16 mai 2012

1) Care din urm_atoarele apeluri de bibliotec_a (libc) invoc_a cel put¸in un apel de sistem: fopen, strstr, memcpy, malloc?

fopen() - DA, invoca apelul de sistem open(), operatie I/O necesita kernel
strstr() - NU, lucreaza cu bufferul curent(sirul). nu are treaba in kernel
memcpy() - NU, lucreaza doar cu spatiul de adresa al procesului curent, nu trece prin kernel
malloc() - NU, aloca memorie pur virtuala, fara apeluri de sistem(trecere prin kernel)

2) Fie urm_atoarea secvent¸_a de (pseudo)cod:

```
close(0);
close(1);
for (i = 2; i < 1000; i++)
tmpfd = dup2(i, i+1);
newfd = dup(tmpfd);
```

Ce valoare va avea newfd s¸i ce descriptor va referi? Argumentat¸i.

Daca taote apleurile sunt cu succes.
In for() toti descriptorii vor fi redirectionati catre 2(stderr).
Astfel tmpfd arata catre stderr.
dup() duplica descriptorul nefolosit cu numarul cel mai mic,in cazul nostru
0(stdin),(inchis mai sus in cod),deci newfd e redirectionat catre stdin.
Obs. By default un proces poate avea cel mult 1024 descriptori de fisier deschisi (linux).

3) In urma unui apel fork() pot rezulta ^_ntre X s¸i Y procese noi. Ce valori au X s¸i Y?

X = 0, daca fork() esueaza
Y = 1, faca fork() e cu succes intorcandu-se de 2 ori,
o data in procesul copil si a doua oara in procesul parinte

4) Fie a_rmat¸ia "Nu se recomand_a scrierea de cod care s_a foloseasc_a semafoare ^_n cadrul handler-ului de semnal." Precizat¸i s¸i justi_cat¸i valoarea de adev_ar a a_rmat¸iei.

Afirmatia e adevarata.Nu se recomanda folosirea semafoarelor-urilor intr-un handler de semnal, deoarece semnalele sunt asincrone(se pot produce practic oricand), ceea ce presupune ca s-ar putea produce cand un thread e blocat dupa un semafor/la acaparat.Avem deadlock,deoarece handlerul de semnal nu poate obtine semaforul-ul cat timp threadul nu se deblocheaza, dar el nu se poate debloca ,fiindca asteapta sa se termine handlerul de semnal(functia ce trateaza semnalul),ce l-a intrerupt in primul rand.

5. Fie următoarea secvenţă de (pseudo)cod:

```
int v[1024*1024];
int j = 0;
for(i = 0; i < 1024*1024; i++) {
    v[i] = i;
}

if (fork() == 0) {
    for(i = 0; i < 1024*1024; i++) {
        if (v[i] == 0)
            v[i]++;
    }
}

else{
    for(i = 0; i < 1024*1024; i++) {
        if (v[i] != 0)
            j++;
    }
}
```

Considerând că apelul `fork()` se întoarce cu succes, unul dintre cele două procese va executa secvența `for` mai rapid. Explicaţi care şi de ce.

Chiar daca ambele procese parcurg intreg vectorul v
procesul copil va executa for-ul mai rapid, deoarece
,exista o singura valoare nula in vectorul v, aceasta devenind nenula
dupa ce o moodifica procesul copil.Vom intra in if o singura data.Procesul
parinte pentru fiecare index din vector va incrementa j.Vom intra in if
de 1024*1024 ori.

5) Dat i exemplu de situat ie/scenariu ^_n care un swap in nu este precedat de swap out.

Sa zicem ca suntem in cazul in care un proces are nevoie de pagini fizice libere,
dar nu mai avem pagini libere, astfel trebuie sa inlocuim una ocupata.Pagina necesara
e adusa de pe disk(swap in), dar analizand dirty bit-ul pagini victime(ce trebuie inlocuite)
se observa ca ea nu a fost modificata,deci nu e necear sawp out-ul, pagina e suprascrisa pur
si simplu cu continutul adus de pe disk.

6) Fie a_rmat ia "Utilizarea mecanismului copy-on-write reduce mai mult overhead-ul cre_arii unui nou thread dec^at overhead-ul cre_arii unui nou proces." Precizat i s i justi_cat i valoarea de adev_ar a a_rmat iei.

Afirmatia e falsa, la crearea unu thread sau a unui proces copy-on-write reduce acelasi overhead,
deoarece e vorba de evitarea copierii unor pagini fizice in RAM, aceleasi pagini fizice fie ca avem
thread sau proces nou(ambii au acces la aceleasi pagini).

8) Fie un _s, ier a.txt, de dimensiune 100 MB, situat pe o partit, ie ext3. Ordonat, i cresc_ator
din punct de vedere al timpului urm_atoarele operat, ii:
a) copierea _s, ierului a.txt pe o alt_a partit, ie ext3;
b) s,
tergerea _s, ierului a.txt;
c) mutarea _s, ierului a.txt pe aceeas, i partit, ie;
d) mutarea _s, ierului a.txt pe o alt_a partit, ie ext3.
Motivat, i alegerea f_acut_a.

b) < c) < a) < d)
Stergerea fisierului insemna gasirea blocurilor necesare,marcarea blocurilor ca fiind sterse(overhead mic).
Mutarea fisierului presupune copierea(overhead mai mare) fisierului in noua locatie si steregerea din vechea locatie.
Daca procesul implica o ala partitie atunci se adauga overheadul lucrului cu acea aprtitie.

9) Pe un sistem se dores, te ca accesul la _s, iere de pe disc s_a _e c^at mai rapid. Care din cele dou_a tipuri de link-uri, hard sau sym, trebuie folosit ^_n acest caz s,i de ce?

Daca folosim linkuri hard vom avea dentry la acelasi inode-uri de fisier, pt sym (soft) vom avea copii suplimentare per fisier,deci cautarea va avea un overhead mai mare,deci e de dorit a folosi hard links.

10) ^In ce mod este afectat_a e_cient,a tehnicii ASLR (Address Space Layout Randomization) de urm_atoarele componente _zice ale sistemului: num_ar de procesoare, arhitectur_a pe 32/64 bit, i,memorie _zic_a, capacitate de stocare?

ASLR e o tehnica de securitate ce presupune reorganizarea aleatorie a diferitelor zone din spatiul de adrese al unui proces:
date, cod, stiva, heap astfel reducand diferite atacuri potentiale.Pt a creste securitatea e necesar a creste entropia,
ce impica cresterea sptiului de cautare,deci memoria virtuala care are la baza RAM-ul,deci memoria fizica ar
fi singura componenta ce ar influenta ASLR-ul.

24 mai 2012

1) Fie un sistem de calcul dual-core. De ce nu este posibil sa fie rulate simultan doua sisteme de operare diferite (de exemplu Windows si Linux) pe un astfel de sistem? Este vorba de rulare fara virtualizare, direct peste hardware.

Nu este posibil sa fie rulate 2 SO diferite pe acest sistem, fiindaca un SO este un lucru foarte complex incepand cu kernelul si terminand cu sistemul de fisiere, sunt foarte multe lucruri de care se tine cont cand ruleaza un SO: management memorie, device-uri, fisiere, procese, etc.Este obligatoriu ca cele 2 sisteme sa fie izolate, rulare lor simultana(daca ar fi posibil) nu ar ajuta deloc la protectia mediului fiecarui SO,sistemul s-ar bloca/deteriora.

2) Fie P1 si P2 PID-urile a doua procese care ruleaza concomitent. Fie FD1 si FD2 doi descriptori de fisier, folositi, respectiv de cele doua procese. Ce se poate spune despre valoarea de adevar a comparatiilor P1 == P2, respectiv FD1 == FD2?

P1 == P2, e falsa, deoarece nu pot exista 2 procese care ruleaza concomitent cu acelasi PID(procesele trebuie diferentiate!)
FD1 == FD2, e adevarata, e posibil ca 2 descriptori sa aiba aceeasi valaorea, se poate obtine prin redirectare(dup2()).

2) Intr-un sistem cu mai multe procese, procesul P1 este in starea RUNNING, iar procesul P2 este in starea READY. La un moment dat, procesului P1 ii expira cuanta si este trecut in starea READY. In ce situatie este posibil ca procesul P1 sa fie planificat pe processor inaintea procesului P2?

In acest caz ordinea de executie a proceselor e dicatata de prioritatea lor, daca
P1 are o prioritatea mai ridicata decat P2, atunci chiar daca P2 era deja in READY,
P1 trece in RUNNING inainte lui P2 datorita prioritatii.

4) Afirmatia de mai jos este adevarata. De ce?
Folosirea memoriei partajate intre doua procese, desi efcienta pentru comunicarea intre procese,produce un overhead mai mare decat comunicarea intre doua thread-uri ale aceluias i proces.

Afirmatia e adevarata, deoarece thread-urile aceluiasi proces folosesc spatiul sau de adresa pt a comunica(spatiul de comunicare deja exista), dar 2 procese au nevoie de ceva suplimentar si anume IPC-uri pt a putea comunica: message queue, shared memeorie, memory-mapped files ceea ce aduce un overhead in plus.

5) Dati un exemplu de situatie in care o schimbare de context nu este generata de un apel de sistem.

Exemple:
- se termina cunata pe procesor a unui proces
- se termina procesul
- un proces e preemptat de altul cu prioritate mai mare

6) Fie urm_atoarea secvent_a de cod:

```
char *a;
...
*a = '1';
*a = '2';
```

In ce situat_ie prima atribuire nu conduce la page fault dar a doua conduce?

In cazul in care info necesara primei atribuiri se alfa pe o pagina ce exista in RAM,iar pt a doua instructiune info se gaseste pe o pagina ce pe disk/swap, atunc iare loc situatia in cauza.

7) O implementare a unei aplicat_ii CPU intensive este portat_a pe un sistem multiprocesor cu implementare de thread-uri la nivelul spat_iului utilizator (user-level threads). De ce nu ofer_anici un avantaj aceast_a portare?

Aceasta portare nu ofera nici un avantaj ,deoarece user-level threads in ciuda faptului ca sunt mai rapide ca kernel-levelthreads au neajunsurile: nu ofera concurenta reala inte threaduri, daca se blocheaza un thread
se blocheza tot procesul, nu sunt folosite toate resurselel hardware, pot compromite sistemul ,daca numarul lor e prea mare, asta nu vinde deloc in ajutorul aplicatiei CPU-intensive.

8) De ce atat alocarea cu liste, cat si alocarea indexata la nivelul sistemului de fisiere sunt considerate robuste la efectele fragmentarii externe?

Ambele tipuri de alocari sunt robuste la efectele fragmentarii externe, deoarece blocurile care compun fisierul pot fi plasate aleator pe disc, deci acopera acele gaps din memoria libera de pe disk ce duc la fragmentarea externa.

