o schimbare de context? Pentru că după ce se rulează handler-ul întretreruji de segfault, fie se va păsa 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 în cadrul unui masini virtuala VMware Workstation, dar nu în invers?

4. (1 punct) Precizați și justificați valoarea de adevăr a următoarei afirmații: *Un apel write blocant apelat de un proces multithreaded cu implementare în 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 bufferarea.

5. (1 punct) Un proces execută sevența:

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

În ce situație poate genera această sevență un TLB flush?

6. (1 punct) Majoritatea planificatoarelor I/O (*I/O scheduler-ur*) realizează operații de tip *sorting and merging* pe cererile de lucru 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 că SSD-urile sunt construite pe o cu totul altă tehnologie față de HDD-uri, tehnologia în care nu se folosește un cap de cirec deci nu există latența la seek, algoritmul de disk scheduling nu își are rostul. În orice ordine s-ar realiza operațiile, overhead-ul și același.

7. (1 punct) Creați un paragraf adevarat, informativ și argumentat din domeniul sistemelor de operare care să cuprindă ca subiecte principale notiunile de ASLR (*Address Space Layout Randomization*) și *shellcode*.

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

8. (1 punct) Creați un paragraf adevarat, informativ și argumentat din domeniul sistemelor de operare care să cuprindă ca subiecte principale notiunile de *stack overflow/stack thread*.

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

Sistem (2,5 puncte) Nume și grupă:

Un program are nevoie să stocheze 1.000.000 de fisiere pe disc, fiecare de 10MB, care să poată fi accesate pe bază de identificatori numerici unici. Alături schema bloc a unui sistem de stocare care să ofere suport pentru acest volum de informații și detaliiți blocul de identificare a fisierului pe disc.

Aici problema constă în faptul că nu este scalabil să stochezi 1.000.000 de fisiere în același director, și nici într-o baza de date nu prea are sens să stochezi ca bloburi toata povestea asta care însumează 10TB de date. Asadar, e clar că fisierile vor trebui stocate pe disc în foldere separate, cel mai bine într-o strucție arborescentă, care să se poată întinde pe oricâte filesystemuri, deoarece și vorba de o capacitate mai mare decât discurile uzuale. Asadar, blocurile implicate în această soluție sunt două:

- 1. Bloc de identificare
- 2. Bloc de stocare

Blocul de identificare se poate implementa cu o tabelă simplă în 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 îl se cere, pentru un nou fisier de stocat, locul în care se va stoca. Pentru a face asta, trebuie să stie că discuri există în sistem, ce capacitate au și ce fișiere sunt, pentru a determina un număr optim de intrări în director pentru fiecare dintre ele. Cand se cere un slot pt un fisier nou, sistemul caută pe volumele existente cel mai bun loc în termeni de spațiu disponibil, intran în director, etc. Se poate implementa și un load balancer care să distribuie fisierile cele mai cerute pe discuri diferite, etc.

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă:

Semnătură:.....

Sistem (2,5 puncte) Nume și grupă:

Sisteme de Operare

14 iunie 2013

Timp de lucru: 100 de minute

Notă: Toate răspunsurile trebuie justificate

1. (1 punct) Fie T1 timpul de mutare a unui fisier pe aceeași partită și T2 timpul de mutare a aceluiași fisier pe o altă partită. Ce relație există între T1 și T2 și de ce?

T1 < T2

In cazul in care se muta un fisier pe aceeași partită, tot ce se întâmplă este crearea unui nou hard link între inode-ul referentiat de fisier și stergerea vecchii link. În situația în care se mută fisiere între partită, datele chiar se mută. Astfel, T2 este mai mare deoarece are overhead de la mutarea datelor.

2. (1 punct) Bibliotecile partajate contin zone de memorie mapate în cadrul proceselor cu următoarele permișuni: r-x (cod), r-- (read-only data) și rw- (date). Care din aceste zone NU trebuie să fie partajate între mai multe procese care folosesc aceeași bibliotecă și de ce?

3. (1 punct) Două sisteme identice din punct de vedere al hardware-ului și al sistemului de operare sunt folosite în scopuri diferite. Primul este folosit pentru calculul de transformă Fourier, iar al doilea este folosi drept server (web, e-mail, ssh, dns, dhcp, etc.). Care dintre cele două va petrece mai mult timp în schimbări de context și de ce?
Al doilea!

There are three potential triggers for a context switch:

Multitasking([edit](#))

Most commonly, within some [scheduling](#) 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](#) or [synchronization](#) operation to complete. On a [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](#). This interrupt ensures that the scheduler will gain control to perform a context switch.

Interrupt handling([edit](#))

Modern architectures are [interrupt](#) driven. This means that if the CPU requests data from a disk, for example, it does not need to [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](#) 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](#) does not spawn or schedule a special

8

7

Sistem Nume și grupă:

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](#))

When a transition between [user mode](#) și [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) În ce situație pot două procese copila ale aceluiași proces să aibă același PID?

Neither `fork()` nor `vfork()` keep the same PID although `clone()` can in one scenario^(*). 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.

^(*)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](#).

5. (1 punct) Două procese scriu și citesc un fisier, în mod sincronizat, prin următoarea secvență de pseudo-cod:

Sistem Nume și grupă:

Nume și grupă:

```
acquire_mutex (&m);
if (condition)
    write_to_file (content);
else
    read_from_file(content);
release_mutex(&m);
```

Care este numărul minim de apeluri de sistem care sunt generate în secvența de pseudo-cod de mai sus?

6. (1 punct) La un moment dat un proces accesează o adresă de memorie, fără a rezulta page fault. După un timp, accesează din nou acea adresă și rezultă page fault. Stând că între cele două accese descrite nu au existat alte accese la pagina aferentă, explicati de ce s-a produs acest page fault.

7. (1 punct) Creati un paragraf adevărat, informativ și argumentat din domeniul sistemelor de operație care să cuprindă ca subiecte principale notiunile de `handler de semnal` și `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-ului de semnal întrerupe programul principal în timp ce acesta încă este ocupat mutex-ului.

8. (1 punct) Creati un paragraf adevărat, informativ și argumentat din domeniul sistemelor de operație care să cuprindă ca subiecte principale notiunile de `pagina fizică (frame)` și `fisier mapat în memorie`.

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

9. (1 punct) Care sunt asociările dintre sectiunile de memorie de mai jos și programe/executabile, procese, respectiv thread-uri? (unul la unul, unul la mai multe, etc.)

text, rodata, data, bss, heap, stack

10. (1 punct) Asociati concepțele de mai jos cu soluțiile de virtualizare VMware Workstation, KVM, LXC: kernel development, bare-metal virtualization, modul de kernel, native virtualization, full virtualization, containers, disk image file.

Soluție (2.5 puncte) Un programator implementează o bază de date care va stoca obiecte de **Nume si grupă** 1MB. Fiecare obiect are un identificator numeric unic (ID), și este salvat pe disc (HDD) ca un fișier separat, folosind sistemul de fisiere ext2. Numărul total de obiecte poate fi foarte mare, fără limită doar de dimensiunea HDD-ului (e.g. 10TB = 10 milioane de fisiere).

Baza de date trebuie optimizată astfel încât să acceseze cât mai repede un fișier identificat cu un ID.

Vă se cere să sugerați optimizări posibile pentru a atinge acest scop. Explicați care sunt factorii care limitează performanța, și argumentați optimizările alese. Desenați o schemă bloc a sistemului propus.

Pentru deschiderea unui fisier și nevoia sa localizam fisierul în dentry - costul este liniar în numărul de fisiere în ext2. Ideea de bază: ca să reducem timpul de acces, trebuie să ne asigurăm că subiectele sunt stocate în directoare care au un număr redus de fisiere.

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

- o tabelă de hash care mapează ID-ul fisierului în directorul care îl conține. Această tabelă va fi stocată în memorie. Presupunând că numărul de directoare este relativ mic, dimensiunea tabelii este data de nr de fisiere * (dim_id + pointer_nume_director) ~ 8 sau 16 octetii. 10 milioane de fisiere ar ocupa numai 160MB de RAM.
- o ierarhie de directoare în care fiecare director are un număr maxim de intrări (prestabilit).

Se pot folosi multiple structuri ierarhice cu adâncimea prestabilită. Se crează astfel mai multe directoare:

- un director "direct" care conține un număr de fisiere X
- un director "indirect" care conține X alte directoare fiecare cu X fisiere
- un director dublu indirect
- un director triplu indirect etc.

Să vor folosi directoarele "directe" după care cele indirecte, dublu-indirecte, etc.

Cum alegem X? Timpul de acces la un fisier depinde de X și de adâncimea structurii de directoare, cîtreia unui director durează: $X(\log N + 1)$. Stînd N (e.g. 10 milioane), se poate optimiza X alegând valoarea care minimizează formula de mai sus.

12. (2.5 puncte) Clientii YouTube se conectează la servere via TCP și transmit o cerere care conține numele fisierului dorit, offset-ul de început și dimensiunea dorită (tipic, câteva zeci de KB, pentru a evita download-ul continuu). În mod inutil, dacă utilizatorul încide clipul. Dacă serverul încide conexiunea, clientul se va reconecta atunci când are nevoie de următoarea secvență de clip și va relua download-ul. Altfel, clientul tine conexiunea deschisă și doar va emite o nouă cerere.

Vă se cere să implementați un server YouTube care să permită unui număr cât mai mare de utilizatori să primească clipuri simultan, astfel:

- desenati o schemă bloc cu sistemul propus;
- explicați cum va trata fiecare client (proces nou / thread / etc.) și motivati alegerea făcută;

11

Soluție Explicație API văd folosi pentru a citi / scrie din socket TCP (nonblocking, event-based, blocking)? și de ce;

- atunci când crește numărul de clienti, ce resurse vor limita scalabilitatea sistemului pe care îl-ati construit? (e.g. nr. de descriptori, nr. de procese/thread-uri, retea/u?)

Ideea principală este de a trata o cerere de continut cât mai repede și de a încide conexiunea – astfel numărul de clienti conectați simultan este mic și astfel lucru reduce stresul asupra resurselor sistemului (thread-uri, descriptori).

Se poate folosi un pool de thread-uri; atunci cand vine o cerere nouă se aloca cerere una din thread-urile din pool. Folosirea pool-ului de thread-uri minimizează costurile de startup, și reduc costurile de switching (no tlb flush), însă toti clientii vor folosi același spațiu de adresă – totuși potențialele probleme de securitate par minore (read-only video).

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

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimăn să copieze la această lucrare.

Nume și grupă:

Semnătură:.....

Sisteme de Operare

27 mai 2013

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. Comanda "ulimit -s unlimited" stabilește ca dimensiunea stivei să fie maximă posibilă. De ce această comandă poate fi folosită pentru a înlăubi atacuri de tip return-to-libc pe sisteme cu suport de ASLR?

Cu cat stiva este mai mare cu atât posibilitatea de pozitionare a acesteia în spațiu de adresa scăd. Astfel, se poate face brute force mai usor.

2. Care este asocierea dintre notiunile de spațiu de adresă și proces? Dar între spațiu de adresă și thread-uri?

Fiecare proces are propriul spațiu de adresa → (spațiu de adresa, proces) - unu la unu Threadurile unui proces partajează același spațiu de adresa (spațiu de adresa, thread) - unul la mai multe

12

Soluție **Nume și grupă:**

3. Folosind mecanismul de memorie virtuală, se pot partaja zone din spațiu de adresă din user space al unui proces cu zona din spațiu kernel. Dati exemplu de situație în care o astfel de abordare este utilă.