9) ^In ce situat_ie este util_a emularea in detrimentul virtualizarii?

*Emulare totala - se emuleaza tot hardul sistemului de calul,adica se foloseste soft pt a virtualiza hardul fizic al altui sistem de calcul.Se poate emula si soft.
*Virtualizare totala - se creeaza niste bariere virtuale intre mai multe medii virtuale care ruleaza pe aceelasi mediu fizic.Chiar daca anumite deviceuri sunt virtualizate, in ultimul rand tot la hardul fizic se ajunge, de aceea virtulizarea e mai eficienta/rapida ca emularea.
Un caz in care s-ar alege emulare in loc de virtualizare ar fi testarea functionaltatilor unui device folosind un altul, astfel ar fi mai avantajos si mai ieftin de folosit soft decat hardware scump pt testare si intretinere.

10) Un proces care ruleaz_a ^_ntr-un chroot jail are acces limitat la sistemul de _s_iere. De ce putem spune c_a un astfel de proces are vulnerabilitate redus_a la atacuri de tipul buffer oveflow?

*chroot() creeaza chroot jail prin faptul ca odata apelat se modifica directorul root al sistemului de fisiere vazut de proces, asta influenteaza si restul apelurilor de sistem efectuate de procesul in cauza, deci va fi imposibil de accesat alte fisiere in afara arborelui root nou creat(prin chroot()).
*Atacurile de tip buffer-overflow modifica portiuni de langa buffer din saptiul de adresa al unui proces

adica daca avem intr-o functie un atac buffer-overflow, atactul va putea modifica: var locale,parametri sau chiar adresa de retur al functiei din stack frameul de pe stiva, facand ca functia sa returneze altundeva
in memorie(in continuare executand alt cod decat cel prevazut de utilizator sau generand erori).
In acest caz procesul are drepturi limitate, iar pt a produce buffer-overflow dezastruos ai nevoie de drepturi depline(modificare data in stiva, adresa retur, generare erori, seg-fault),deci vulnerabilitatea e redusa.

26 mai 2012
1)

```
char *a;
int pid, i = 0;
a = malloc(1024 * 4096);
/* TODO: start */
...
/* TODO: stop */
pid = fork();
switch (pid) {
case -1:
exit(EXIT FAILURE);
case 0:
/* page fault start */
for (i = 0; i < 1024 * 4096; i++)
a[i]++;
/* page fault stop */
break;
default:
break;
}
```

Completat, i, dac_a este posibil, sect,iunea marcat_a cu /* TODO: start */ s,
i /* TODO: stop */
astfel ^_nc^at s_a NU existe nici un page fault ^_n sect,iunea marcat_a cu /* page fault start */
s,
i /* page fault stop */. Motivat, i alegerea f_acut_a.

/* TODO: start */
mlockall(MCL_CURRENT | MCL_FUTURE);
/* TODO: stop */
Cu ajutorul acestui apel blocam paginarea, astfel incat paginile necesare vor fi aduse la cerere, si nu vor mai fi swaped out (pe disk),deci memoria in cazua va fi rezidenta.Acest lucru va sigura ca in zona marcata sa nu fie existe nici un page-fault.

2) O bibliotec_a de lucru cu liste simplu inlantuite foloses,te un mutex global pentru a asigura accesul exclusiv la elementele listei. Care este neajunsul acestei implement_ari? Este posibil_a o implementare care foloses,te c^ate un mutex pentru _ecare element? De ce?

Neajunsul acestei implementari e ca la un moment dat un singur element va putea fi accesat thread-safe(prin mutual exclusion) de un singur thread/proces, dar daca am avea mai multe threaduri/procese care doresc sa aceseze mai multe elemente in acelasi moment dat, atunci ar trebui cate un mutex pt fiecare element al listei care e ok, daca lista isi pastreaza o dimensiune relativ mica, altfel nu e rentabil(deoarece avem de adaugat cate un mutex pt fiecare element nou).

3) Atunci cand alegem dimensiunea memoriei swap ce aspecte ale hardware-ului trebuie luate in considerare si de ce: numarul de procesoare, arhitectura CPU (32/64 biti), dimensiunea spat iului de stocare, dimensiunea memoriei?

Swap-ul foloseste disk-ul pt swap in si swap out (de pagini), dar este influentat si de numarul total de pagini fizice, deci dimensiunea spatiului de stocare cat si dimensiunea memoriei RAM dicteaza dimensiunea spatiului de swap.

4) Pe un sistem cu arhitectura pe 64 de bit,i, cu un singur procesor, exista urmatoarele latente:
_ schimbarea registrelor { 10 ns
_ flush TLB { 20 ns
_ schimbarea tabelei de pagini { 100 ns
_ schimbarea tabelei de descriptori de fisier { 100 ns
Consider^and c_a nu exist_a alte latente, cat va dura o schimbare de context intre dou_a thread-uri ale aceluias, i proces si de ce?

Schimbarea de context in acest caz va fi de cel mult: 20 + 10 ns = 30ns.
Fiecare thread are setul sau proriu de registri ,deci registrii trebuie schimbati(10ns), avand in vedere ca se schimba thredurile intre ele avem un context switch, dar la context switch TLB face flush,deci mai avem 20ns overhead.Paginile procesului sunt partajate de threaduri si la fel si file descriptorii,deci nu trebuie modificate.

4) Un programator dores, te s_a foloseasc_a o funct, ie thread-safe pentru calculul factorialelor unor numere intr-un program multi-threaded. Care din implementarile de mai jos este thread-safe? Care este preferabila si de ce?

```
unsigned long fact1 (unsigned long n){
   unsigned long i, ret = 1;
   for(i = 1; i <= n; i++)
       ret = i * ret;
   return ret;
}
```

```
unsigned long fact2 (unsigned long n){
   if (n == 1)
       return n;
   else
       return n * fact2(n - 1);
}
```

Pentru ca o functie sa fie thread-safe e important ca e sa fie reentranta(de ex. fct. recursive).
Cum implementarea din stanga e iterativa nu e thread-safe si nici multi-threaded.
Implementarea din stanga e recursiva,deci reentranta si thread-safe, deoarece noua valoarea e intoarsa prin

intermediul stivei si se folosesc parametrii functiei.

5) Fie doua fisiere a si b, aate pe aceeas, i partit, ie ext3 si doua procese P1 si P2 care ruleaza simultan. P1 efectueaza scrieri in fisierul a, iar P2 efectueaza citiri din fisierul b. Stiind ca inode-ul fisierului a este diferit de inode-ul fisierului b, in ce situatie este necesara sincronizarea celor doua procese?

Este necesara sincronizarea celor doua procese daca ele ar comunica,adica P2 ii transmite datele lui P1 (printr-o coada de mesaje ce asigaura sincronizarea).

6) Fie a_rmat, ia "Pe orice partit,ie ext3 exist_a mai multe dentry-uri dec^at inode-uri." Precizat, si justi_cat, i valoarea de adev_ar a a_rmat, iei.

Afirmatia e falsa.Datorita structurii arborescente a sistemului de fisiere maparea dentry inode e many-to-one(mai multe dentry arata catre acelsi inode).Practic dentry sunt legaturile din arborele de fisiere, iar fisierele - nodurile, deci dentry sunt mai putine. *In orice director exista 2 dentry predefinite: .(dir curent), ..(dir parinte)

7) Dat, i un exemplu de situat, ie ^_n care un apel read asincron non-blocant aio_read(handle,buf, BUFSIZ); se va ^_ntoarce cu valoarea BUFSIZ (tot, i octet, i solicitati vor fi cititi pana la intoarcerea apelului).

Avem un proces care doreste sa citeasca date dintr-un fisier text abia deschis, iar buf are o dimensiune rezonabila.Fiind la inceputul fisierului apelul se va intoarce cu BUFSIZ.

8) Un utilizator dores, te s_a descarce 20 de _s, iere mari, de peste 10 GB. Desc_arcarea secvent, ial_a a _s, ierelor ^_i indic_a utilizatorului o vitez_a de desc_arcare constant_a, egal_a cu latimea de banda a placii de retea. ^In cazul init, ierii tuturor desc_arc_arilor simultan, suma vitezelor de descarcare reprezint_a mai put, in de 10% din valoarea anterioar_a. Consider^and c_a sistemul are resurse hardware su_ciente (num_ar de core-uri, cantitate de RAM, spat, iu pe disc, etc.) care este o cauz_a probabil_a pentru comportamentul de mai sus?

In acest caz (presupun) ca latimea de banda ar trebuie sa fie mare pt descarcarea a 20 de fisiere uriase!

9) Un administrator de sistem foloses, te un sistem gazd_a cu arhitectur_a pe 32 de bit, i care sust,ine o topologie de mas, ini virtuale. Dupa o perioad_a, topologia trebuie mutat_a pe un system cu arhitectura pe 64 de bit i. Care solut, ie de virtualizare necesit_a cel mai mic efort de portare dintre VMware Workstation si LXC? Ignorat, i efortul de portare a sistemului de operare gazda a hipervizorului si a VMM-ului.

In cazul de fata VMware ar fi cea mai buna solutie, deoarece VMware ofera virtualizare totala, astfel toate masinile virtuale sunt izolate,deci mai usor de protat, in schimb LXC ofera virtualizare

la nivel de SO, deci nu avem masini virtuale ,ci containere, mai greu de portat(izolarea nu e asa accentuata).

8 iunie 2012

1) Fie un sistem Unix f_ar_a virtualizare. Procesele P1 s,
i P2 sunt procese copil ale procesului
p_arinte P. Presupun^and c_a P1 s,
i P2 ruleaz_a simultan, este posibil ca pid(P1) == pid(P2)?
Dar dac_a P1 si P2 nu ruleaz_a simultan? Motivat, i.

Daca P1 si P2 ruleza simultan nu e posibil ca pid(P1) == pid(P2) (procesele ce ruleaza
trebuie sa poata fi identificate,deci unice in tabela proceselor)
Daca P1 si P2 nu ruleaza simultan atunci e posibil ca pid(p1) == pid(P2) (desi probabilitatea e scazuta).
,deoarece pidul procesului copil ce nu mai ruleaza pote fi refolosit.

2) Fie procesul P care execut_a urm_atoarea secvent,_a de cod:

```
#include <unistd.h>
int main ()
{
while ( fork() )
;
return 0;
}
```

Stiind c_a sistemul de operare are o limit_a de 20000 de procese simultane, nu exist_a niciun alt
proces care s_a ruleze pe sistem, iar latent,a de creare a unui proces nou este de 1ms, de ce NU
se va bloca niciodat_a sistemul? (nu se va atinge limita maxim_a de procese din sistem)

Din datele prezentate reiese sistemul nu se blocheaza(nu se ajunge la limita maxima), deoarece sistemul reuseste sa termine un anumit numar de procese, mecanism ce e echilibrat cu numarul de procese ce se creeaza.

3) Fie urm_atoarea secvent,_a de cod:

```
int seek;
/* TODO: start */
...
/* TODO: stop */
pid = fork();
switch (pid) {
case -1:
exit(EXIT FAILURE);
case 0:
seek = lseek(STDIN_FILENO, 0, SEEK_CUR);
break;
default:
break;
}
```

Apelul lseek din secvent,a de cod de mai sus ^_ntoarce pozit, ia curent_a ^_n cadrul _s, ierului referit
prin descriptorul STDIN_FILENO.
Completat, i, dac_a este posibil, sect,iunea marcat_a cu /* TODO: start */ s,
i /* TODO: stop */
astfel ^_nc^at valoarea variabilei seek ^_n procesul copil s_a _e 42. Motivat, i alegerea f_acut_a.

```
/* TODO: start */
int i, fd;
fd = open("test.txt", "w", 0644);
for(i = 0;i < 42;i ++)
write(fd, "1", 1);
dup2(fd, 0);
/* TODO: stop */
```

Practic creez un fisier nou, scriu in el 42 de caractere,deci mut cursorul de fisier cu 42 de octeti de la inceputul sau.Redirectez STDIN la fisier, aastfel STDIn arata la fisier, iar lseek arata pozitia curenta a cursorului in fisier.

4) Prezentat, i un motiv pentru care zona de date a unui proces nu este, ^_n mod normal, partajata cu alte procese.

In mod normal zone de date a unui proces nu e partajata cu alte procese din considerente de securitate, datele pot fi usor corupte la un acces incorct, modificare gresita a lor. Datele sunt foarete importante, astfel se pastreaza integritatea lor.

5) Stiind c_a a este o variabil_a global_a neinit, ializat_a de tip ^_ntreg, dat, i un exemplu de situat, ie ^_n care instruct,iunea a=42 genereaz_a un TLB miss.

Fie ca adresa variabilei a se afla pe o anumita pagina care este indexata in TLB, cand trebuie sa se execute instructiunea pagina necesara se gaseste in tablea de pagini si nu in TLB.

6) Fie a_rmat, ia "Un proces P are acces exclusiv la toate paginile _zice (frame-urile) pe care le foloses, te." Precizat, i si justi_cat, i valoarea de adev_ar a a_rmat, iei.

Afirmatia e falsa.Procesul are acces la toate frameurile sale, dar nu exclusiv, el defapt nu stie ca detine frameuri, el stie doar de paginile virtuale(ce sunt mapate peste cele fizice).

7) Pot dou_a thread-uri din procese diferite s_a partajeze o pagin_a de memorie _zic_a? Pot dou_a thread-uri ale aceluias, i proces s_a dispun_a de pagini _zice proprii, inaccesibile celuilalt thread? De ce?

-Da, 2 threaduri din procese diferite pot sa partajeze o pagina fizica,daca se foloseste memoria partajata/memory mapped files atunci fiecare threaad va folosi anumite pagini din spatiul de adresa al procesului sau ce reprezinta zona partajata,deci poate fi si o pagina partajata(nu stiu probabilitatea).
-Da, 2 threaduri ale aceluiasi proces pot sa dispuna de pagini fizice proprii, inacesibile

celuilalt thread prin folosirea TLS/TSD (zona de memorie privata per thread), daca doresc sa foloseasca anumite variabile private.

8) Fie FT mult,imea funct,iilor care pot _ folosite ^_n implement_ari multi-threaded s, i FH mult,imea funct,iilor care pot _ folosite ^_n handler-e de semnal. Ce relat,ie exist_a ^_ntre cele dou_a mult,imi?

Semnalele sunt facute pt procese si nu invers, deci implicit pt threaduri.
FH se include in FT, adica toate functiile ce implementeaza signal handlers sunt folosite in multi-threading, dar in FT mai avem functii care nu tin de semnale.

9) Fie o partit,ie care cont,ine put,ine _s,iere de dimensiuni mari. Prezentat,i un avantaj s, i un dezavantaj al aloc_arii cu liste ^_n fat,a aloc_arii indexate, ^_n cazul acestei partitii.

A: fisierele fiind mici(in mare) timp bun la cautare
D: timp de acces ridicat pt ultimile blocuri

10) Un dezvoltator dores,te s_a monitorizeze performant,ele unui plani_cator de procese pentru Android. Ce tip de virtualizare trebuie s_a foloseasc_a, stiind c_a mas,ina gazd_a este un system Intel cu arhitectur_a pe 32 de bit,i? De ce?

Ar trebui sa foloseasca o virtualizare totala(masina virtuala), pt a permite anliza kernelului virtual, ce contine planificatorul de procese.Daca ar folosi o vittualizare la nivel de SO ar folosi kernelul host,deci nu ar analiza ce doreste cu adevarat.
S-ar putea utiliza si emularea(mai inceata).

30 mai 2010 - 2011

1) In urma unui apel fork, at^at procesul copil c^at si procesul p_arinte apeleaz_a o funct,ie non-reentranta. De ce nu reprezint_a acest lucru o problem_a?

Acest lucru nu reprezinta o problema, deoarece functia fork() nu e
menita sa fie thread-safe, reentranta, ea se intoarce de 2 ori:
o data in procesul copil si o data in procesul parinte, asigurand info necesare
ce trebuie sa le cunosca procesul parinte cat si cel copil.

2) ^In ce situat,ie dou_a adrese virtuale ale aceluias,i proces acceseaz_a aceeasl adres_a _zic_a?

Acest lucru e posibil, atunci cand folosim threadurile aceluias proces.
Threadurile partajeaza implicit spatiul de adresa al procesului, deci
fiecare adresa virtuala din cele 2 (cate una per thread) va accesa acceasi adresa fizica.

3) ^In ce context instruct,iunea:
a = 0x42
declans,eaz_a o operat,ie de swap in?

Operatia se declanseaza atunci cand pagina care contine 0x42 se afla pe disk ,iar a e o variabila rezidenta in RAM.

4) Dat,i exemplu de situat,ie ^_n care un proces are deschis,i mai mult,i descriptori de _s,ier dec^at exist_a _s,iere (inode-uri) ^_n sistem.

In cazul in care avem mai multe deschideri cu open()
sau folosim dup2() pt duplicarea descriptorului de fisier pt acelasi fisier fizic(inode).

5) Fie urm_atorii timpi:
_ t1: timpul de schimbare ^_ntre dou_a procese;
_ t2: timpul de schimbare de context ^_ntre dou_a thread-uri cu implementare user (user level threads);
_ t3: timpul de schimbare de context ^_ntre dou_a thread-uri cu implementare kernel (kernel level threads).
Sortat,i cresc_ator (de la mic la mare) cei trei timpi s,i justi_cat,i sortarea.

t2 < t3 < t1
t2 < t3, deoarece user level threads nu trec prin kernel(overhead la context switch),
t3 < t1, deoarece threadurile partajeaza multe resurse,ceea ce nu se intampla pt procese (overhead la context switch)

6) Fie o operat,ie sincron_a non-blocant_a (read) s,i o operat,ie asincron_a (aio_read), apelate ca
mai jos (pseudocod):
write(handle, buf, BUFSIZ);
aio_write(handle buf, BUFSIZ);

C^at, i octet,i vor _ scris, i ^_n handle-ul dat, ^_n _ecare caz, dup_a ^_ncheierea _ec_arei operat,ii initiate de instruct, iunile de mai sus?

read() va scrie <= BUFSIZ octeti,deoarece proprietatea non-blocanta nu garanteaza
scrierea tuturor ocetilor
aio_read() va scrie <= BUFSIZ octeti, idem(buffer mic, sfarsit de fisier aproape, semnal).

6) De ce, ^_n cazul programului ping, prima operat, ie realizat_a ^_n funct, ia main este crearea unui socket raw?

In progrmul ping.c prima operatie in main e crearea unui socket raw, deoarece deschiderea socketului se face cu drepturi de superuser, dupa care se limiteaza drepturile procesului prin apelurile: getuid(), setuid()
din considerente de securitate.
*setuid() seteaza proceseului alt userid normal diferit de superuser id.
ping.c e in Cursul 12
http://www2.fiit.stuba.sk/~kosik/doc/sandboxed-ping.pdf

7) De ce o bun_a parte din ush-urile de TLB au loc imediat dup_a primirea unei intreruperi de ceas?

Cand TLB primeste o intrerupere de ceas este notificat ca are loc o schimbare de context
intre procese.In acest caz intrarile din TLB devin invalide, astfel pt a elimina
intrarile invalide el este flush-uit.

8) De ce, ^_n general, un apel mmap nu declans,eaz_a o operat, ie de ^_nlocuire de pagin_a, indifferent cat spat, iu se solicit_a s_a _e alocat?

Apelul mmap nu genereaza o operatie de inlocuire de pagina ...,deoarece
mecanismul de mapare e similar cu mecanismul de mapare a memoriei virtuale.
In caz de insuficienta de spatiu se foloseste acelasi mecanism de swap in/out.

10) Dou_a procese P1 si P2 comunica printr-un mecanism de cozi de mesaje unidirectionale.
Se folosesc dou_a astfel de cozi (M1 si M2) pentru comunicat, ie bidirect,ional_a. Cozile nu exista init ial.
Procesul P1 foloses te M1 pentru a trimite un mesaj c_atre P2, iar P2 foloses, te M2 pentru a trimite raspunsul. Dup_a aceasta, comunicat, ia se incheie si cozile sunt distruse.
Descriet, i secvent,a de operat, ii de forma create, open, read, write, close, destroy pe care trebuie sa le execute fiecare proces. Ce proces trebuie s_a porneasc_a primul?

Ambele procese pot sa proneasca primele,deci avem 2 cazuri:
Varianta P1 porneste primul
P1:
m1 = create("M1", O_WRONLY, 0644);
m2 = create("M2", O_RDONLY, 0644);
write(m1, buf, BUFSIZ);

read(m2, buf, BUFSIZ);
close(m2);
destroy(m1);
P2:
m2 = open("M2", O_WRONLY, 0644);
m1 = open("M1", O_RDONLY, 0644);
read(m1, buf, BUFSIZ);
write(m2, buf, BUFSIZ);
close(m1);
destroy(m2);
In varianta cand P2 porneste primul e vice-versa.
P2 creeaza cozile, P1 le deschide.

27 mai 2010-2011

1) O implementare de thread-uri ^_n kernel (kernel level threads) respectiv o implementare de
thread-uri ^_n spat iul utilizator (user level threads) execut_a urm_atoarea secvent _a (valid_a) de cod:

```
for (i = 0; i < 1024; i++)
write(fd, "a", 1);
```

In cadrul c_areia dintre cele dou_a implement_ari vor _ generate mai multe apeluri de sistem?