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 numărul de intrări în TLB este de ordinul sutelor? De ce nu se alocă dimensiuni mai mari pentru TLB (număr de intrări de ordinul miliarilor sau zecelei 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.

13

Soluție **Nume și grupă:**

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_mmuvld 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 preferă folosirea spinlock-urilor pentru regiuni critice de dimensiuni mici iar mutex-urile pentru regiuni critice de dimensiuni mari?

Deoarece spinlock-ul folosește busy-waiting este indicat să fie folosit pe portiuni mici unde timpul de asteptare este mic și 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 decât asteptarea într-o coadă pana la un unlock pe mutex.

6. Care este legătura și diferența dintre notiunile de "drepturi de crea unui fisier" și "drepturi de deschidere a unui fisier"?

7. Care este un avantaj al fiecărei dintre formele de virtualizare: full virtualization și paravirtualization?

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

14

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 reprezintă un server web are prioritățe mai mari decât un proces care este folosit pentru înmulțire de matrice?

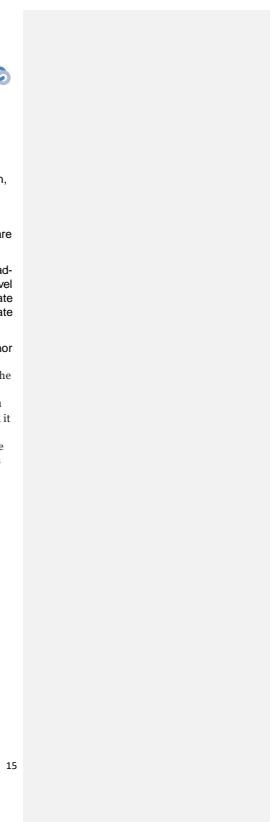
CPU INTENSIVE VS I/O
9. Procesul P folosește thread-uri hibride, având 9 thread-uri user-level, mapate pe 3 thread-uri kernel-level (câte trei user-level threads pe un kernel-level thread). Fiecărui thread-user-level TU1 îi corespund 3 thread-uri kernel-level TK1. În ce mod vor fi efectuate schimbările thread-urilor mapate pe TK1, în cazul în care TU1 realizează un acces invalid la memoria? Dar thread-urile mapate pe celelalte thread-uri kernel?

10. De ce este necesară operația de "Safe Remove" a dispozitivelor USB, în urma copierii unor fisiere 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.

În conformitate cu ghidul de etică al Catedrei de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimăn să copieze la această lucrare.

Nume și grupă:



Sisteme de Operare

24 mai 2014

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Precizați o diferență între un thread și un proces.

Procesele au propriul spațiu de adresa pe care cind threadurile unui proces împart același spațiu 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 legătura între memorie virtuală și spațiu de swap?

Memoria virtuală este memoria pe care o vede un proces care rulează în sistem. Pentru că în RAM nu încapă 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) Dati exemplu de dispozitiv de tip caracter. De ce este acesta un dispozitiv de tip caracter?

Tastatura este un dispozitiv de tip caracter pentru ca transferul de date se realizează 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) Descrieți, în pseudocod sau literal, un scenariu în care are loc un atac de tipul buffer overflow. Precizați în ce condiții se realizează acest atac.

exemplu, avem urmatorul program:

```
#include <stdio.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; }

Dacă rulăm programul cu parola: thegeekstuff, se produce ceea ce ne așteptam, și primim drepturi de root.

Acest program însă are posibilitatea de a produce buffer overflow. Funcția gets() nu verifică marginile unde scrie și poate scrie string-uri de lungime mai mare decât marimea celui în care scrie el de fapt. Asadar, dacă atacatorul dă o parola mai lungă decât marimea buffer-ului, acesta poate să ajungă să scrie peste zona de memorie a variabilei "pass", devenind ceva diferit de 0 => drepturi de root pentru atacator.

5. (7 puncte) Un fișier are două link-uri hard (a.txt și b.txt). Ce se întâmplă dacă stergem unul dintre link-urile (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 întrebare soft links vs hard links:

Underneath the file system files are represented by inodes (or 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/overwritable) 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 necesară prezenta bitului setuid (suid) pe executabilul /usr/bin/passwd?

Pentru a executa passwd user-ul nu are nevoie de privilegii de root (ex: vrea să își schimbe parola). Cu toate astea, procesul are nevoie de privilegii pentru a modifica fișierul /etc/passwd. Aici intervine bitul setuid care setează efectiv user id-ul la owner-ul executabilului - root.

7. (10 puncte) Într-un sistem rulează la un moment dat 100 de procese. Câte tabele de pagini sunt alocate? Justificați.

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 inconvenience if those frames had to be contiguous (i.e., physically next to each other). Using virtual-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) În urma a două apeluri accept() în codul unui server sunt creati doi socketi care au aceeași adresă IP și port. Cum diferențiază sistemul de operare socketul căruia îi 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 ambi socketi celelalte doua vor putea identifica peer-ul asociat (port sursa, ip sursa).

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

10. (10 puncte) Apelul lseek() actualizează cursorul de fisier. De ce această actualizare nu produce nici o modificare a inode-ului fisierului?

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](#), 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](#). 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) Aveti la dispozitie un sistem cu mai multe core-uri și vreti să dezvoltati o bibliotecă de video transcoding (CPU-bound) pentru stream-uri video dintr-un fisier. Presupunem că aveți detaliile unui algoritm de transcoding paralel. Acest algoritm permite efectuarea operatiei de transcoding separat pe blocuri diferite din stream (transcode_block()). Este însă nevoie, la finalul trascodingului unui bloc de o operatie de unificare la final pentru două blocuri adiacente (merge_adjacent_stream_blocks()); în această parte de unificare se „lipesc”, respectiv, părțile de început și sfârșit ale celor două blocuri. Blocurile sunt citite și scrise dintr-un/într-un fisier. Dimensiunea blocului este prestabilită.

Care vor fi principiile de proiectare a bibliotecii? Veti folosi thread-uri sau procese? Câte? Ce mecanisme de comunicare și sincronizare veti folosi în cadrul bibliotecii? Care sunt factorii de overhead din implementare?

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimenei să copieze la această lucrare.

Nume și grupă:

Semnătură:.....

5

4

Sisteme de Operare

3 iunie 2014

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Precizați două entități diferite pe care le poate referi un descriptor de fisier în Linux.

Entități diferite ce le poate referi: standard input/output/error, fisier, pipe, resursă 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) În urma unui buffer overflow adresa de return este suprascrișă. Către ce zonă poate poarta adresa suprascrișă pentru a genera un atac, în cazul în care sistemul are DEP (Data Execution Prevention)? Alegeti dintr-pe text, stack, data. Justificati.

3. (7 puncte) De ce este utilă paginarea ierarhică?

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 – 2^{64} addressable bytes / 2^{12} bytes per page = 2^{52} 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 = 2^{32} bytes.

2^{32} bytes of memory/ 2^{12} bytes per page = 2^{20} 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 = 2^{52} page table entries * 4 bytes = 2^{54} 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 52/10ceiled 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 diferă apelul fork() din Linux față de apelul CreateProcess() din Windows?

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) Când folosim mutex-uri în loc de operatii atomice pentru asigurarea accesului serial la date?

6. (10 puncte) Fie instructiunea:

a = b;

În ce situatie instructiunea generează două page fault-uri fără a conduce la terminarea procesului curent?

DEFINIEȚI:
A page fault is a trap to the hardware 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](#), 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

8

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) În ce situație folosiți apeluri de tip read/write în loc de maparea fișierului în 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) În cadrul unei conexiuni TCP, transmîtătorul realizează apelul:

```
send(s, send_buffer, 5000, O); /* send 5000 bytes */  
iar receptorul realizează apelul
```

```
recv(s, recv_buffer, 7000, O); /* receive 7000 bytes */  
Câtă octetă va primi receptorul (la întoarcerea din apelul recv)?
```

Receptorul poate primi oricărui octet între 1 și 5000, presupunând că niciun alt send nu s-a realizat sau se va realiza din partea transmîtătorului. În funcție de congesia rețelei, a încarcării bufferelor din kernel și cele de pe placă de rețea aceasta valoare variază între 1 și 5000. Dacă transmîtătorul include conexiunea înainte ca datele să fie transmise, apelul recv se va întoarce cu -1.

9. (10 punct) Un thread folosește malloc pentru a aloca memorie. Precizati un set de pași (în pseudocod) în care alt thread al aceluiași proces accesează zona de memorie alocată 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](#) and [this](#).
- 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 fisiere ext2 poate folosi fisiere cu dimensiune de până la 16GB. Ce limitează dimensiunea maximă a fișierelor?

There are various limits imposed by the on-disk layout of ext2. Other limits are imposed by the current

9

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 să implementăm un proxy server pentru conexiuni web. Un proxy server servește cereri din cache-ul local dacă paginile web se găsesc în cache; altfel face cereri către serverul web destinație, obține pagina și apoi o cache-uește. Facem următoarele presupuneri:

- Paginile cerute sunt, în mare parte, de dimensiuni mici (ordinul kiloobțetelor).
- Paginile se modifică greu; nu este nevoie să vă gândiți la expirarea paginilor în cache.

- Se poate folosi pentru caching atât memorie cât și spațiu pe disc, ambele limitate.
- Există un număr mare de cereri pe secundă pe care le primește proxy serverul.

Ce tehnologii veți folosi în proiectarea proxy serverului? (operații asincrone, multiplexare, multithreading, multiproces, etc.). Justificați alegerea.

Ce politică de înlocuire a paginilor în cache veți folosi?

Ce pagini veți plasa în cache-ul de memorie și ce pagini veți plasa în cache-ul de pe disc?

Cum veți asigura accesul sincronizat/consecvent/coherent la datele din cache?

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă:

Semnătură:

Sisteme de Operare

5 iunie 2014

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Când 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 processors.

2. (7 puncte) Ce conțin blocurile de date aferente unui director?

3. (7 puncte) De ce considerăm tastatura un dispozitiv de tip caracter a mai fost

4. (7 puncte) De ce este timpul de creare a unui thread al aceluiași proces mai mic decât 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) Câte tabele de descriptori de fișiere 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 legătură între planificatorul de procese și sistemul de întreruperi? Sistemul de întrerupere furnizează întreruperea de ceas, ce determină planificatorul sa ia decizii privind procesul ce va rula în următoarea cantă de timp.

7. (10 puncte) În cadrul unei conexiuni TCP un client trimite către un server mesajă într-o buclă:

```
while (1) {  
    send(s, buffer, 8192, 0); /* send 8192 bytes */  
}
```

La un moment dat, apelul send se blochează (pentru o perioadă de timp). Care este o cauză posibilă pentru această blocare?

Toate bufferurile între transmisor și destinatar s-au umplut ori rețea este congestiонată. Prin buffer ma refer la atat la bufferurile din kernel cat si la cele de pe placile de retea.

8. (10 puncte) Descrieți o situație în care un buffer overflow pe un array aflat în 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

12

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); }  
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
```

13

assembler dump. 3a. Overflowing the program # ./blah
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user inputxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
./blahxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Segmentation fault (core dumped) # gdb
blah core (gdb) info registers eax: 0x24 36 ecx: 0x804852f 134513967 edx:
0x1 1 ebx: 0x11a3c8 1156040 esp: 0xbfffffdb8 -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 puncte) Fie sevența de instrucțiuni de mai jos:

```
printf("%d", *a);  
*a = 42;  
unde a este un pointer la un întreg (int *). Dati exemplu de situație în care prima instrucție  
(printf) NU cauzează page fault, dar a doua (*a = 42) cauzează page fault.
```

10. (10 puncte) Un set de thread-uri lucrează cu o structură de tip listă dublu înălțuită. Unele thread-uri modifică lista (adaugă, sterg elemente), altele doar parcurg lista. De ce trebuie asigurat accesul exclusiv la listă pentru ambele tipuri de thread-uri, nu doar pentru cele care modifică lista?