User Level Threads nu folosesc kernelul in comutarea de context, kernelul vede firele de executie din proces ca un sigur fir(vede doar KLT), deci in aceasta varianta vor fi folosite putine apeluri de sistem(daca e sa fie).In varianta ce foloseste Kernel Level Threads vor fi folosite apeluri de sistem (cel putin 1024 pt apelul write()).

2) De ce nu are sens operat ia de seek pe socketi?

Operatia de seek nu are rost pe socketi, deoarece socketul e un mediu de comunicare e un endpoint de comunicare in retea, nu are continut care poate fi parcurs!
Socketurile, pipe-urile nu pot fi mapate in memorie ,nu permit accesul secvential.

4) Un sistem foloses te pagini _zice (frame-uri) de 64KB. Precizat i un avantaj, respectiv dezavantaj al folosirii unei astfel de dimensiuni de pagin_a _zic_a (frame), ^_n locul unei pagini _zice (frame) de 4KB.

Avantaj:
- mai multe adrese per pagina
- translatare mem fizica - mem virtuala mai rapida
Dezavantaj:
- fragmentare memorie

5) Un sistem dat foloses te copy-on-write. Cu toate acestea se observ_a c_a timpul de creare a unui thread este semnicativ mai mic decat timpul de creare a unui proces. Care este principal cauz_a a acestei diferent e?

Timpul de crearea a unui thread este semnificativ mai mic decat
timpul de creare a unui proces, intrucat firele de executie ale unui proces
partajeaza multe resurse: spatiul de adrese, descriptori de fisier, pipe, socketi
Insa fiecare fir de executie are contextul sau: stiva + registri.
Copy-on-write nu influenteaza aceste diferente.

5)
Fie urm_atoarea secvent,_a de cod:

```
char *a;
a = mmap(NULL, 1024 * 4096, PROT READ j PROT WRITE, MAP ANONYMOUS j MAP PRIVATE, -1,
0);
/* TODO: start */
...
/* TODO: stop */
pid = fork();
switch (pid) f
case -1:
exit(EXIT FAILURE);
case 0:
int d = 0;
/* page fault start */
for (i = 0; i < 1024; i++)
d += a[i*4096];
/* page fault stop */
break;
default:
break;
}
```

Completat, i sect,iunea marcat_a cu /* TODO: start */ s,
i /* TODO: stop */ astfel incat sa nu existe nici un page fault ^_n sect,iunea marcat_a cu /* page fault start */ si /* page fault stop */.

/* page faul start */
mlockall(MCL_CURRENT | MCL_FUTURE);
/* page fault stop */
Daca e sa se genereze pagefaulturi se vor genera la mapare(mmap()),
apoi se va bloca paginarea folosind mlockall(),deci paginile deja existene
in RAM nu vor fi swapate, acest lucru va sigura faptul ca paginile necesare
vor exista la nevoie, si in zona marcata nu se va genera pagefault.

6) Se dores,te rezolvarea problemei _lozo_lor folosind doar monitoare. Care este num_arul minim de monitoare necesar pentru rezolvarea problemei?

Vom avea nevoie de 1 montor si 2 varibile de conditie(x si y), pt 5 filozofi.
,deoarece sincronizarea va fi efectuata de variabilele de conditie legate la monitor
(2 varibile pt 2 furculite folosite curent).
Pentru a evita situatia de dealock si starving procedam astfel:
Asociem fiecarei furculite cate o variabila de conditie.
Fiecare dintre primii 4 filozofi: semnaleaza var de conditie x acaparand furculita din stanga,
,la fel si cu y, acaparnd furculita din dreapta.Manaca.Apoi elibereaza cele 2

furcuilte facand wait pe varibilele x si y.Apoi gandeste. Ultimul filozof(al 5-lea)
procedeza invers, acapareaza furculita din dreapta si apoi cea din stanga.

7) Pe un sistem cu suport OpenVZ ruleaz_a 10 containere. Pe un alt sistem ruleaz_a 10 masini virtuale VMware. Toate mas¸inile virtuale/containerele ruleaz_a acelas¸i sistem de operare. In care dintre cele dou_a sisteme, plani_catorul de procese al sistemului de baz_a (host-ului) lucreaza cu mai multe procese?

OpenVZ - virtualizare la nivel de SO(fiecare utilizator are containerul sau virtual, dar imparte sistemul de oeprare, kernul, hardware-ul).
(procese lnu sunt mascate, coexista cu cele ale hostului).
VMware - virtualizare completa(virtualizare la nivel de hardware, kernel, shell, etc.)
(procesele sunt mascate sub cateva procese vmware)
In primul caz vom avem substantial mai multe procese.

8) Pe un sistem aat ^_n rulare, care dintre cozile READY, WAITING, RUNNING poate s_a nu cont¸in_a nici un proces?

Coada RUNNING contine procese,deoarece sistemul ruleaza.
De obicei apar mai multe procese noi decat reuseste CPU-ul sa execute imediat,deci coada READY ar avea
mereu procese.
Coada WAITING ar putea sa nu contina nici un proces,deoarece exista posibilitatea
ca sa avem mai multe procese CPU-bound si foarte putine I/O-bound, procesele care asteapta in coada WAITING
termina de asteptat si sunt scoase din coada WAITING trecute in READY.

9) De ce este considerat TransmitFile un apel zero-copy?

Apelul TransmitFile() este folosit pentru a eficientiza transmiterea de fisiere in retea. TransmitFile foloseste cache-ul sistemului de operare. Este o operatie zero-copy,fiindca nu necesita alocarea de buffere in user-space si diminueaza numarul de apeluri de sistem.

10) La o listare rezult_a urm_atorul output:
$ ls -lh /usr/
[...]
drwxr-xr-x 2 root root 12K May 20 10:08 sbin
drwxr-xr-x 6 root root 4.0K May 14 08:52 src
[...]
De ce directorul sbin/ are dimensiunea (12K) mai mare dec^at a directorulului src/ (4K)?

sbin e mai mare in dimensiune ca src, dar diferenta e datorata si hardlinkurilor(avem multe cai catre acest director
pt rularea diferitelor fisiere binare, in alte directoare decat sbin)

12 sept 2010-2011

1) ^Intr-un sistem de _s,iere:
_ a.txt reprezint_a numele unui _s,ier;
_ b.txt este un hardlink la acelas,i _s,ier;
_ c.txt este symlink la a.txt.
Se execut_a, ^_n trei programe diferite, secvent,ial, urm_atoarele operat,ii ^_n pseudocod:

```
/* P1 */
f1 = open_and_truncate("a.txt")
write(f1, "abc")
f2 = open("a.txt")
write("def")
/* P2 */
f1 = open_and_truncate("a.txt")
write(f1, "abc")
f2 = open("b.txt")
write("def")
/* P3 */
f1 = open_and_truncate("a.txt")
write(f1, "abc")
f2 = open("c.txt")
write("def")
```

Ce va cont,ine, la _nele _ec_arui program, _s,ierul a.txt?

P1: abc
P2: abc
P3: abc
In fiecare proces(P1, P2, P3) se deschide fisierul 'a.txt' se trucnhiaza si se scrie in el 'abc',deci abc si ramane scris in 'a.txt'.

2) In ce situat,ie ^_s,i poate un proces schimba PID-ul?

Un proces isi mentine PID-ul in decursul intregului ciclu sau de viata.
Deci fiecare PID nou e un alt proces! Un proces nu isi poate schimba PID-ul nici pe Linux nici pe Windows.

3) Care dintre apelurile de mai jos poate genera o schimbare de context ^_ntre dou_a procese?
Cum se ^_nt^ampl_a?
_ open
_ memcpy
_ lock
_ unlock

lock(&mutex) - incercarea de a obtine un mutex inaccesibil trece procesul in starea WAITING - DA
unlock(&mutex) - la iesirea din zona critica, procesele care asteptau la mutex vor trece din starea WAITING in starea READY - DA

4) In momentul ^_n care este mapat_a ^_n memorie, zonele unei biblioteci ocup_a, ^_n memoria _zic_a (RAM):
_ zona de cod (text): 10 pagini;
_ zona read-only (rodata): 1 pagin_a;
_ zona de date (data): 2 pagini.
Biblioteca este folosit_a pe post de bibliotec_a partajat_a de un num_ar de 100 de procese ^_n sistem.
C^at spat,iu ocup_a ^_n total, ^_n memoria RAM, zonele aferente bibliotecii?

Biblioteca pratajata odata incarcata in memorie poate fi folosita de mai multe procese odata (shared).Asta nu inseamna ca poate fi chiar partajata intre toate procesele,deoarece fiecare proces are propriul sau spatiu de adrese si doar pe acesta il vede.In cazul nostru biblioteca e formata din 3 zone dintre care una statica.Cred ca zona statica va fi chiar partajata intre procese, deci avem 1 pagina partaja la 100 de procese si 12 pagini per procese,deci in total zonele aferente ale bibliotecii partajate vor ocupa 1201 pagini in RAM.

5) Un sistem dispune de 2GB RAM. Imaginile proceselor care ruleaz_a ^_n acel moment cumuleaz_a 1.5GB, su_cient pentru a ^_nc_apea ^_n RAM. Se observ_a, ^_ns_a, c_a sistemul foloses,te si o parte din spat,iul de swap. Cum explicat,i?

Chiar daca imaginile proceselor cumuleaza sub limita maxima de memorie RAM, swap-ul ar putea fi totusi folosit din
considerente de paginare.Exista posibilititatea sa se genereze mai multe page faulturi,atunci trebuie sa fie aduse in RAM
paginile fizice necesare, daca in RAM nu mai este loc cele mai putin utilizate pagini fizice vor fi swapped out si inlocuite
cu cele necesare.

6) Un proces dispune de trei thread-uri. Unul dintre thread-uri apeleaz_a mmap() s,i stocheaz_a valoarea ^_ntoars_a de mmap ^_ntr-o zon_a privat_a (Thread Local Storage / Thread Speci_c Data).
Este aceast_a abordare suficienta pentru a proteja regiunea alocat_a cu mmap() de accesul celorlalte thread-uri?

Da, aceasta abordare este suficienta, chiar daca mmap() functioneaza dupa tehnici similare memoriei virtuale
TLS/TSD creeaza o zona privata invizibila celorlalte threaduri ale aceluias proces, TLS/TSD-ul
facand parte doar din contextul threadului care o creeaza.

7) Care dintre primitivele de sincronizare de mai jos foloses,te, ^_n cadrul implement_arii interne,
(cel put,in) o coad_a pentru gestiunea proceselor?
_ mutex
_ spinlock
_ semafor
_ monitor

Mutex-ul, Spinlock-ul si semafor-ul nu folosesc in implementare cozi pt gestionarea proceselor. Monitorul foloseste cel putin o coada(coada de intrare in monitor) (alte pt variabilele de conditie la care pot fi inregistrate thredurile sau procesele).