11. (15 puncte) Dorim să implementăm o bibliotecă de tip engine de baze de date. Această bibliotecă va oferi un API de adăugare, stergere, inserare, modificare elemente în baza de date și va realiza și stocarea fiecărei baze de date într-un fișier pe disc. Un program care va folosi biblioteca va putea să stocheze informații într-o bază de date dintr-un fișier pe disc într-un format intern. Biblioteca trebuie să fie thread safe. Trebuie ca operațiile executate în threaduri diferite ale procesului să mențină datele coerente.

Definiti schematic API-ul pe care îl va expune biblioteca: structuri de date și funcționalități expuse ca interfață pentru programul ce va folosi biblioteca. Gânditi-vă doar la interfață expusă nu la internele implementării.

Cum vă asigurați că în cadrul implementării bibliotecii, în mod eficient, parteau de thread safety?

Cum propuneți să asigurați o viteză bună de lucru cu fișierul de bază de date și în același timp să oferăți o asigurare că mai bună că datele ajung pe disc?

Cum vă implementați parteau de tranzacție? Adică un set de operații să fie executate atomic în cadrul bibliotecii.

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă:

Semnatură:.....

Sisteme de Operare

8 iunie 2014

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Un apel mmap() rezervă 16 pagini de memorie virtuală. Câte pagini de memorie fizică alocă apelul?

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

3. (7 puncte) Cu ce diferă un spinlock de un mutex?

Mutex Vs Spinlock

Criteria	Mutex	Spinlock
Mechanism	Test for lock.	Test for lock.
	If available, use the resource	If available, use the resource.
	If not, go to wait queue	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.	Used when process should not be put to sleep like interrupt service routines.
	Use when there will be considerable time before process gets the lock.	Use when lock will be granted in reasonably short time.
	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

16

17

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 exemplu de apel de sistem care poate conduce la o schimbare de context între procese. Explicați inclusiv în ce condiții 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() și recv() au o sintaxă de forma:

```
accept(sockfd1, ...)  
recv(sockfd2, ...)
```

unde sockfd1 și sockfd2 sunt descriptori de socket. Cu ce diferă cei doi socketi?

sockfd1 - socket de tip listen asupra căruia se poate efectua doar operația de accept deoarece campurile port sursa și ip sursa nu sunt complete

sockfd2 - socket normal cu ajutorul căruia se realizează comunicația.

Caracterizat de atributele: ip + port sursa, ip + port destinație

6. (10 puncte) Într-un executabil sunt definite secțiunile text (codul programului), data (variabile globale) și rodata (variabile read-only). Care secțiuni vor fi partajate de două procese pornite separat din acest executabil?

7. (10 puncte) Ignorând câmpurile de tip timestamp din cadrul unui inode, dati un exemplu de apel de sistem de lucru cu fișiere (din forma open(), read(), write(), seek(), close(), chmod(), stat() etc.) care modifică un inode și altul care nu modifică un inode.

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

seek() - nu modifica inode-ul deoarece cursorul de fisier, ca și entitate, aparține de calea structură ce definește un fisier deschis și 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). Precizați cum se face un atac de tipul return-to-libc. Cum se obține adresa/adresele necesare?

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

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

DEP effectiveness (without ASLR)
In a previous blog post series we went into detail on what DEP is and how it works [part 1, part 2]. 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 /YNAMICBASE 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

18

19

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 *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 secvența de instrucțiuni de mai jos:

```
printf("%d\n", *a);
printf("%d\n", *(a+1));
unde a este un pointer la un întreg (int *). Dati exemplu de situație în care prima instrucțiune NU cauzează page fault, dar a doua cauzează page fault.
```

10. (10 punct) Care este un avantaj, respectiv un dezavantaj al folosirii suportului de *huge pages*? Adică pagini de 2MB (2 megabytes) în 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

20

21

Ce structuri interne veti folosi în cadrul alocatorului pentru gestiunea alocărilor?

Ce probleme posibile (de viteza/eficiență) pot apărea la nivelul alocatorului?

Cum veti asigura viteza bună de alocare/dezalocare?

Ce fel de aplicații/scenarii de test veti folosi pentru a testa alocatorul?

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă:

Semnatură:.....

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 să implementăm un alocator îmbunătățit de memorie. Alocatorul va expune funcțiile `malloc()`, `calloc()`, `realloc()` și `free()`, apeluri standard în lucrul cu memoria. În back end va folosi apelurile de sistem de tip `mmap()` sau `brk()` expuse de sistemul de operare pentru rezervarea de memorie virtuală. Cerințele alocatorului sunt viteză foarte bună și thread safety.

Sisteme de Operare

4 septembrie 2014

Timp de lucru: 60 de minute

Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Care este un avantaj al folosirii tabeliei de pagini ierarhice față de tabela de pagini simplă?

A mai fost inca o data.

2. (7 puncte) De ce un sistem este cu atât mai încărcat cu cât numărul de procese din cozile READY crește (adică mai multe procese în cozile READY înseamnă încărcare mai mare)?

In a real time system, admitting too many processes to the "ready" state may lead to oversaturation and **overcontention** for the systems resources, leading to an inability to meet process deadlines.

overcontention: **Bus contention**, in [computer design](#), is an undesirable state of the [bus](#) in which more than one device on the bus attempts to place values on the bus at the same time.

3. (7 puncte) În ce situație o operatie de tip `lock()` pe un mutex blochează thread-ul curent și în ce situație nu îl blochează?

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 [recursive mutex](#). If this mutex is a [non-recursive mutex](#), this function will **dead-lock** when the mutex is locked recursively.

4. (7 puncte) De ce au dispozitivele de tip bloc nevoie de operații mai rapide (cu throughput mai mare) decât dispozitivele de tip caracter?

5. (7 puncte) Cu ce diferă un apel de bibliotecă de un apel de sistem?

System calls are operating system functions, like on UNIX, the [malloc\(\)](#) function is built on top of the [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) În ce situație se poate rula cod pe stivă, chiar în 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*; 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 fisiere se poate folosi opțiunea *noatime*. Opțiunea înseamnă că nu va fi actualizat câmpul atime (timestamp de acces) al unui inode în momentul accesării (citirii sau scrierii inode-ului). De ce este avantajoasă această opțiune în cadrul unui sistem de fisiere încărcat (cu accese dese la fisiere)?

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 sevența de instrucțiuni de mai jos:

```
*a = 42; /* first dereferencing */
sleep(5); /* sleep for 5 seconds */
*a = 42; /* second dereferencing */
unde a este un pointer la un întreg (int *). Dati exemplu de situație în care prima dereferențiere nu cauzează page fault, dar a doua dereferențiere cauzează page fault.
```

9. (10 punct) Un proces în Linux are, în mod obișnuit, tabela de descriptori de fisiere limitată la 1024 de intrări. De ce în cazul unui server TCP încărcat, care primește multe conexiuni, este nevoie de creșterea acestei limite?

10. (10 puncte) De ce este mai probabilă apariția unui *stack overflow* în cazul unui proces multi-threaded față de un proces single-threaded? (*stack overflow* = depășirea limitei stivei în cadrul spațiului 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 altel 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, aplicații diferite pot comunica unele cu celelalte. Există patru concepții importante: Publishers (cei care produc mesaje), Consumers (cei care consumă mesaje din cozi), Exchanges (cei care primesc mesajele în cozi de la Publishers și apoi le transmit către Consumers), Queues (cozi de mesaje, create de Consumers și care stochează mesajele). Framework-ul trebuie să asigure scalabilitate și performanță. În general, Consumers, Exchanges și Publishers se găsesc pe sisteme fizic diferenți; sunt conectați prin Internet/retea.

Definiti, la nivel de pseudocod, metodele framework-ului folosite de Publishers și Consumers.

Ce probleme de scalabilitate pot apărea la nivelul framework-ului? (adică de la un nivel în sus se vor rezimti probleme de performanță)

Ce soluții de rezolvare a problemelor de scalabilitate există? (atât la nivelul framework-ului, cât și la nivelul infrastructurii folosite în instalare)

Cum asigurați o performanță ridicată a framework-ului și pentru o utilizare simplă (fără probleme de scalabilitate)?

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă:

Semnătură:.....

Sisteme de Operare
 13 septembrie 2014
 Timp de lucru: 60 de minute
Notă: Toate răspunsurile trebuie justificate

1. (7 puncte) Câte tabele de pagini se găsesc la un moment dat într-un sistem de operare? Într-un sistem se găsesc la un moment dat atatea tabele de pagini către procese există.

2. (7 puncte) De ce overhead-ul unui apel de sistem (oricât de simplu ar fi apelul de sistem) este, în majoritatea cazurilor, semnificativ mai mare decât overhead-ul unui apel de bibliotecă? Presupun că vorbim de un apel de bibliotecă care nu face în 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) În ce situație o operatie de tip down() pe un semafor blochează thread-ul curent și în ce situație nu îl blochează?

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 stivă proprie? Thread-ul reprezintă unitatea minimală ce descrie un fir de execuție. Fara o stivă nu s-ar putea menține și memoră contextul unui thread atunci cand se face context switching.

5. (7 puncte) Un proces CPU intensive este pornit și rulează timp de 30 de minute. La încheierea sa, se observă că acesta a consumat doar 20 de secunde timp de procesor. Cum se explică acest lucru?

6. (10 puncte) În general, nu se pot crea hard link-uri la directoare. Cu toate acestea numărul de link-uri aferente unui director diferă între directoare diferite; putem observa acest lucru prin rularea comenzi stat pe diverse directoare: /, /home, /usr/lib. De ce diferă numărul de

link-uri între 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 două intrări defaul: . si .. Astfel, putem sa zicem ca numarul de link-uri diferă între directoare deoarece diferă numarul de subdirectoare continue.

7. (10 puncte) De ce codul dintr-un shellcode se încheie, în general, cu o instrucțiune pentru realizarea unui apel de sistem (de forma int 0x80)?

8. (10 puncte) Un utilizator rulează, în terminal, **de mai multe ori** cele două comenzi de mai jos (rulăm de mai multe ori ca să fie informatile citite cache-uite și să nu afecteze rezultatul):

find /usr > find.out
 find /usr

Utilizatorul observă că prima comandă (cu redirectare în fisier) durează semnificativ mai puțin decât a doua (fără redirectare, care afișează pe terminal). De ce?

9. (10 punct) Care este avantajul unui server web care folosește mai multe procese pentru servirea cererilor făcute de unul care folosește 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 its 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 ocupă, pe parcursul rulării, maxim 100KB de memorie RAM. Cum explicăti?

11. (15 puncte) Dorim să implementăm o retea peer-to-peer. Reteaua va pune în prea mai mult decât orice pe disponibilitatea continutului (*availability*). Fiecare peer va rezerva un spațiu dat pe hard disk-ul propriu pentru fișiere care nu sunt ale sale dar care ne propunem să fie disponibile.

Perioada de timp cât un peer este activ și întâimea să de bandă sunt factori în stabilirea nivelului de implicare (*involvement*) al acestui peer. Un peer implicat va putea descărca mai rapid date de la alți peeri; fiecare peer își controlează banda de upload și acordă mai mult peerilor implicați.

Reteaua este folosită pentru transferuri de fișiere mici (muzică, documentație, mici fișiere video). Transferul se realizează doar între un peer și alt peer.

Care vor fi primitivele protocolului de comunicare între peeri?

Cum ati reține, în cadrul retelei peer-to-peer, informațile despre implicarea unui peer (involvement), pentru a fi accesate de peeri? Motivați alegerea.

Ce facilități veți folosi pentru o performanță cât mai bună a aplicației specifică unui peer

(pentru transferul fișierelor pe retea, stocarea fișierelor)?

Ce facilități oferă rețea pentru a asigura disponibilitatea conținutului? Adică fiecare fișier să fie stocat în cât mai multe locuri.

În conformitate cu ghidul de etică al Departamentului de Calculatoare, declar că nu am copiat și nu voi copia la această lucrare. De asemenea, nu am ajutat și nu voi ajuta pe nimeni să copieze la această lucrare.

Nume și grupă: Semnătură:.....

FARA NEGRU

Carina
CristianM
Pj Rosu <3
Remus TODO
Vlaicu
Cosminel
Mares Orange
Tahase
Iuga Turcu'
Anda
Baronescu <|>
Fabi
Trump

Puteti sa adaugati si din alti ani, la final.
Oricine doreste acces, sa ceara.

8 iunie 2016

- Precizați un apel care modifica cursorul de fisier și un apel care modifica dimensiunea unui fisier.

Modifica dimensiunea: truncate(), write(), open() cu O_TRUNC, close()

Muta cursorul: lseek, read, write, open - pozitioneaza la inceput sa la sfarsit.

- Metrica pentru un nivel de interactivitate al proceselor pt un planificator.

Timpul de așteptare (waiting time) = timpul de așteptare al unui proces în coada READY a planificatorului; acesta trebuie să fie cat mai scurt pentru a avea un sistem interactiv/responsiv
Nr. de procese pe secundă
Cuantă de timp - mică -> interactivitate

- Fie sectiunea de cod de mai jos:

char arr[128];

...

arr[140] = '\0';

In ce situatie instructiunea de atribuire rezulta in eroare de acces la memoria (de tip segmentation fault) si in ce situatie nu?

Instructiunea de atribuire rezulta in eroare de acces la memoria (segfault) atunci cand adresa din memorie ce trebuie accesata este într-o pagină la care nu avem drepturi de scriere (sau nu a fost alocată - **cuvantul potrivit cred ca paginata**) și nu rezulta in segfault atunci cand adresa referită e în aceeași pagină cu vectorul alocat, la care avem drepturi (sau într-o altă pagină, eventual cea următoare, care să fie alocată și la care să avem drepturi).

4. Apelul malloc() aloca memorie, în vreme ce apelul calloc() o să zeroizeaza (umple spațiul alocat cu valori de zero). De ce în cazul apelului calloc() spațiul de memorie fizică/residenta (resident set size) a procesului crește, dar în cazul malloc() nu (sau foarte putin)?

Apelul malloc face doar rezervare de pagini în memoria virtuală, fără a face corespondență cu spațiul de memorie fizică (care se va întâmpla ulterior, la accesarea efectivă a paginilor virtuale rezervate).

COMPLETARE: ..

Calloc e la baza malloc() + memset, iar memset implica mai multe apeluri de sistem, deci overhead

5. Pe un sistem de operare dat, stiva unui thread este limitată la 8MB. Care este un avantaj și un dezavantaj al creșterii acestei limite (de exemplu la 32MB)?

Dezavantaj: Poti face mult mai putine threaduri, capăt serverul lui perju (sugiri)

Avantaj: Faci recursivitatea pana vrea neamul lui tanase si ii trece si lui tema la SD

6. (10 puncte) Un sistem pe 64 de biți are suport pentru DEP (Data Execution Prevention) și ASLR (Address Space Layout Randomization). Presupunând că identificăm într-un program o vulnerabilitate de tip buffer overflow, ce acțiuni urmarim pentru a putea obține un shell?

Dacă are DEP, sigur nu se poate injecta cod și apoi să îl executăm, prin urmare va trebui să gasim/cautăm locul în care aslr-ul a palasat biblioteca libc pentru a reuși să apelăm /bin/bash prin suprascrierea adresei de return dintr-o funcție care /bin/bash.

Alternative answer: Putem injecta cod chiar dacă are DEP dar doar cod read-only .

Dep te impiedică doar să execute cod read-write. Deci am putea injecta un cod compilat și să îl setăm cu mprotect ca readonly iar pentru a scapa de ASLR putem introduce la începutul acestui cod un bloc de no-opuri și suprascriind adresa de return de unde se găsește vulnerabilitatea cu adresa mijlocului blocului de no-opuri din codul nostru aslr-ul o să dea o adresa de mai sus sau mai jos cel mai probabil din acel bloc de no-opuri și programul nostru va începe execuția. C: PS putem chiar injecta cod într-o pagină protejată read only și după să îl dam drepturi cu mprotect

7. (10 puncte) Precizați un scenariu în care două fisiere (inode-uri) diferite ajung să refere același bloc de date.

Se întâmplă astăzi dacă inoduri sunt linkuri simbolice -> pot referi același bloc de date și, simbolice link sunt inoduri de sine statatoare

8. (10 puncte) Un buffer circular este un buffer pentru care operațiile ajunse la sfârșitul său, continuă de la început (daca ai ajuns la ultimul element al buffer-ului, continuă de la primul element). Buffer-ul are două capete (doi indecs): in și out. Un consumator citește de la out, iar un producător scrie la in. Precizat, în pseudocod o implementare sincronizată a funcțiilor consume() și produce(item) folosind un buffer circular.

Sem N = 1, M = 0 // cred(99%) ca N e initial dimensiunea bufferului circular, nu 1

in,out

Producător:

While true

Aux = produce()

P(N) // acquire

Buffer[in] = aux

In = in mod max + 1

V(M) // release

Consumator:

While true:

P(M)

Consumă

Out = out mod max + 1

V(N)

9. (10 puncte) Pentru un program/executabil putem folosi funcionalitatea de setuid pentru acțiuni privilegiate. Întrucât aceasta funcționalitate este de tip totul sau nimic (obținând înregul set de privilegii ale utilizatorului administrativ), se recomandă folosirea de capacitat. O capacitate este avantajoasă pentru că permite doar un anumit tip de acțiuni privilegiate. De ce nu are sens să existe o capacitate pentru acțiuni privilegiate cu sistemul de fisiere (adică scrierea și citirea oricărui fisier)?

Un astfel de privilegiu ar permite acces la fisiere precum /etc/shadow, /etc/passwd, iar scrierea și citirea acestor fisiere de către oricine ar putea avea consecințe nedorite în securitatea sistemului.

10. Load average este o metrică care este proporțională cu numărul de procese aflate în stare READY. De ce load average-ul este mai mare pentru un sistem cu mai multe procese de tip CPU intensive?

Spre deosebire de procesele I/O intensive, care desorei se vor afla în stareia WAITING, procesele CPU intensive nu au nevoie de interacțiuni cu utilizatorul și deci vor solicita mereu accesul la procesor, deci vor fi în READY dacă procesoarele sunt deja ocupate. Astfel, mai multe procese CPU intensive conduc la un load average mai mare.

a terminat de scris. În cazul în care, argumentul count nu este 0, iar scrierea se face cu succes, atât cursorul cat și dimensiunea se modifică.

Completare: dacă cursorul este pozitionat la începutul fisierului, de exemplu, sau în mijloc și scriem un număr x de caracter x < numărul de caractere ramase până la sfârșitul fisierului , write va **suprascrie** cele x număr de caractere existente de la în fisier și nu v-a modificat dimensiunea acestuia. Dacă write modifică dimensiunea fisierului doar atunci cand cursorul trece de SEEK_ENDUL initial (dinainte de scrierii).

3. (7 puncte) Precizați și justificați o metrică numerică pentru nivelul de productivitate al unui planificator de procese.

Turnaround time cat mai mic. Număr de context switch-uri cat mai mic.

Numărul de context switch-uri într-o unitate de timp. Cu cat sunt mai multe, cu atât nivelul de productivitate este mai mare (?!?????????????????!!??!?!?!!?!!?!!?!!?!!?!!?)
C:??????^

Cred ca voia sa zica de interactivitate, nu de productivitate

C:agree

4. (7 puncte) De ce operația de tipul seek() nu are sens pe dispozitive de tip caracter și pe socketi, ci doar pe dispozitive de tip bloc?

Pentru că pe dispozitive de tip caracter se primesc datele una câte una și nu ai pe ce sa faci seek, în schimb dacă ai un bloc de date, ai unde să faci seek

Pentru că pe dispozitivele de tip caracter datele vin și sunt citite/ scrise octet cu octet, ca într-o teavă. Nu putem anticipa date și ne putem plasa mai sus sau mai jos pe banda de date. În cazul dispozitivelor de tip bloc însă, datele se găsesc pe un spațiu de stocare pe care ne putem plimba/glisa; putem "căuta" date prin plasarea pe un sector/bloc al dispozitivului de stocare și atunci operația seek() are sens. Pe socket se primesc în pachetele datele și nu ai cum să faci seek la ce se află în capitolul celalalt

5. (7 puncte) De ce este de preferat să folosim spinlock-uri pentru regiuni critice și mutexuri pentru regiuni critice mari?

Spinlock-ul folosește busy-waiting și are operații de lock() și unlock() ieftine prin comparație cu mutex-ul. Operațiile de lock() și unlock() pe mutex sunt de obicei costisitoare. Întrucât pot ajunge să invoke planificatorul. Având operații rapide, spinlock-ul este potrivit pe secțiuni critice de mici dimensiuni în care nu se fac operații blocante; în aceste cazuri faptul că face busy-waiting nu contează aşa de mult pentru că va intra rapid în regiunea critică. Dacă am folosi un mutex pentru o regiune critică mică, atunci overhead-ul cauzat de operațiile pe mutex ar fi relativ semnificativ față de timpul scurt petrecut în regiunea critică, rezultând în ineficiență folosirii timpului pe procesor.

Să folosesc spinlock-uri atunci cand codul din regiunea critică se va executa foarte rapid și celelalte threaduri nu vor aștepta foarte mult să ia lock-ul în busy-waiting.

10 iunie 2016

1. (7 puncte) Fie următoarea secțiune de cod:

char a[128];

...

a[900] = '0';

Instructiunea de atribuire nu rezulta în exceptie de memorie (segmentation fault). Acest lucru se întâmplă, însă în momentul folosirii instrucțiunii a[901] = '0'.

Cum explică?

Instructiunea de atribuire rezulta în eroare de acces la memoria (segfault) atunci cand adresa din memorie ce trebuie accesată este într-o pagină la care nu avem drepturi de scriere (sau nu a fost alocată - **cuvantul potrivit cred ca paginata**) și nu rezulta in segfault atunci cand adresa referită e în aceeași pagină cu vectorul alocat, la care avem drepturi (sau într-o altă pagină, eventual cea următoare, care să fie alocată și la care să avem drepturi).