9) Pe o infrastructur_a de virtualizare ce foloses,te LXC se poate folosi comanda cp pentru a copia _s,iere de pe sistemul de baz_a (host) pe container (guest). Acest lucru nu poate fi, insa, realizat nativ pe o infrastructur_a de virtualizare ce foloses, te VMwareWorkstation (se presupune ca nu foloses,te feature-uri precum Shared Folders sau mount-uri in ret,ea). De ce?

In cazul VmWare nu se poate folosi cp, deoarece VMware ofera virtualizare totala, deci avem masini virtuale
compelt izolate una de alta ce ruleaza pe sistemul host,deci avem sisteme de fisire diferite pt fiecare sistem.
Pt a copia trebuie de folosit mecanize de virtualizare speciale.In cazul LXC - avem virtualizare la nivel de SO, deci SO host are pe langa process tree-ul sau si celelalte process tree, dar toate lucreaza in cadrul aceluiasi sistem de fisiere(al hostului),deci putem copia cu cp.

10) Un mesaj de tipul security advisory precizeaz_a Signed/unsigned comparison leads to buffer overflow. Care este cauza unei astfel de situat,ii? Dat,i exemplu de snippet de cod care corespunde situat,iei din mesaj.

O astfel de situatie poate avea loc atunci cand violam dimensiunile unui buffer prin incercarea de a accesa un index dincolo de limitele sale.
Ex. 1
```
int index, len = 9;
int array[len];
if (index < len) value = array[index];
else pritf ("Mesaj!\n");
```
problema e ca nu verificam si limita minima (>= 0), astfel indexul verificat poate ajunge a fi negativ Valoarea in memorie fiind foarte mica sau foarte mare generand un buffer overflow.
Ex. 2
```
#define MAX 10
char input[MAX];
char output[MAX];
fillBuffer(input);
int len = getInputLength();
if (len <= MAX) {
memcpy(output, input, len);
```
la comparatia signed int cu unsigned int signed int e convertit fortat la unsigned, deci pote sa aiba o valoare foarte mare care sa depaseasca zona de memorie alocata, deci generand un buffer overflow.

10) Un programator testeaz_a operat,iile blocante si non-blocante. Foloses,te funct,iile write() respectiv write_nonblock()^_n acest sens. Fie n_bytes si n_bytes_nonblock valoarea intoarsa de cele dou_a apeluri. Programatorul observ_a c_a, ^_n cazul ^_n care nu apar erori, valoarea minima a variabilei n_bytes este 1, iar valoarea minim_a a variabilei n_bytes_nonblock este 0. Cum explicat,i?

Primul write() este blocant si sincron deci suntem asigurati ca scrie numarul necesar de octeti in buferul specificat.Al doilea apel write_nonblock() este sincron si non-blocant() ,dar nu asigura scrierea numarului necesar de octeti, astfel in primul caz s-a scri sceva in buffer, iar in cel de al doila caz nimic.

10 iunie 2010-2011

1) Un sistem pe 32 de bit, i foloses, te memorie virtual_a s,
i pagini de 4K. Care dintre urm_atoarele
adrese virtuale pot referi, la un moment dat, aceeas, i adres_a _zic_a:
_ 0x43210078
_ 0x65430278
_ 0x76540018
_ 0x54320078
_ 0x87650021

0x43 - ERROR_BAD_NET_NAME
0x65 - ERROR_EXCL_SEM_ALREADY_OWNED
0x76 - ERROR_INVALID_VERIFY_SWITCH
0x54 - ERROR_OUT_OF_STRUCTURES
0x87 - ERROR_IS_SUBSTED
1, 2, 4 ca se termina in 78.

2) Cum explicat, i faptul c_a, des, i, ^_n general, procesele daemon au ca p_arinte procesul init, nu toate au fost create de init?

Nu toate procesele daemon sunt create de init, deoarece exista posibilitatea ca un alt proces sa apeleze fork() si apoi imediat sa isi incheie ciclul de viata, astfel incat procesul copil tocmai creat va deveni orfan si va fi adoptat de init.

3) De ce mecanismul ASLR (Address Space Layout Randomization) are sens pe sistemele pe 64 de bit, i, dar relevant,_a sc_azut_a pe sistemele pe 32 de bit, i?

Deoarece pentru sistemele de 32 bit , doar 16 bits sunt folositi pentru address randomization, astfel incat un atacator poate invinge acest sistem prin forta bruta intr-un timp relativ scurt(de ordinul minutelor) In cazul sistemelor de 64 bits... posibilitatile sunt evident mult mai numeroase(de ordinul milioanelor, deci metoda fortei brute nu mai este fezabila,deci ASLR-ul poate ajuta in acest caz.

4) De ce nu au sens algoritmi de plani_care de operat, ii pe disc precum C-SCAN, C-LOOK, pe discuri flash?

Algoritmii C-SCAN, C-LOOK sunt disk scheduling algorithms. Ei planifica miscarile bratului si capatului hard discului astfel incat operatiile de citire/scriere pe disc sa fie eficiente(in cazul C-SCAN) si determinta ordinea in care request-urile de citire/scriere pe disc sunt procesate(C-LOOK).
Acesti algortmi au sens in contextul unui numar mare de cereri de citire/scriere ca in cazul hardiscurilor. In cazul flash drive-urilor unde operatiile de citire/scriere sunt evident mult mai putine, acesti algoritmi nu-si au loc/sens.
Flash diskuri nu sunt folosite pt swap, HDD-urile ce folosesc C_SCAN si C-LOOK da.

5) ^In ce situat, ie, instruct,iunea
*a = 42;
genereaz_a o operat, ie de swap out?

Cand memoria fizica este full/alt proces vrea sa foloseasca pagina actuala(unde se afla adresa lui a) si un alt proces
a modificat datele din acea locatie de memorie spre care pointerul a arata, astfel incat este necesara operatie de swapare
a locatiei de memorie in backing storage, pentru a scrie valoarea 42.

6) Un proces execut_a urm_atoarea secvent,_a de cod:
```
fd1 = open(\a.txt", O RDWR);
write(fd1, \anaaremere", 10);
```
Se execut_a, ulterior, urm_atoarea secvent,_a ^_n diferite contexte:
```
fd2 = open(\a.txt", O RDWR);
write(fd2, \bogdanarepere", 13);
```
Contextele sunt:
_ ^_n cadrul aceluias, i proces
_ ^_ntr-un thread nou
_ ^_ntr-un proces copil al procesului init, ial
_ ^_ntr-un proces diferit s,
i nelegat de procesul init, ial
Fis, ierul a.txt va avea, ^_n _nal, acelas, i cont, inut, indiferent de contexte. Justi_cat, i. Se presupune c_a toate apelurile reus, esc.

Fisierul va avea acelasi cotinut independent de contextul rulat, deoarece
in fiecare context:cel de sus,de jos avem:
cele 2 contexte sunt independente si vizeaza acelasi fisier deschis cu
acelasi drepturi de acces, scrierea avand loc printr-un write() simplu.

7) Dat, i exemplu de situat, ie ^_n care un apel read sincron blocant nu se blocheaz_a (apelul reus es te - se intoarce cu succes).

citirea dintr-un fisier cu dimensiunea 0.

8) Inspectarea spat, iului de adres_a al unui proces arat_a c_a dimensiunea stivei la crearea procesului este de 128KB. Cu toate acestea, pe durata execut, iei procesului se apeleaz_a o funct, ie care foloses, te o variabil_a local_a ce are dimensiunea de 8MB f_ar_a a cauza o eroare. Cum explicatI acest lucru?

Deoarece stiva in momentul apelarii unei functii creste(in jos) deci isi mareste dimensiunea in concordanta cu necesitatile functiei. In cazul de fata desi initial ea are 128kb , ea se va mari pentru a acomoda variabilele locale ale functiei apelate. In momentul in care functia ajunge la sfarsit stiva se va micsora.

9) Fie urm_atoarea secevent,_a de cod:
```
#include <stdio.h>
void f(void)
```

```
{
printf("hellonn");
}
int main(void)
{
int *ptr;
ptr = (int *) f;
printf("0x%08xnn", *ptr);
*ptr = 0x12345678;
return 0;
}
```

La rularea codului se obt,ine rezultatul:

0xe5894855

Segmentation fault

Cum explicati?

Pt a printa adresa corect a functiei f() s-ar fi procedat astfel:
in main(): printf("0x%08x\n, *f);
In cazul de fata, deoarece se pune in ptr adresa lui f castuit la int*,
printf() nu va afisa adresa corecta(,dar e valida).Seg fault-ul apare, deoarece
se modifica valoarea lui *ptr cu o adresa invalida, astfel se produce stack overflow
(ptr e pe stack,stiva creste in jos si se supracrie ceva/aadresa de retur si PC sare
la o adresa aleatoare, crash-uind programul cu seg fault).

11) De ce, pentru o implementare de thread-uri la nivelul nucleului (kernel-level threads) durata schimb_arii de context ^_ntre dou_a thread-uri din procese diferite este vizibil mai mare dec^at durata schimb_arii de context ^_ntre dou_a thread-uri ale aceluias, i proces?

In cazul Kernel Level threads, managementul si planificarea firelor de executie sunt realizate
in kernel. Astfel comutarea contextului este efectuata de kernel cu o viteza de comutare mai mica
,deoarece trebuie sa se treaca dintr-un fir de executie in kernel si apoi kernelul va intoarce
controlul celui de-al 2 lea fir. Cand firele de executie apartin aceluiasi proces acest pas suplimentar
nu mai este necesar astfel incat viteza de comutare va fi mai mare.

9 iunie 2010-2011

1) Cum se modi_c_a zona de cod (text) a unui proces ^_n cazul cre_arii unui nou thread cu implementare la nivelul nucleului (kernel-level threads)?

zona de cod nu se modifica, ea e partajata intre toate threadurile procesului.

2) Se realizeaz_a un apel mmap pe urm_atoarele platforme:
_ un container OpenVZ
_ o mas,in_a virtual_a Xen
_ un sistem emulat prin Bochs (emulator)
Care este ordinea operat, iilor ^_n funct, ie de timpul de rulare (de la mic la mare)?
Justi_cat, i.