Adresa ce trebuie accesată nu se află într-o pagină cu drepturi de scriere (sau nu a fost alocată);

Adresa a[900] se află chiar la sfârșitul paginii cu drept de write, iar a[901] la începutul unei pagini fară drept de write.

2. (7 puncte) În ce situație un apel de tipul write() modifica atât cursorul de fisier cat și dimensiunea fisierului?

write(int fd, const void *buf, size_t count); - Write() scrie acolo unde este pozitionat cursorul, nr de octeti trimisi ca argument, iar cursorul ramane acolo unde

6. (10 puncte) Fisierile executabile contin o secțiune numita PLT (sau function stubs) prin care se intermediaza apelul funcțiilor de biblioteca. Atacurile de tipul ret-to-plt urmăresc să folosească PLT pentru apelul funcțiilor de biblioteca. De ce sunt de interes aceste atacuri pe sisteme care folosesc DEP (Data Execution Prevention) și ASLR (Address Space Layout Randomization)?

Sunt de interes deoarece prin intermediul aceleia secțiuni putem afla unde a plasat ASLR-ul bibliotecile lIBC, astfel putem realiza atacuri return-to-lIBC.

ret-to-plt este folosit pentru a trece peste DEP și ASLR. În loc să facă return la o funcție din lIBC, a cărei adresa este randomizată de ASLR, atacatorul face return la un plt a11i funcției, a cărui adresa nu este randomizată. Pentru ca function@PLT nu este randomizată, atacatorul nu mai trebuie să ghicească adresa de bază a lIBC, putând face simplu return la function@PLT pentru a invoca funcția function.

7. (10 puncte) Fisierul /etc/shadow conține rezumate hash (funcții de tip one-way) ale parolelor. De ce se urmărește ca funcțiile care calculează rezumatul hash al unei parole să fie costisitoare din punctul de vedere al timpului: o rulare de funcție (care calculează un hash pentru o parola) să dureze, să nu fie obținut rezultatul instant? Sa nu fie obținute parolele prin brute force (dar nu sunt sigura) (e bine, am gasit pe net "bcrypt") Altfel ai putea user implementa un brute force îei un string gol, pui oricare caracter (256 posibile) si compari hash-ul obtinut de tine cu cel pt care cauti parola. Si concatenezi caractere in continuare pana ai match. Deci incerci 256^n_caractere_parola

8. (10 puncte) Thrashing este o problema a unui mecanism de stocare în care componentele acestuia (pagini, intrari) sunt foarte des schimbată, dacă și operațiunile scăzute; în loc să fie folosite, acele componente sunt foarte des schimbate. Cum are loc TLB thrashing?

Cred ca: ca TLB thrashing este loc cand se realizeaza alocari alternative de blocuri de memorie mai mici cu blocuri de memorie mai mari, de EX: daca in tlb avem mapate pagini pentru date de dimensiune mica(tlb plin) folosite des si avem nevoie de a accesa un bloc de date mai mare , tlb-ul va mapa multitudinea de pagini asociate blocului mare de date si va scoate paginile asociata blocurilor mici iar cand v-a venit randul de acces al blocurilor mici se vor cauza tlb-misses.

Cred ca e mai simplu de atat: tlb are o dimensiune limitata(desigur nowadays e un monstru pe multe nivele). Daca procesul are multe pagini mapate peste ram, atunci se va umple tlb si vei pune info peste intrările vecنه..t

9. (10 puncte) Un kernel preemptiv este unul în care poate întrerupe un proces de pe un procesor atunci când îl expira cantă sau este un alt proces prioritar, iar procesul rulează în spațiul kernel (kernel space). Un kernel nepreemptiv poate "întrerupe procesul" în acelle condiții doar dacă rulează în spațiul utilizator (user space). Un kernel preemptiv este mai util pentru interactivitate sau pentru productivitate?

Deoarece procesele copil mostenesc fișele descriptorii deschise de procesul parinte prin apelul pipe(). Daca se doreste comunicarea intre procese ce nu sunt inrudite este nevoie de named pipes.

5. (7 puncte) De ce numarul de schimbări de context între procese este mai mare atunci când se folosesc un planificator de procese cu prioritati (fata de folosirea unui planificator de procese fără prioritati)?

Deoarece procesele IO bound au o prioritate mai mare decât cele CPU bound (foarte probabil sa faca un apel blocant) deci numarul de schimbări de context creste, astfel obtinându-se interactivitate mare.

6. (10 puncte) memcached este un sistem de caching al obiectelor în memorie folosit în special în conjuncție cu sisteme de baze de date. Pentru o performanță superioară se recomandă folosirea Huge Pages, adică pagini de 2MB în loc de 4KB (pe x86_64). De ce? Alocările și dezalocările afectează putină performanța, mare parte din alocări facându-se la începutul rularii iar dezalocările sunt rare.

Căutarea în TLB este mult mai rapidă. Paginile fiind mai mari și nevoie de mai puține intrări în TLB (ai destulă memorie RAM într-un server).

Cred ca ar mai trebui ceva aici.

7. (10 puncte) Un utilizator porneste Firefox dupa un reboot al sistemului. Dupa un timp inchide procesul Firefox si apoi il reporneste la scurt timp dupa. Observa ca a doua oara procesul a pornit semnificativ mai rapid decat prima oara. Cum explicati?

C: probabil anumite blocuri de date sunt ramase prin nivelele de cache L2/L3 sau ram, si nu mai este nevoie de aducerea lor de pe disk (foarte lenta, mai repede iti vine pizza de la dominos) . In Windows (nu stiu sigur de Linux) este mecanismul de prefetch care face posibil acest lucru

8. (10 puncte) Un read-write lock este un mecanism de sincronizare care protejează o regiune critică din cadrul thread-urilor: thread-uri de tip reader (care citesc date) sau threaduri de tip writer (care modifica date). Există două posibilități:

i. în regiunea critică se găsesc doar thread-uri de tip reader, oricât de multe;

ii. în regiunea critică se găsesc un singur thread de tip writer (și doar acesta, fără alte thread-uri).

Fără de ce, în general, implementările de read-write lock sunt de forma write-biased? Adică dacă în regiunea critică se găsesc thread-uri reader și se găsesc la regiunea critică thread-uri writer și thread-uri reader, threadurile writer nu sosită vor avea prioritate în fața thread-urilor reader nou sosită.

Dacă se găsesc la regiunea critică și threaduri writer și reader, prioritatea vor avea cele writer deoarece acestea vor actualiza informația ce va fi citită de threadurile reader. Dacă threadurile reader ar avea prioritate acestea ar citi date ce vor fi modificate de către date invalide.

Interactivitate (procesele nu trebuie să aștepte mult timp pentru a rula pe procesor, se tine cont de o cantitate de timp, de prioritate)

10. (10 puncte) a.txt și b.txt sunt două link-uri (nume) la același inode. Inode-ul are contorul de link-uri egal cu 2. Pe un sistem dat, după operația mv a.txt /boot/ contorul de link-uri ajunge la 1 pentru inode. Cum explică?

/boot se află pe alta partitură.

Poate cineva să mai dezvolesc? E valabil pe cazul în care a fost creat un link la fișier și în acest caz b fiind hard link nu poate referi ceva altă parte.

Până la urmă cred că pe ambele pot să le interpretezi ca fiind hard link-uri, pentru că de fapt doar pointea către același inode și nu contează, de fapt orice fișier și un hard link către un inode (chiar dacă și doar unul). Mai era o întrebare de genul, de ce un fișier mare se mută pe aceeași partitură instant iar pe alta partitură durează, tocmai pentru că de fapt pe aceeași partitură și doar un "hard link" și doar se mută link-ul, inode-ul ramane tot acolo. (E CINEVA CARE GREDE ALTEL???????????????????????????? POATE GRESEC EU)

16 iunie 2016

1. (7 puncte) Ce tip de apel de sistem nu are loc în cazul folosirii operatorului & (ampersand) în shell (operatorul de rulare în background)?

Apelul wait() nu mai are loc. (parantezele nu mai asteapă procesul copil)

2. (7 puncte) O funcție într-un program consumă un procent semnificativ din timpul de rulare și ne propune să paralelizăm implementarea acesteia. De ce nu are sens folosirea de thread-uri cu implementare user-level?

Threadurile user level nu au suport multi-core, fiind necesara o implementare cu kernel level threads.

3. (7 puncte) Un program are o vulnerabilitate de tip buffer overflow și folosește canary values (stack smashing protection). Cum putem totuși ataca programul pentru a-l altera fluxul de execuție?

Canary, DEP, ASLR

Facem un handler pentru semnalul de stack smashing

Dacă reușești să cumperi să gasesc valoarea Canary-ului și în momentul în care suprascrii folosindu-te de buffer overflow să începi cu valoarea gasită, am înțeles că a spus ceva de genul la curs, stie cineva să spună cum???????

4. (7 puncte) De ce pipe-urile anumite (create cu ajutorul apelului pipe()) pot fi folosite doar într-un proces înrudite?

Starvation ... Dacă priorizezi threadurile reader, este posibil ca threadul writer să nu între în caciulă. (se stie!!! :))

9. (10 puncte) De ce dimensiunea tabeliei FAT nu este afectată de numărul de fișiere aflat pe o partitură formatată FAT32?

Numele de entries în tabela FAT e stabilit de la input. (Nu stiu sigur)

10. (10 puncte) Un atacator identifică o vulnerabilitate într-un program și o exploatează obținând un shell pe un sistem la distanță; shell-ul nu este de utilizator privilegiat. Programul însă rulează într-un chroot jail prevenind atacatorul să distrugă sau să obțină informații, și relevante de pe sistem. Cum poate continua atacatorul atacul?

1. Open a file handle to the root of the jail
2. Create a sub directory in the jail and chroot yourself there (you are root, so you are allowed to chroot). You're now even deeper in the jail.
3. Change directory using the file handle to the root of the old jail. You're now outside of the chroot jail you created in the sub directory. You're free! Well, kind of. Actually all locations starting with '/' are still mapped to that sub directory. You don't want this.
4. cd ..; cd ..; ...until you reach the real root.
5. Chroot yourself in the real root. Now you're properly free.
6. (DE AICEA AM LUAT)

11. (25 puncte) O firmă proiectează să implementeze un sistem de publicare de imagini. Firma dorește să atragă clienți și să publiceze folosind ca diferențiator performanța sistemului său. Vi se cere să proiectați și să implementați o aplicație/sistem care să evaluateze performanța sistemului de publicare de imagini.

a. Ce metrii veți urmați pentru a evalua performanța sistemului? (5 puncte)

b. Ce scenarii de utilizare veți folosi pentru a măsura acele metri? Cum veți folosi aplicația voastră pentru testarea sistemului? (6 puncte)

c. Ce particularități de proiectare și implementare va avea sistemul vostru pentru realizarea scenariilor de mai sus? Cum urmariti testarea limitelor sistemului de publicare de imagini? (7 puncte)

d. Ce veți recomanda firmei proiectante să modifice (fine tuning) la nivelul sistemului de publicare de imagini pentru a putea determina ce valori sunt adecvate pentru o performanță ridicată? (7 puncte)

17 iunie 2016

1. (7 puncte) Explicati daca si cum poate fi exploataata o vulnerabilitate de tip buffer overflow intr-un program scris in Java.
[Intrebare lucrare curs CA]
De ce nu sunt posibile buffer overflows in Java?
Raspuns: In Java, toate accesurile la memorie sunt verificate sa fie "in bounds". C'est magnifique
Alte opinii?
2. (7 puncte) Ce efect va avea comanda rm a.txt ? intr-un sistem de fisiere Unix? (a.txt este un fisier, iar comanda se intorce cu succes)
C: Se va sterge fisierul de pe disc, se va reduce numarul de inoduri din folderul in care se gasea a.txt (DACA MAI EXISTA UN HARD-LINK CATRE ACELAJI INODE???????)