Box > Open VZ > Xen(cel mai prost este box) - timpul de rulare cel mai mare
Box fiind emulator, acesta va consuma multe resurse in incercarea de replicare "hardware" si a functiilor pe care vrea sa-l emuleze, cu rezultate mai lente.
Open VZ fiind un Virtual Enviroment, este un mediu izolat de executie virtualizarea facandu-se la nivelul sistemului de operare(operating sistem-level virtualization) . Partajarea resurselor fizice pentru VE. Avand o implementare avansata ce include managementul resurselor si folosind politica de de fair cpu scheduler, el se situeaza evident peste Box.
Xen - paravirtualization avand o interfata apropiata cu cea a sistemului fizic deci mecanismele de calcul intens computationale sunt rezolvate prin intermediul unor hook-uri ce permit rularea in mediul
nonvirtualizat sporind viteza si micsorand timpul de rulare.
When it comes to Xen and OpenVZ, to compare them, Xen will have to win. First of all, Xen is more of a hardware virtualization, which means its more closer to a physical system, unlike OpenVZ, which is more similar to chroot environment or software virtualization. Other issues, in OpenVZ, would be the fact that you are very limited to what you can setup, what you can modify or even build.
In all cases, OpenVZ doesn't really support any modules and in fact I do believe it can't even load any kernel modules, also because of this, iptables is very limited too. OpenVZ doesn't have swap space, is using the physical system swap space, can't have its own time server or locale, as is using the physical system's settings, but if you need a small system for a website, with not a lot of hits, maybe a blog or company site, then OpenVZ will be able to do the job. If you need something more serious, like maybe development environment, true hardware resources, jvm servers, an e-shop, then Xen is the winner and not just, like I said, its more useful to have Xen, as the performance is in every way much better, the only issue would be that it can't be managed using a panel like OpenVZ and can't have burstable RAM or CPU.

2) Justi_cat,i valoarea de adev_ar a a_rmat,iei: ^In cazul unui symlink, pointerii din cadrul inode-ului init,ial (_s,ierul origine) si pointerii din cadrul inode-ului symlink-ului refer_a aceleas,i blocuri.

Adevarat, lucrul prin symlinks e ca si cum ai lucra direct cu fisierul.
Chiar daca avem inodes diferite ele trebuei sa arata la aceleasi blocuri pe disk,
deoarece refera acelasi fisier.

3) Cum poate un apel printf s_a conduc_a la scrierea informat,iei pe un socket?

Prin executia instructiunilor:
inchidem stdout prin apelul close(1).
Redirectam stdout la descriptorul socketului prin dup2(sockfd,1);
Apelul printf("...") va scrie in socket.

5) Un semafor este init,ializat la valoarea 5. Mai multe thread-uri execut_a urm_atoarea secvent,_a de pseudocod:

```
down(&sem);
/* critical section */
up (&sem);
```

Care este num_arul maxim de thread-uri care poate as,tepta, la un moment dat, la semafor, respectiv care se pot g_asi, la un moment dat, ^_n regiunea critic_a (critical section)?

Daca avem n threaduri ce vor sa intre in regiunea critica si n > 5,
atunci 5 threaduri maxim se vor gasi in regiunea critica, iar n-5 vor
astepta, pt n <= 5 toate thredurile vor intra in regiunea critica.

6) De ce paginarea (^_n cazul memoriei) previne fragmentarea extern_a, dar nu s,
i fragmentarea intern_a?

Pagimarea presupune impartirea memoriei(virtuale si fizice) in pagini de o dimensiune fixa
astfel incat in momentul in care un proces are nevoie de memorie, ii este atribuita una/mai multe pagini.
Fragmentarea externa este prevenita deoarece avand paginile la o dimensiune fixa alocarea memoriei
va fi contigua. Fragmentarea interna pe de alta parte nu este prevenita deoarece unui proceS ii poate
fi atribuita o pagina de dimensiune fixa insa mai mare decat cea necesara.

7) Fie urm_atoarea secvent,_a de cod:

```
int flag = 0;
void *func(void *arg)
{
int a;
if (flag == 0) {
flag = 1;
a = 5;
}
else {
a++;
}
printf(\a= %dnn", a);
return NULL;
}
```

Funct,ia func este executat_a, serial/succesiv (nu se ^_ntrep_atrund), de dou_a thread-uri. Ce informat,ii vor _ a_s,ate?

flag-variabila globala.. threadurile partajeaza variabilele globale.
a=5; - primul thread
a=6; - al doilea thread

8) Care dintre urm_atoarele operat,ii cauzeaz_a un TLB ush:
_ write blocant apelat de un proces singlethreaded
_ write blocant apelat de un proces multithreaded cu implementare ^_n user-space
_ write blocant apelat de un proces multithreaded cu implementare ^_n kernel-space

A doua operatie(multithread, user level threads) va cauza TLB flush, deoarece
la blocarea unui singur thread user level se bloca intreg procesul, deci scheduler-ul
va fi nevoit sa fac un context switch cu un alt proces.La context switch TLB e flushed
pt a elimina inconsistenta datelor din TLB.

9) ^In urma rul_arii unui executabil folosind strace (pentru analiza apelurilor de sistem) rezult_a
urm_atoarele apeluri legate de biblioteca standard C (libc.so):

```
open("/lib/libc.so.6", O RDONLY) = 3
```

mmap(NULL, 3680360, PROT READjPROT EXEC, MAP PRIVATEjMAP DENYWRITE, 3, 0) =
0x7f312ab35000
mmap(0x7f312aeae000, 20480, PROT READjPROT WRITE, MAP PRIVATEjMAP FIXEDjMAP
DENYWRITE, 3, 0x179000)
= 0x7f312aeae000

De ce primul apel mmap foloses, te PROT_READ|PROT_EXEC, iar al doilea foloses, te PROT_READ|PROT_WRITE?

Acest caz este unul standard de incarcare(de catre loader(ld.so pe Linux)) a unei biblioteci dinamice in spatiul de adrese al procesului curent(open() si mmap()).
- Apelul open() deschide libraria in mod de acces citire.
- Apelurile mmap() sunt cu modificatorul MAP_PRIVATE, deci daca ar fi sa se modifice ceva in biblioteca,s-ar
modifica doar copia privata incarcata in procesul curent,nu si biblioteca insasi(care a fost incarcata).
- La primul apel mmap() se foloseste PROT_READ|PROT_EXEC pt ca avem cod binar executabil(care nu poate fi modficat),
dar poate fi folosit de proces.
- Al doilea apel mmap() mapeaza o zona modificabila de adrese in spatiul de memorie mapat cu primul apel mmap().
Al doilea apel prevede o zona ce va fi folosita pt date,deci se permite atat citirea cat si modifcarea lor.

10) Fie urm_atoarea secvent,_a de cod:

```
char *a;
void func(void)
f
for (int i = 0; i < NUM PAGES; i++)
a[i*PAGE SIZE] = 42;
g
int main(void)
f
/* pseudocod */
a = mmap(NUM PAGES * PAGE SIZE);
for (int i = 0; i < NUM PAGES; i++)
a[i*PAGE SIZE] = 42;
/* pseudocod */
create thread(func);
wait thread();
return 0;
g
```

C^ate page fault-uri se obt, in ^_n cadrul funct, iei func?

Toate page-faulturile necesare vor fi rezolvate in threadul main(ele se vor genera
din cauza demand paging-ului).Daca dupa crearea noului thread toate paginile
raman in RAM atunci in functia func() se nu se vor genera page faulturi,daca
unele pagini vor fi inlocuite/swapate atunci in func() se vor executa un numar mic de page faulturi.

5 septembrie

1) ^Intr-un sistem de _s,iere, numele _s,ierului este ret,inut ^_n inode. Poate cont,ine sistemul de _s,iere link-uri simbolice (symlinks)? Dar hard links?

Da sistemul de fisiere detine hard links - cai diferite catre acelasi fisier.
Da sistemul contine si symlinks(pointer catre fisierul original) - de obicei folosite la diferite apeluri de sistem(open() de exemplu).

2) O structur_a de date este accesat_a din contextul unui handler de tratare a unui semnal sI din contextul normal de rulare a unui program. Cum se asigur_a sincronizarea accesului la acea structur_a?

In cazul nostru parea ca am putea utiliza mutexuri pt sincronizare,dar nu putem, deoarece vom crea un deadlock.Cel mai bine ar fi sa blocam si sa deblocam semnalul necesar printr-o masca de semnale.Fie semnalul nostru SIG atunci un pseudocod posibil ar fi(in functia main()):

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIG);
sigprocmask(SIG_BLOCK, &set, NULL); //blocare semanl SIG
/* zona critica - acces la structura */
sigprocmask(SIG_UNBLOCK, &set, NULL); //deblocare semnal SIG
```

3) Un proces det,ine mai multe thread-uri. Se detectez_a c_a un thread a suferit un atac de tipul stack overflow ^_n timpul rul_arii sale. Este su_cient_a terminarea thread-ului pentru a garanta ^_ncheierea atacului?

Fiecare fir de executie are propriul context ce include stiva.Atackul stack overflow modifica doar stiva thredului victima.Daca thredul e terminat, stiva se va goli si problema e rezolvata.

4) Fie programul de mai jos:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
        long *ptr;
        ptr = (long *) main;
        printf("*ptr = %xnn", *ptr);
        *ptr = 0x1234;
        Retu     rn 0;
}
```

La rularea sa se obt,ine:

```
$ ./exec
*ptr = e5894855
Segmentation fault
```

Cum explicat,i?

Pt a printa adresa corect a functiei f() s-ar fi procedat astfel:
in main(): printf("*ptr = %x\n, *f);

In cazul de fata, deoarece se pune in ptr adresa lui f castuit la long*,
printf() nu va afisa adresa corecta(,dar e valida).Seg fault-ul apare, deoarece
se modifica valoarea lui *ptr cu o adresa invalida, astfel se produce stack overflow
(ptr e pe stack,stiva creste in jos si se supracrie ceva/aadresa de retur si PC sare
la o adresa aleatoare, crash-uind programul cu seg fault).

5) Fie cele dou_a instruct¸iuni de mai jos. Presupun^and c_a nu exist_a erori, ^_n ce situatie instruct¸iunea (A) dureaz_a mai mult dec^at (B) si invers?
_ (A) *ptr = 0x1234;
_ (B) read(fd, buffer, 1024);

In cazul normal in care avem spatiul deajuns in RAM,pe disk si pe SWAP (B) dureaza mult mai mult ca (A).
In cazul exceptional in care RAM e plin, avem foarte putin loc in SWAP si pagina necesara instructiunii (A) e pe disk, atunci paginarea va lucra foarte incet, fiind necesar sa fie inlocuita pagina
de pe RAM cu cea necesara de pe disk atunci, daca dupa ce se executa (A) sistemul revine la normal
si se executa (B), durand mai putin ca (A).

6) ^In implementarea uzual_a a unui nucleu sistem de operare, un pipe este reprezentat de un buffer. Dat¸i c^ate un exemplu de situat¸ie ^_n care, primind ca argument un descriptor de pipe, apelul read(), respectiv write() se blocheaz_a.

read() s-ar bloca daca bufferul e gol, s-au citit
toate datele si se incearca inca o citire
write() s-ar bloca daca bufferul e plin, s-au scris
toate datele si se incearca inca o scriere.

7) Un proces cont¸ine trei thread-uri. Unul dintre thread-uri realizeaz_a un access nevalid la memorie. Ce se ^_nt^ampl_a cu celelalte dou_a thread-uri?

Daca sunt User-Level-Threads atunci procesul se blocheaza.
Daca sunt Kernel-Level-Threads atunci celelalte threaduri isi continua executia,
threadul in cauza e terminat.

8) Trei thread-uri acceseaz_a o aceeas¸i structur_a de date: unul dintre thread-uri acceseaz_a structura in mod read-write, iar celelalte thread-uri ^_n mod read-only. Ce thread trebuie s_a foloseasc_a primitive de sincronizare?

Threadul ce acceseaza structura in mod read-write are nevoie de sincronizare
,deoarece el e singurul ce ar putea corupe datele(in timp ce el scrie ,altul citeste)
,in timp ce celelalte threaduri pot accesa structura concomitent fara probleme.

10) De ce este mai facil de realizat limitarea spat¸iului de disc pentru o mas¸in_a virtual_a VMware dec^at pentru un container OpenVZ?

In acest caz este mai usor de utilizat VMware decat OpenVZ ,deoarece VMware
ofera virtualiza completa, resure izolate, astfel se pot crea diskuri virtuale pt fiecare

masina virtuala din timp(izolare), in timp ce OpenVZ ofera aceleasi resurse pt toate containerele, deci e necesar mai multa management si efort din partea kernelului OpenVZ.

11) Fie trei procese A, B, C aate ^_n relat, ie de p_arinte - copil: A este proces p_arinte pentru B,
iar B este proces p_arinte pentru C. Cronologic, au loc urm_atoarele act, iuni (t2 > t1):
_ t1: C se termin_a, f_ar_a a _ as,teptat de B
_ t2: B se termin_a, f_ar_a a _ as,teptat de A
_ dup_a t2: A ruleaz_a ^_n continuare, f_ar_a a-l as,tepta pe B
^Incep^and cu momentul t1 utilizatorul investigheaz_a starea proceselor s,
i observ_a, in fiecare moment, un singur proces ^_n starea zombie. De ce?

Dupa t1: C e zombie - un singur proces zombie.
La t2: B moare, C e parentat de init il asteapta pe C si C nu mai e zombie/dispare.
Dupa t2: B devine zombie - un singur proces zombie.

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(f, buf, 32*1024);`
`fflush(f);`
`fwrite(f, buf, 32*1024);`
	`char buf[32*1024];`
	`int fd;`
	`char *a;`
	`fd = open("a.dat", O_RDWR | O_CREAT | O_TRUNC, 0644);`
	`/* fill buffer */`
	`…`
	`write(fd, buf, 32*1024);`
	`a = mmap(NULL, 32*1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);`
	`memcpy(a, buf, 32*1024);`

Operația fwrite 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 fault-uri la fiecare pagină. Aceasta se întâmplă pentru că mmap alocă memorie virtuală pură (demand paging). Primul acces la o pagină va conduce la page fault.

În concluzie, operația memcpy durează mai mult decât fwrite.

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 acestei implementări?

Folosirea mutexului global înseamnă că un singur filozofi 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 mașina 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 structurilor sistemului de fișiere (inode, bitfield, data block, dentry etc.)

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

*Se calculează blocul aferent adresei 32*1024 prin parcurgerea pointerilor de bloc din inode-ul MINIX. Întrucât se depășește spațiul referibil prin referință directă se va citi blocul aferent pentru referință indirectă.*

Se parcurge bitfield-ul și să găsește primul bloc liber. Adresa acelui 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 (procesul 1 așteaptă)
20ms-30ms: procesul 3 (procesul 2 așteaptă, procesul 1 este gata de execuție)
30ms-40ms: procesul 4 (procesul 3 așteaptă, procesul 1 și procesul 2 sunt gata de execuție)
40ms-50ms: procesul 5 (procesul 4 aștepată, 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, secvențiale). 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 a unui apel de sistem de 7ms.

Dacă zona indicată de a (20 de octeți) este poziționată 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 poziționată 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 schimbate 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 altei regiuni deja alocate și a adresei de retur (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_RDWR);
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(“%c”, *a1);
*a2 = ‘a’;
```

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

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 registrelor
- flush TLB
- schimbarea tabelei de pagini
- schimbarea tabelei 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 registrelor.

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. Știind că un acces la memorie durează 50ns iar o pagină este de 4KB, cat va dura secvența 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] = i;
```

*În absența TLB fiecare acces la memoria fizică necesită accesarea tabelei 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 memorie cu o durată totală de 64*1024*50ns.*

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

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

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

```
int esp;
int stack_val;

/* se obtine valorea registrului esp (registru de stiva) al thread-ului planificat anterior */
esp = get_former_esp();
stack_val = *(esp + 4);
```

Care va fi rezultatul execuției secvenței 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 acelei valori (nu se va obtine segmentation fault decât dacă thread-ul anterior și-a încheiat execuția).

5. Pe un sistem de fișiere MINIX se execută operația:

```
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 aceastase parcurge bitmapul de inode-uri și se alocă un inode. Se completează cu 1 poziția liberă găsită. Se creează un dentry cu numele "a.txt".
Dacă fișierul exista este trunchiat. Se parcurg pointer-ii de blocuri ai inode-ului și se marchează cu NULL (sau ceva echivalent). Se parcurge bitmapul de blocuri și marchează cu 0 pozițiile 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 (Fe_2O_3) (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 (alocare 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? 1r înseamnă operație de citire în cadrul paginii virtuale 1; 2w înseamnă operație de scriere în cadrul paginii virtuale 2.

Conținutul celor 3 pagini fizice, împreună cu evenimentul aferent este prezentat, evolutiv, în tabelul de mai jos; la fiecare două pagefault-uri se resetează bitul referenced):

frame 1	1r (R)	1r (R)	1r	1w (W)	3r (R)	3r (R)	3r	3r (R)	3w (RW)	3w (W)	2r (R)	2r (R)	2w (W)	6r (R)	6r (R)	6r	2r (R)
frame 2		2w (W)	2w (W)	2w (W)	2w (W)	2w (W)	4w (W)	4w (W)	4w (W)	4w (W)	4w (W)	5r (R)	5r (R)	5r	3w (W)	3w (W)	3w (W)
frame 3			4r (R)	4r (R)	4r	1w (W)	1w (W)	1w (W)	1w (W)	1w (RW)	1w (RW)	1w (W)	1w (W)	1w (W)	1w (W)	1w (W)	1w (W)
	PF	PF	PF	PF	PF	PF	PF		PF		PF	PF	PF	PF	PF		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 adresă validă) și nu există nici o modificare adusă apelului malloc.

Primul apel malloc generează un page fault, urmarea fiind alocarea unei pagini fizice întregi (chiar dacă se solicită alocarea unui singur octet). 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 tipul Overalapped I/O pe un sistem care dispune de un singur hard-disk?

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

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

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

Sisteme de operare

25 iunie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie **justificate**

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

Sistemele de operare moderne folosesc suportul de paginare pus la dispoziție 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 o alocare rămâne virtual contiguă. În felul acesta dispar problemele de fragmentare externă – adică de găsire a unui spațiu continuu pentru alocare (rămân însă problemele de fragmentare internă).
Excepție fac alocările din kernel-space care pot solicita alocare de memorie fizic contiguă sau alocările impuse de hardware (de exemplu DMA).

2. Un sistem de operare dispune de un planificator de procese care folosește o cuantă 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 situațiile 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 comunicației cu dispozitive de I/O rapide (procesul P1 planifică o operație I/O și se blochează, operația se încheie rapid și trezește procesul etc.).
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 reentrante 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 variabile partajate între thread-uri: funcția este thread-safe, dar nu este reentrantă (nu pot fi executate simultan două instanțe ale acestei funcții). Proprietatea de reentranță 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 fișiere FAT un fișier ocupă 5 blocuri: 10, 59, 169, 598, 1078. Știind că:

- un bloc ocupă 1024 de octeți
- o intrare în tabela 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 tabela FAT 4 octeți, deci sunt 256 intrări FAT într-un bloc. În tabela FAT intrările 10, 59, 169 se găsesc în primul bloc, intrarea 598 în al treilea bloc și 1078 în al cincilea bloc. Vor trebui, astfel, citite 3 blocuri asociate tabelei FAT. Fișierul ocupă 5 blocuri, deci vor fi citite, 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, cursoare de fișier diferite.

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 retur a unei funcții cu o adresă de pe stivă. Bitul NX previne execuția de cod pe stivă. Dar adresa de retur 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_data();
        wake_thread2();
        wait_for_data_from_thread3();
    }
}
```

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

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

Care este dezavantajul acestei abordări? Propuneți o alternativă.

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

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

Fiind 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 astfel 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 fi configurat 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 alocă memorie pur virtuală (fără suport de memorie fizică). Alocarea de memorie fizică se va realiza la cerere (demand paging). malloc va întoarce NULL în momentul în care procesul rămâne fără memorie virtuală în user-space. N va avea, așadar, valori aproximative de 2048 și 3072. Dimensiunea memoriei RAM a sistemului este nerelevantă 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 deschiși. stderr, stdout, stdin sunt 3 descriptori inițiali, rezultând 1012 descriptori. Restul au fost creați prin alte metode. File descriptorii pot fi creați și altfel: 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

26 iunie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie **justificate**

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 down(&sem); durează doar 1µs. Apelul se realizează cu succes. Care este explicația?

Apelul down este implementat în user-space (de exemplu o implementare de tip futex). 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 unei implementări de thread-uri kernel-level, acest lucru va presupune și un apel de sistem (planificatorul este implementat în kernel-space).

3. De ce este mai avantajos ca, pe un sistem uniprocesor, 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 ar fi planificat primul, atunci apelurile de scriere ale acestuia vor rezulta în duplicarea paginilor (overhead temporal și consum memorie) datorită copy-on-write. Dacă procesul copil face exec, atunci acele duplicate au însemnat un consum inutil de resurse.

4. Există vreo 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-uri ar conduce la incosistența informațiilor referitoare la erorile apărute.

5. Un sistem uniprocesor (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 o schimbare de context presupune doar schimbarea registrelor 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 comenzii *pmap* se observă următoarele informații despre biblioteca standard C:

```
# pmap 1
base address          size    rights     name
[...]
00007f8c480e6000    1320K     r-x--      libc-2.7.so
00007f8c4842f000      12K     r----      libc-2.7.so
00007f8c48432000       8K     rw---      libc-2.7.so
```

Presupunând că în sistem rulează 50 de procese care folosesc biblioteca standard C, care este spațiul 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, așadar, 50*8K + 12K + 1320K.*

7. Descrieți și explicați 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 retur a funcției strcpy. Valoarea de tip canary trebuie stocată la o adresă mai mică decât adresa de retur a funcției strcpy (practic, la fel ca la o stivă care crește în jos). Întrucât apelantul este cel care construiește stack frame-ul apelatului, 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 octeți – numele
- 4 octeți - numărul inode-ului

O instanță a unui astfel de sistem de fișiere deține un director rădăcină, 5 subdirectoare, iar fiecare subdirector conține 5 fișiere. Câte dentry-uri deține sistemul de fișiere?