3. (7 puncte) Ce se intampla daca un proces shell foloseste doar apelul exec(), nu si fork(), in rularea unei comenzi?
C: intreaga imagine a procesului curent va fi suprascrisa cu imaginea executabilului de la calea data in exec. (nu stiu sigur ce se intampla cu stiva procesului vechi)
4. (7 puncte) De ce este recomandat ca procesul parinte sa execute un apel wait() sau waitpid() pentru fiecare dintre procesele copil?
C: pentru a afii daca procesul copil s-a terminat cu succes sau nu. Astfel procesul copil nu va ramane zombie si procesul parinte stie daca treaba procesul copil a fost executata bine si daca poate folosi date modificate de acel proces
Cred ca e recomandat si ca sa poata sa elibereze resursele alocate proceselor copil.

5. (7 puncte) Explicati de ce intr-un handler de semnal nu este recomandat sa folosim variabile globale sau functii ne-reentrantne.
C: pentru ca un handler de semnal poate fi apelat de mai multe threaduri la un moment-dat, deci trebuie sa fie o functie reentrant (sa se poata fi apelata de mai multe ori in paralel si ea sa scoata acelasi output)

6. (10 puncte) In procesul P1, descriptorul d are offset-ul 0 intr-un fisier cu continutul abc. Procesul P1 executa fork() si rezulta procesul P2. Procesul P2 citeste din d caracterul a. Procesul P1 doarme o secunda dupa fork(), dupa care citeste un octet din d. Ce caracter va citi procesul P1?
C: va citi caracterul b deoarece procesul P1 va avea acelasi cursor de fisier ca si procesul P2 (mostenit de la parinte) si in momentul in care P2 citeste un caracter, cursorul din fisier se va muta si el cu o pozitie

7. (10 puncte) Explicati cum este partajat TLB-ul (Translation Lookaside Buffer) intre procesele care ruleaza pe acelasi sistem de calcul.
Fiecare proces are propriul TLB. No sharing. Se face flush la fiecare context switch.

8. (10 puncte) Descriptorul s reprezinta un socket TCP conectat, iar apelul send(s,buf,1000,0) intorce valoarea 500. Care e numarul minim de octeti pe care i-a primit receptorul pana la intorcerea apelului send()
C: 0 deoarece send se intorce in momentul cand a pus datele in bufferul de send din kernel si, intorcea numarul de octeti scrisi in acel buffer. Insa este posibil ca mesajul sa nu fie inca trimis cu adevarat catre destinatie sau sa nu fi ajuns inca. In concluzie numarul minim de octeti pe care i-a primit receptorul este 0.

9. (10 puncte) O aplicatie transmitter foloseste urmatorul protocol peste TCP: fiecare mesaj incepe cu 4 octet, care reprezinta dimensiunea mesajului, urmat, i de mesajul efectiv (maxim 1000 octet, i). Dup'a fiecare mesaj, se aseteaza primirea unei confirmari de la receptor. Explicati, i problemele care vor aparea atunci c'and transmis, "atorul execut'a dou'a apeluri send() per mesaj (unul pentru dimensiune s, i altul pentru mesaj) "in loc s'a faca un singur apel send().

10. (10 puncte) Intr-un sistem cu paginare, tabela de pagini ajuta la translatarea din adrese virtuale in adrese fizice. Tabela de pagini, 'ins'a, este stocata tot in memorie. Cum asta MMU (Memory Management Unit) adresa fizica a unei pagini?
- C: Adresa din memoria RAM a tabeliei de pagini a procesului curent este data de un registru dedicat numit generic PTBR (Page Table Base Register). Acest registru este incarcat de sistemul de operare cu valoarea aferenta procesului curent si este interogat de MMU.

11. (25 puncte) Un programator a implementat un server care primesc cereri de la client, i cont, in and numele unui fisier dorit, s, i transmite fis, ierul folosind sendfile(). Fis, ierile sunt stocate pe un disc de tip SSD atasat serverului. Serverul foloseste un pool de thread-uri pentru a procesa aceste cereri. Programatorul crede c'ea un singur parametru va afecta performanta, a sistemului: numarul T de thread-uri din thread pool. Ajutat, i programatorul s'a m'asoare performanta, a sistemului s'a s, i s'a il optimizeze.
a. Ce metrica este cea mai potrivita pentru a asurarea performantei server-ului? (5 puncte)

- b. Explicati, i cum poate fi m'asurata a ceasta metrica la server. (5 puncte) Programatorul doresc, te s'a s, tie dac'a bottleneck-ul il reprezinta SSD-ul, CPU-ul sau placa de retea.
c. Explicati, i ce experimente poate rula pentru a afla acest bottleneck. Ce valoare trebuie s'a foloseasca pentru T dac'a exista N procese care 'in sistem? (8 puncte)
d. Argumentati, i de ce implementarea unui cache de fis, iere 'in memorie nu ar imbun'at at, i performanta server-ului. (7 puncte)

C: M-am plăcutis rauuuu si nu am chef sa invat la PM :((

30 August 2016

1. (7 puncte) 'In urma rul'arii comenzi stat pe un fis, ier obt, inem c'a dimensiunea acestuia doresc, te s'a s, tie dac'a bottleneck-ul il reprezinta SSD-ul, CPU-ul sau placa de retea.
C: Ce apare acolo la dimensiunea fisierului poate fi modificat printre functii (truncate) dar nu ocupa si spatiul real.
2. (7 puncte) Prioritate, iel statice nu pot fi schimbate pe parcursul rul'arii unui proces. Care este problema cu un planificator care foloseste doar prioritate, i statica?
- Se poate ajunge la starvation (procesele cu prioritate mica vor sta foarte mult timp in coada de asteptare)

3. (7 puncte) Thrashing-ul este un fenomen negativ al sistemului de memorie virtuala 'in care se fac schimbari continue 'intre memoria principală s, i disc (swap in s, i swap out). Descrieti, i un scenariu care duce la aparitia, i thrashing-ului.
Thrashing-ul poate fi cauzat de programe sau workload-uri care nu prezinta localitatea referentie (datele accesate frecvent nu se afla in aceeasi parte a memoriei). Astfel, daca intreg setul de date pe care se lucreaza nu poate incepea in memoria fizica, atunci se poate produce swapping constant (thrashing).

4. (7 puncte) De ce un proces intr-un sistem de operare modern nu va dispune niciodata de mai multe pagini fizice decat pagini virtuale?
C: deoarece sistemele moderne sunt majoritatea pe 64 de biti -> un spatiu foarte mare de adrese virtuale, mult mai mare decat ai putea avea rami.
5. (7 puncte) De ce putem afirma ca paginarea ierarhica este un compromis spatiu-temp (space time tradeoff)?
C: deoarece prin paginarea ierarhica se miscoreaza dimensiunea tabelelor de pagini, insa este ingreunata cautarea. (a doua parte not sure)

6. (10 puncte) De ce este indicat sa folosim capabilitati in loc de setuid pentru executabile care au nevoie de privilegii?

Din motive de securitate. E mai safe folosirea de capabilitati, decat schimbarea userului. Un exemplu e comanda "ping" care trebuie sa execute actiuni privilegiate(deschidere de socketuri) si care e executata folosind capabilitati

7. (10 puncte) De ce unele sisteme pe 32 de bit, si care folosesc memory mapped I/O pot avea maxim 3GB de memorie RAM (numit si 3GB barrier) in loc de maxim 4GB?
C: pentru ca in acel 1 G lipsa sunt mapate functii din kernel pentru a nu fi necesar un context-switch la fiecare apel de sistem

8. (10 puncte) De ce mecanismele de sincronizare sunt mai complexe si mult mai frecvent folosite 'in codul care ruleaza 'in kernel mode fat, 'a de codul care ruleaza 'in user mode?
C: wtf?

9. (10 puncte) 'In ce situat, ie apelul send(sockfd, buffer, 1000) intorce o valoare cuprinsa 'intre 0 s, i 1000 (excluzand 0 s, i 1000)?
C: 0 nu cred ca are sens, cat despre valoare intre 0 si 1000, apelul send va intorce numarul de octeti pusi in bufferul send_buffer din kernel, buffer din care urmeaza sa fie trimisi catre destinatie. Datele odata puse acolo sunt considerate ca si trimise (asigura TCP), deci poate sa intorce oricat. Legal de dimensiunea maxima nu stiu sigur.

10. (10 puncte) Pe un sistem dat un proces poate dispune de un numar maxim de thread-uri. Cum putem creste aceasta limitare?
Micșorând dimensiunea stivei.

11. (25 puncte) Vi se cere s'a proiectat, i s, i s'a implementat, i o aplicat, ie de reverse engineering s, i binary analysis pentru executabile, care s'a fie capabila at'at de analiza statica c'at s, i de analiza dinamica. Se presupune c'au avut, i acces la codul sursei s, i c'a doriti, i s'a afiat, i c'at mai rapid informat, ii legate de executabili s, i de modul s'a de functionare. Urm'ariti, i act, iuri precum dezasamblare, construire de graf de flux de control (control flow graph), investigarea apelurilor de sistem, de biblioteci, a copierii codului (coverage), zone de cod hot s, i zone cold (apelat des, respectiv rar).

- a. Proponet, i o arhitectura de principiu a aplicat, iei (ce componente va avea s, i cum se vor conecta 'intre ele). (5 puncte)
b. Ce functi, i ionalit'at, i trebuie s'a ofere sistemul de operare pentru a putea implementa cerintele aplicat, iei? (7 puncte)
c. Cum at, i implementeaza in cadrul aplicat, iei descoperirea zonelor de cod hot s, i cold? (6 puncte)
d. Cum at, i folosi aplicat, ia pentru descoperirea vulnerabilitat, ilor de memorie care ar putea conduce la execut, i cod arbitrar? (7 puncte)

5 septembrie 2016

1. (7 puncte) Cum putem preveni aparitia de atacuri de tipul fork bomb?
Limitam nr de procese ale unui user (comanda ulimit)

2. (7 puncte) De ce o vulnerabilitate la nivelul nucleului sistemului de operare este mult mai periculoasa decat una la nivelul unei aplicatii in user space?

C: Deoarece nucleul sistemului de operare ruleaza in acces privilegiat, deci poate executa comenzi ce necesita drept de root. Pe cand o vulnerabilitate a unei aplicatii din user space nu este asa de periculoasa deoarece nu poate executa comenzi ca si root, neruland in mod privilegat.

3. (7 puncte) Are sens ca un sistem pe 32 de bit, i (cu 4GB spat, iu virtual de adrese pentru un proces) s'a ai'b' mai mult de 4GB de memorie fizic'a (RAM)? Justificati, i.

Magistrala de adrese procesor-ram are tot 32 de biti, nu poti adresa mai mult de 4gb de ram. Deci nu are sens sa ai mai mult de 4gb de ram.

4. (7 puncte) De ce, 'in general, nu are sens operat, ia lseek() pe dispozitive de tip caracter?

C: deoarece lseek poate inainta cursorul de citire/scriere intr-un bloc de date, iar in cazul dispozitivelor de tip caracter, datele sunt primite secentinal, nu in blocuri, deci nu am avea pe ce sa face seek

16 iunie 2016 - intrebarea 4

5. (7 puncte) De ce este preferat un sistem de fisiere FAT32 in fata unui sistem de fisiere NTFS pentru sisteme de fisiere pentru dispozitive mici (de forma USB flash drive)?

In principal pentru compatibilitatea intre platforme, iar fiind vorba si de dispozitive mici nu e o problema asa de mare limitarea de 4Gb a formatului.

6. (10 puncte) De ce este avantajos ca, in momentul planificarii unei noi entitati, planificatorul sa aleaga un thread din acelasi proces cu al thread-ului care a rulat anterior?

Overhead mai mic de context switch?

Nu se mai face tlb flush, inlocuirea tabeliei de pagini.

7. (10 puncte) In ce situatie operatia send() la transmit_ator s, i operat, ia recv() la receptor vor fi simultan blocate 'in cadrul unei conexiuni TCP?

C: pot fi simultan blocate daca pachetele se pierd undeva pe retea, astfel recv va ramane blocat fiindca e blocand, iar send se va bloca deoarece se va umple bufferul de send din kernel. TCP obliga SO sa pastreze cadrele pana se primeste ACK pentru ele.

Receptorul are bufferul de receive gol iar senderul are bufferul de send plin. S-a petrecut ceva pe traseu - la middle box ur

8. (10 puncte) Descrieti, i un scenariu prin care dou'a intr'ari din tabela de pagini a unui proces refer'a aceeas, i pagin'a fizic'a.

C: daca ele refera o biblioteca, acea biblioteca se va gasi intr-o singura pagina fizica(frame), iar cele 2 intrari din tabela de pagini virtuale a unui proces pot referi acea biblioteca (apeluri de functii maybe?)

9. (10 puncte) Folosind spinlock-uri sau mutex-uri pentru realizarea accesului exclusiv la resurse a comun' a 'intre thread-uri cu implementarea kernel level. De ce implementarea de spinlockuri poate fi realizat'a 'in user space (s, i nu necesita un apel de sistem) pe c'and implementarea mutex-urilor are nevoie de suport 'in kernel space?

C: cred ca deoarece spinlock-urile fac busy waiting si ele incearca acolo sa intre mereu, pe cand mutexurile realizeaza apelul wait, si asta inseamna ca vor fi trecute din running in coada waiting pana se primeste semnalul ce trebuie sa le deblocheze

10. (10 puncte) Dispozitivul /dev/mem permite accesul din user space la toata memoria fizica a sistemului. Scrierea la al N-lea octet din /dev/mem 'inseamna a scrierea la adresa N de memorie fizic'a. De ce doar utilizatorul privilegiat (root) are acces la acest dispozitiv?

C: cred ca are voie decat root pt ca din /dev/mem se pot scrie si chiar ce tin de kernel/system de operare si nu ar trebui sa poata fi scris/modifycate de oricine /dev/mem is a character device file that is an image of the main memory of the computer. It may be used, for example, to examine (and even patch) the system.
Daca orice taraamp; ar putea scrie in ram atunci nu si-ar mai avea sens separarea user space-kernel space si tot ce tine de securitatea soiului.

11. (25 puncte) Vi se cere s'a proiectat, i s, i s'a implementat, i un remote execution gateway, adic'a un sistem care preia cererile s, i le executa pe alte sisteme de execut, ie (de tip backend) 'in medii virtualizate. Gateway-ul primește task-uri, care conțin informatii, ii despre ce trebuie rulat (de exemplu, teste pentru teme), s, i o specificat, ie de mas, 'in a virtual'a. Apoi transmite task-ului s, i comanda 'mas, ina virtuala' pt unul dintre sistemele de execut, ie (backend). Rezultatul rul'arii sunt apoi trimise clientului care a comandat task-ului. Serverul trebuie sa raspund'a la un numar mare de cereri.

a. Realizat, i o diagrama bloc a componentelor software ale gateway-ului s, i a legatelor dintre ele. (5 puncte)

b. Stabili, i cum arat'a protocolul de comunicare 'intre clientul care trimite task-ului s, i gateway.

G'andit, i-v'a la trimitera task-ului s, i la primirea rezultatului. (7 puncte)

c. Stabili, i cum arat'a protocolul de comunicare 'intre gateway s, i sistemele de execut, ie. (6 puncte)

d. Precizat, i cum va arata sistemul de fis, iere la nivelul gateway-ului pentru a ment, ine informatii, ii despre task-urile 'in rulare s, i despre rezultatul obt, inut. Presupunet, i c'ea exist'a mai mult, i utilizatorii s, i trebuie sa fie p'astrat'a separat, i sigur'a 'intre submisiile utilizatorilor diferit, i. (7 puncte)

Puteti sa adaugati si din alti ani

11 iunie 2015

1. (7 puncte) Ce cauzeaza a treccerea din user mode 'in kernel mode?

Mecanismul de trecere din user mode in kernel mode este declansat de apelurile de sistem, revenirea in user mode facandu-se prin intorcerea din apelul de sistem.

Separatia kernel mode - user mode este importanta pentru ca sa nu fie posibil sa se execute (kernel mode) pentru operatii critice. Un mod privilegiat in care ruleaza sistemul de operare inseamna ca operatii critice (IPC, lucru cu I/O, lucru cu memoria) vor fi validate de sistemul de operare si un proces (aplicatie user space) nu poate face pagube sistemului. Pentru operatii privilegiate va fi necesara trecerea in kernel mode prin intermediul unui apel de sistem, si astfel invocarea sistemului de operare care actioneaza un gardian al operatiilor, garantand securitatea si integritatea sistemului.

2. (7 puncte) Ce cont, iei, 'in mod usual, o intrare 'in tabela de pagini a unui proces?

Mapare 'intre adresa virtuala a unei pagini si adresa unui frame fizic(RAM). Poate contine si bit de prezena, dirty bit(de modificare) si ID de proces.

3. (7 puncte) De ce este, 'in general, mai rapid a schimbarea de context 'intre dou'a thread-uri ale aceliasi, i proces decat schimbarea de context 'intre dou'a procese?

Nu se face TLB flush. Dezvoltare

Schimbarea de context intre 2 procese presupune inlocuirea tabeliei de pagini si flush la TLB => overhead. La schimbarea de context intre 2 threaduri nu este necesar acest lucru deoarece threadurile partajeaza tabela de pagini a procesului si au acelasi spatiu de adresa.

4. (7 puncte) Descrieti secenta de apeluri Linux prin care descriptorul 3 din tabela de descriptori de fisier a unui proces va referi iesirea standard, iar descriptorul 1 va referi fisierul a.txt:

```
dup2(1, 3);
int fd = open("a.txt", O_WRONLY);
dup2(fd, 1); // nu e dup2(fd, 1); ???
```

5. (7 puncte) Ce se intampina 'in cazul operat, iei up() pe un semafor?

Operatia up incrementeaza valoarea semaforului, daca nu exista nici un proces blocat pe acel semafor. Daca exista, valoarea semaforului ramane zero, dar unul din procesele (de exemplu, la intamplare) blocate vor fi deblocate. Operatiile up si down sunt, evident, atomice (ce este o operatie atomica?)

6. (10 puncte) Pe un sistem de fisiere un inode contine 7 pointeri directi catre blocuri de date si 2 pointeri cu indirecție simpla. Stiind ca un bloc are 4096 de octeti si ca un pointer ocupa 4 octeti, care este dimensiunea maxima a unui fisier pe acest sistem de fisiere?

Logic ar fi sa fie $7 \cdot 4096 + 2 \cdot 4$ (not sure).

7. (10 puncte) 'Intr-o zona a unui program exista urmatoarea secenta, i:

```
char arr[100]; [...]
/* instructiuni oarecare nu afecteaza array-ul arr */
arr[23] = 'a';
arr[24] = 'b';
In ce situat, i prima instruct, iune de atribuire NU cauzeaza a page fault, dar a doua instruct, iune de atribuire cauzeaza a page fault?
```

8. (10 puncte) Un programator a scris un program 'in limbajul C care foloseste apelul recv(s, buf, 2000) iar dimensiunea buf este de 1000 de octeti. Sistemul in discutie foloseste tehnici Address Space Layout Randomization (ASLR) si Data Execution Prevention (DEP). Poate fi acest bug exploata? Daca da, explicati cum.

9. (10 puncte) Dat, i exemplul de trei mecanisme ale sistemului de operare care permit restrictionarea daunelor pe care le poate provoca un proces controlat de un atacator.

1.Sandboxing (chroot jail), ii punte intr-un spatiu de unde nu poate sa iasă si nu poate face damage mult.

2.Nu se permite schimbarea ID-ului(cu setuid).

3....

10. (10 puncte) Explicati, i cum s, tie sistemul de operare care ar trebui sa fie destinat un segment TCP primit. Diferentiat, i 'intre segmentele cu bit-ul SYN activat sau dezactivat.

11. (25 puncte) Unui programator i se cere s'a implementeze un server de imagini care va deservi un numar mare de utilizatori. Fiecare utilizator poate 'incarcă sau descarcă imagini de dimensiuni mici (<1MB). Sistemul trebuie să poată să sustine simultan minim 1000 de utilizatori (upload si, i download), iar timpul maxim de upload sau download trebuie să fie 1s per fis, iei. Sistemul trebuie să retine, înă pentru fiecare utilizator numărul de octet, i incarcat, i sau descarcat, i, precum si, i numărul total de octet, i pentru tot, i utilizatori.

a. Ajutat, i programatorul să aleagă hardware-ul minim necesar pentru acest server. Aleget, i placă de retea de viteza, dimensionat, i memorie RAM, aleget, i numărul de core-uri/procese, HDD vs. SSD, etc. Explicati, i toate aleggerile facute. (7 puncte)

b. Specificati, i arhitectura serverului: dacă vor fi folosite procese sau thread-uri (c'ate?), apeluri blocante sau neblocante, etc. (8 puncte)

c. Pentru arhitectura aleasă, descrieti, i 'in pseudocod codul de tratare a unui client s, i modul 'in care statistice sunt actualizate. (10 puncte)

15 iunie 2015

1. (7 puncte) De ce procesele I/O intensive primesc, 'in general, o prioritate mai mare decat procesele CPU intensive?

Pentru ca stau foarte putin in starea running, efectuand o operatie blocanta si trecand in waiting. Astfel nu ocupa mult timp pe procesor, lasand loc altor procese.

2. (7 puncte) De ce consideram apelul chroot() o forma de sandboxing?

Multe sisteme folosesc chroot extensiv in etapa de dezvoltare si build pentru a detecta problemele generate de dependințele dintre diversele module software. Această metodă de build poartă uneori nume de sandboxing. Comanda chroot poate fi utilizată pentru a crea o copie virtuală a sistemului de operare. Aceasta poate fi utilizată pentru:

Testare și dezvoltare

Un mediu de test poate fi setat cu chroot, evitându-se astfel rularea testului pe un sistem aflat deja în producție.

Controlul dependințelor

chroot este o modalitate foarte bună de control al dependințelor dintre diferite module software aflate în dezvoltare.

Compatibilitate

Uneori este nevoie să rulăm software mai vechi care necesită versiuni mai vechi ale unor biblioteci. Un mediu chroot este ideal pentru rularea acestui software.

Recuperare

Sisteme care nu mai pot fi pornite de pe hard disc, se pot uneori porni rapid într-un mediu chroot plecând de la un Live CD sau alt mediu de pornire.

Separarea privilegiilor

Programe care în mod potențial constituie o problemă de securitate se pot rula într-un mediu chroot. Se aplică în general serverelor.

3. (7 puncte) Ce se întâmplă în cazul operat, iezi down() pe un semafor?

Marchează semaforul ca este ocupat și astfel nu poate fi accesat de alt proces.

Operația de down are un overhead ridicat, datorită schimbării de context.

4. (7 puncte) De ce schimbarea de context între două user level threads ale aceluiași proces este mai rapidă decât schimbarea de context între două kernel level threads ale aceluiași proces?

Threadurile user level nu necesită schimbarea spațiului de memorie și permisiunilor. Ele schimbă doar stackul specific.

5. (7 puncte) Ce conține o intrare în TLB (Translation Lookaside Buffer)?

TLB-ul menține mapările de pagini fizice și pagini virtuale. Conține un subset al tabeliei de pagini.(care conține pointeri către structuri de fisiere deschise)

6. (10 puncte) Pe un sistem cu suport DEP (Data Execution Prevention), un atacator urmărește să realizeze un apel mprotect(..., PROT_EXEC, ...) pe un set de pagini apartinând steivei. Ce urmărește atacatorul cu un astfel de apel? Cum îl va ajuta în continuarea atacului? Vrea să obțină drept de execuție asupra zonei de memorie

7. (10 puncte) Un programator masoara durata celor două instrucții, una de mai jos, realizate consecutiv: p = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); p[0] = 'a'; În urma măsurătorii observă că prima instrucție (mmap) durează 900 de nanoseconde iar a doua durează 3000 de nanoseconde (adică mai mult decât prima). Ce putem spune despre overhead-ul de timp cauzat de un apel de sistem făcând "de ovehead-ul de timp cauzat de page fault handler?

8. (10 puncte) Într-un program se realizează un apel: fd = open("/home/student/myfile", O_RDONLY); Apelul reusesc, este. Sătim că "/home/student/myfile" este un legătură simbolică (symbolic link). Câte inode-uri sunt citite în cadrul apelului open() de mai sus?

Ar trebui să fie citite toate inodeurile începând de la myfile până la inodul final.

9. (10 puncte) De ce, în general, pornirea unui proces durează mai mult la prima sa pornire decât la următoarele porniri?

10. (10 puncte) În ce situație, iezi at apelul send() pe transmitator, cînd s, i apelul recv() pe receptor ("în aceeași, i conexiune TCP) sunt simultan blocați?

11. (25 puncte) Se propune implementarea unui profiler de aplicat, îi multithreading cu obiectivul de a obține informații legate de profilul de performanță a acestor aplicații, ii.

a. Ce informații și metriți sunt utile de a fi furnizate de profiler dezvoltatorului aplicației, iezi? În ce formă sunt utile să fie furnizate acestea dezvoltatorului (numere, medii, grafice, tabele)? (7 puncte)

b. Ce facilitează, i trebuie să ofere sistemul de operare s, i hardware-ul pentru funcționalitatea profilerului? (6 puncte)

c. Cum va accesa profilerul facilitează, iile oferite de sistemul de operare s, i hardware (apeluri de sistem, memorie partajată, sisteme de fisieră, ierarhie, dispozitive virtuale etc.)? (5 puncte)

d. Ce facilitează, i metriți vor fi de interes pentru dezvoltatorul unei aplicații, ii de transcoding (conversie video)? Cum va fi folosit profilerul de către dezvoltator pentru a profila/optimiza aplicația, ia de transcoding? (7 puncte)

1 septembrie 2015

1. (7 puncte) De ce, în general, un prim apel open() reușește, însă în cadrul unui proces pe un sistem de operare Unix întoarce valoarea 3?

a. Propunet, i un model de specificare a profilului de sandboxing: ce va conține, iea fisierul de profil, cum va fi completat acest fisier, iea de profil de sandboxing? (6 puncte)

b. Ce funcții, ierarhie, i trebuie să ofere nucleul sistemului de operare pentru a putea folosi funcții, ierarhie de sandboxing pe baza profilului? Nucleul este cel care va gestiona aplicarea specifică, ilor din profil. (7 puncte)

c. Cum va fi asociat un profil de sandboxing unui proces dat? Din punctul de vedere al utilizatorului acesta trebuie să poarte aplicația s, i aceasta va folosi automat profilul de sandboxing definit de administrator pentru acea aplicație, iezi. (7 puncte)

d. Cum va fi protejată fisierul, ierarhie cu profilele de sandbox? Acestea trebuie să fie editabile/accesibile doar de administrator. (5 puncte)

10 septembrie 2015

1. (7 puncte) Explicăți, i de ce apelul printf nu este un mod robust de a face debugging unui program care primește semnalul SIGSEGV.

2. (7 puncte) Dată, i un exemplu în care planificatorul de procese Shortest Job First rezultă într-o alocare inechitabilă a resurselor procesorului.

3. (7 puncte) Numite, i un avantaj s, i un dezavantaj al folosirii memoriei virtuale.

4. (7 puncte) Un atacator descoperă o vulnerabilitate într-un program care-i permite să execute cod arbitrar. De ce sunt interesante pentru atacator apelurile read s, i write?

5. (7 puncte) Dată, i două avantaje ale implementării unui server de web cu mai multe procese, i de mai multe thread-uri.

6. (10 puncte) Explicăți, i de ce este necesară starea zombie pentru un proces în sistemele Unix.

7. (10 puncte) Un sistem Unix are spațiu, iu de adrese pe 32 bit, i s, i spațiu de swap de 40GB. Stiva unui thread are minim 4KB pe acest sistem. a. Estimăți, i numărul maxim de thread-uri ce pot fi create de un proces pe acest sistem. b. Estimăți, i numărul maxim de procese ce pot fi create pe acest sistem, presupunând că nu există limită la numărul de descripitori sau identificatori de proces.

8. (10 puncte) Un proces este blocat în apelul de sistem recvmsg. Un pachet este primit la placă de rețea. Descrieți, i secentă de pas, i pe care îl execută placa de rețea s, i sistemul de operare până când apelul recvmsg se va întoarce s, i aplicație, ia și, i va continua să se execute.

9. (10 puncte) Explicăți, i cum sistemul de operare căruia proces și este destinat un segment TCP primit. Diferențiat, i între segmentele cu bit-urile SYN activat sau dezactivat.

2. (7 puncte) Care este necesitatea spat, iului de swap într-un sistem de operare?

3. (7 puncte) Numite, i un avantaj s, i un dezavantaj al mecanismului de memorie virtuală.

4. (7 puncte) Care este un avantaj s, i un dezavantaj al folosirii chroot() în detrimentul virtualizării pentru izolare unui grup de procese?

5. (7 puncte) De ce este recomandat să folosim, cînd se poate, operat, ii atomice pentru sincronizare în loc să folosim alte mecanisme de sincronizare în cadrul unui proces multi-threaded?

6. (10 puncte) Ce limitează, în general, numărul maxim de thread-uri ce pot fi create în cadrul unui proces? Ofereți, i estimare de formulă (nu ceva specific, ci aproximativ) pentru calculul numărului maxim de thread-uri ce pot fi create pe un sistem dat.

7. (10 puncte) Un sistem de fisieră, ierarhie dispune de limite de număr maxim de fisieră, iere care pot fi create (inode-uri), numărul maxim de nume de fisieră, iere (dentry-uri), dimensiunea maximă a unui fisier, ier s, i spat, iul total ocupat de toate fisierele, ierale. Un utilizator creează un "într-o buclă" infinită hard link-uri. Care din limitele de mai sus va cauza oprirea creării de hard link-uri? Dar în cazul creației de symbolic link-uri?

8. (10 puncte) Un proces alocă un buffer de dimensiunea unei pagini: char buffer[PAGE_SIZE]; În ce situație, iezi operat, ia de mai jos rezultă în primirea unei exceptii, ii de tip Segmentation fault s, i în ce situație, iezi nu se întâmplă acest lucru? buffer[PAGE_SIZE+100] = 'a';

9. (10 puncte) Într-un sistem de operare un proces A este planificat des de procesor dar pentru intervale scurte de timp, în vreme ce un proces B este planificat mai rar, atunci cînd A nu rulează, dar timpul total de rulare pe procesor este mai mare decât în cazul procesului A. Ce poate, i spune despre cele două procese?

10. (10 puncte) Un program are o vulnerabilitate de tip buffer overflow pe un buffer de pe stivă. Sistemul dispune de suport DEP (Data Execution Prevention). Cum poate fi exploatață vulnerabilitatea?

11. (25 puncte) Se propune crearea unei soluții, ii de application sandboxing pe un sistem de operare dat. O astfel de soluție, iezi să asigure o cîntă mai bună izolare a aplicației, ierlor/proceselor sistemului în astfel de fel să fie diminuata pagubele cauzate de potent, iale vulnerabilității, i de securitate. O astfel de soluție, iezi trebuie să asigure accesul limitat la sistemul de fisieră, ierarhie, la componente de I/O, la primitive de tip IPC, la rețea etc. Soluție, iezi trebuie să permită definierea în cadrul unor fisieră, ierare a profilelor de sandboxing care apoi să fie folosite pentru sandboxing-ului aplicației, ii date. Un profil de sandboxing poate fi folosit de mai multe aplicații, ii.

10. (10 puncte) Un programator foloseste apelul de sistem write pentru a actualiza un fis, ier. După o perioadă de curent, programatorul descoperă ca unele informații au fost scrise cu write dar nu apar în fis, ier. Explicați, i cauza problemelor și, i dat, i o posibilă soluție, i.e.

11. (25 puncte) Vă se cere să implementați, i două programe, un sender și un receiver, care sunt capabile să transmită date între ele folosind două interfețe de rețea de 10Gb/s. Datele de transmisie sunt în memoria sender-ului, și se presupune că sunt infinite (e.g. dacă se ajunge la sfârșitul bufferului în care sunt stocate se va transmite de la începutul bufferului). Aplicat, ia receiver are nevoie să stocheze datele în memorie în aceeași ordine în care au fost citite de aplicat, ia receiver. Vă se cere să folosiți API-ul de sockets și, i protocolul TCP pentru a utiliza cele două interfețe disponibile la maxim (throughput total 20Gb/s):

- Detaliați, i protocolul pe care îl va folosi aplicat, ia pentru a împărtăși, și datele pe cele două căi și, i le pună în ordine la receiver. (7 puncte)
- Descrieți, i apelurile de sistem necesare pentru stabilirea conexiunii pe mai multe interfețe. (3 puncte)
- Descrieți, i în pseudocod implementarea pentru sender, atunci când conexiunea este deja deschisă. Avet, i în vedere folosirea corectă a operatiunii send. (5 puncte)
- Descrieți, i în pseudocod implementarea pentru receiver, atunci când conexiunea este deja deschisă. Avet, i în vedere folosirea corectă a operatiunii receive. (5 puncte)
- Analizați, i soluția propusă, discutând minim 2 posibile probleme de performanță care pot apărea în practică, și oferind soluții de remediere. (5 puncte)