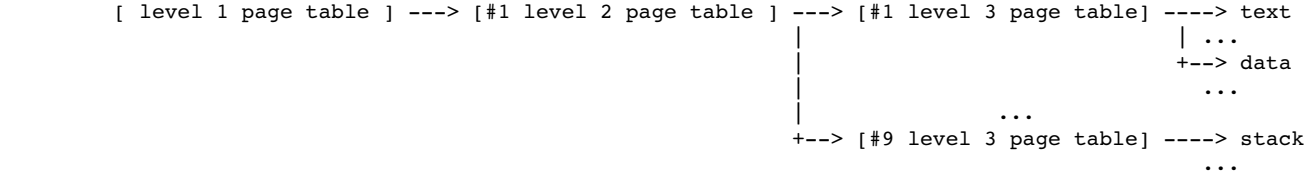
*Fiecare intrare în sistemul de fișiere (director sau fișier) conține cel puțin un dentry. Ignorând intrările speciale . și .. rezultă (1 +) 5 + 5*5 = 30 (31) intrări. Directorul rădăcină poate să nu aibă dentry. Considerând intrările speciale, rezultă un plus de 1 + 2*5 intrări = 11 intrări (directorul rădăcină nu are referință ..).*

9. Un sistem pe 64 de biți folosește pagini de 8KB și 43 de biți pentru adresare 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 componență a spațiului de adrese (se începe de la adresa 0):

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

Știind că o intrare în tabela de pagini ocupă 64 de biți, câte pagini ocupă tabelele 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 există 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. A 9-a pagină de tabela va folosi primele 8 intrări pentru a referi paginile de pe stivă.
Prima pagină de tabelă de pe nivelul 2 va avea valide 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:



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

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

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 data1[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] = 2009;
}

int main(void)
{
    int *data2 = malloc(1024*1024 * sizeof(int));

    print_time("before init data1");
    init_array(data1, 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: 1245582962s, 753431us
after  init data1: 1245582962s, 767496us

before init data2: 1245582962s, 767524us
after  init data2: 1245582962s, 776012us




---
Se observa ca:
-   durata initializare data1 - 14065us
-   durata initializare data2 -  8488us
```

Cum explicați faptul că inițializarea vectorului *data1* durează mai mult decât inițializarea vectorului *data2*?

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

1. Care din următoarele instrucțiuni **ar putea** suprascrie adresa de retur 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 si alocare */

/* instructiuni */
a)  a = my_func;
b)  *(&a + 4) = my_func;
c)  *(a + 0x4000000) = my_func;
d)  memcpy(my_func, a, sizeof(void *));
```

Cuvânt cheie: ar putea. Unele situații sunt improbabile 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 adresă de heap (punctează către zona de 30 de octeți alocată 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 unele variabile între [ret][ebp] și [a] atunci &a+4 poate puncta către adresa unde se găsește valoarea de retur și *(&a + 4) o poate suprascrie*
c) posibil; a+0x4000000 înseamnă adunarea a 0x4000000 la o adresă de heap (valoarea lui a); se poate ajunge (greu probabil, dar posibil) la o adresă de retur de pe stivă
d) ciudată, se suprascrie o informație din zona de cod (zona read only); pot apărea erori de acces sau comportanent nedeterminist (în nici un caz nu suprascrierea adresei de retur dintr-o 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 prioritatea mai mare? De ce?

Proces care calculează transformate Fourier – CPU intensive. Proces care caută informatii într-o ierarhie de fisiere – 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 sistemului*
- *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 mapări de pagini fizice și pagini virtuale. Conține un subset al tabelei de pagini. Memoria fizică și memoria virtuală pot fi oricât de mici/mari. Nu sunt afectate de dimensiunea TLB-ului.

4. Completați 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 (functie), stderr și stdout (FILE *).

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

În pseudocod Linux, lucurile 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:

```
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 observația 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, proababil eroare (SIGSEGV) în cazul în care pagina anterioară este nevalidă (destul de probabil)*
- *n >= 2048 (peste cele două pagini), probabl 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 echilvalent 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 (întreruperi dese, se diminuează timpul de așteptare pentru fiecare proces*

100ms

- *productivitate (throughtput) 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 situații este mai probabilă înlocuirea majorității intrărilor din TLB? Motivați. (Argumentele și modul de folosire a funcțiilor se prespun 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 comune procesulelor, zone din spațiul kernel). De aici cuvântul "majorității".

Când se realizează un context switch?

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

În cazul celor două apeluri doar ultima variantă are sens (blocarea 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 magistrala 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 dintr-o rutină de tratare a unui semnal (signal handler). De ce nu este malloc o funcție signal-safe? Oferiți o secvență de cod pentru exemplificare.

După cum s-a menționat în cateva rezolvări (fără a aduce o contribuție în cadrul răspunsului, însă) funcțiile signal-safe sunt practic echivalente cu funcțiile reentrante. O funcție non-signal-safe este o funcție care folosește variabile statice, astfel că, dacă un semnal întrerupe funcția în programul principal și funcția este rulată în handler este posibil să apară inconsistențe (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 întrerupe funcția malloc și, în handler, rulează funcția malloc structurile interne folosite de libc pentru gestiunea alocării memoriei procesului vor fi date 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/34At)

Scenariul de exemplificare este de forma:

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

int main(void)
{
		....
	void *a = malloc(BUFSIZ);    /* aici sosește 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 < BUFLEN; i++)
    printf("%c", buf[i]);
```

iar altul

```
for (i = 0; i < BUFLEN; i++)
    write(1, buf+i, 1);
```

Care secvență durează mai mult? De ce?

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

- *se umplu buffer-ele*
- *se apelează fflush(stdout)*

- *se transmite newline (\n)*

Apelul de sistem write face apel de sistem de fiecare dată rezultând un overhead important.

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

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

razvan@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 starea READY. Precizați și motivați două acțiuni care determină trecerea acestuia in 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 din coada de procese READY*
- *lui P îi este crescută prioritatea peste a lui Q*
- *lui Q îi expiră cuanta și P este primul în coada de procese READY*
- *Q efecuează o operație blocantă (trece în blocking) și P este primul proces în coada de procese READY*

6. De ce obținerea ordonată/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 așteptare mutuală (deadly embrace) = deadlock.

Nu există nici o legătură directă în lock-uri și fenomenul de starvation. Fenomentul de starvation caracterizează o durată de așteptare foarte mare pentru un proces gata de rulare. Alte procese îi iau tot timpul "fața" și procesul nu ajunge pe procesor. Se spune că sistemul nu este "fair" (echitabil). Principala formă de asigurare a echității este folosirea noțiunii de cuantă și, în lumea Linux, de epocă și folosirea priorității dinamice a proceselor. Orice formă de locking duce la creșterea nivelului de starvation. Un proces care așteaptă la un semafor intrarea într-o regiune critică dar alte procese intră înaintea sa. Asigurarea fairness-ului poate fi asigurată prin strategii de tipul 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-ul referit corespunde sistemului local de fișiere, nu altui sistem de fișiere (nu există un identificator al sistemului de fișiere, se presupune cel local).

2. Fie următoarea secvență de comenzi:

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

Comanda ln 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 (aferent fișierului proaspăt creat)
ln a.txt b.txt → 1 dentry (b.txt) ca hard-link la a.txt
ln -s b.txt c.txt → 1 dentry (c.txt) și 1inode (aferent simbolic link-ului proaspăt creat)

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

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

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

Într-una 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 generează un page fault pentru fiecare pagină. Depinzând de alinierea celor 5000 de octeți pot rezulta două sau trei page fault-uri.

De exemplu, în cazul în care se alocă [500 octeți, 4096 octeți, 404 octeți] pe parcursul a trei pagini, vor rezulta 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, operație care durează T1. Imediat după prima rulare, se execută din nou programul și durează T2. T2 este semnificativ mai mic decât T1. Cum explicați?

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 timpul de citire va fi redus – diferența de acces la memorie față de disc este mare.

5. Care proces este părintele proceselor zombie?

Un proces zombie este un proces care și-a încheiat execuția dar a cărui stare nu a fost "analizată" de procesul părinte – adică procesul părinte nu a apelat wait pentru culegerea de informații despre procesul copil. Drept urmare, procesul zombie are același proces părinte ca procesul obișinuit înainte să-și fi încheiat 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 referită 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 aanalizează 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 invalid. 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 retur a unei funcții este probabil să fie suprascrisă. În cazul particular al secvenței de cod de mai sus, se realizează un buffer overflow la nivelul heap-ului, adică la nivelul variabilei buffer (alocată pe heap folosind malloc). Drept urmare, folosirea stack smashing protection nu are nici o utilitate.

8. Un sistem S1 foloseşte segmentare. Timpul de translatare 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 fizică duce la interogarea tabelei 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/selectorii de segment sunt menținuți în registre ale procesorului atunci timpul T1 este mai mic decât timpul T2.

Dacă descriptorii/selectorii 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:
a) unei mașini virtuale VMware Workstation;
b) 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 vreun fel.

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

10. Fie următoarele două secvențe de programe

```
       /* 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);
```

```
       /* 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(&tid, thread_handler, NULL);
pthread_join(&tid, NULL);
write(fd, "b", 1);
```

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 secvenței 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 secvenței 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 nevalid. Operația va întoarce EBADFD (Bad file descriptor).

11. Fie următoarea secvență de operații:

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

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

*Secvența de mai sus conduce la N accese la elemente 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 memorie. În al doilea caz se produc 2*N accese, deci fiecare acces la un element al vectorului înseamnă 2 accese la memorie.*

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 adresare 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 - țin de evoluția procesului – alocarea memoriei; alocarea pe heap se realizează prin malloc iar alocarea 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 unui page fault; page fault-ul va conduce la alocarea unei pagini fizice noi și marcarea acesteia cu drepturi de scriere.

3. Presupunem că avem 3 page frames la dispoziție, toate inițial goale. Se realizează următorul șir 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**	2	**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 anomalia 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 prezentate 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. 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 și nu se produce o analiză a priorităților proceselor, nu va conduce la invocarea scheduler-ului. Apelul 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" replanificării acestuia din cauza priorității proceselor sau a altor euristici de planificare ale nucelului.

6. Un sistem de fișiere dispune de un bitmap pentru inode-uri (un bit specifică folosirea sau nu a unui inode) de 32KB. Câte 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 câteva inode-uri aferente directorului rădăcină și fișierelor reale către care punctează symbolic link-urile.*

7. Se dă următoarea secvență de execuție :

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

Realizați sincronizarea celor 2 fire de execuție folosind semafoare astfel încât work_a1 să se execute înainte de work_b2 şi 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 a ajuns la punctul de întâlnire*
** s_b, thread-ul B a ajuns la punctul de întâlnire.*

Soluția de sincronizare este cea de mai jos:

```
            sem_t s_a, s_b;
            sem_init(&s_a, 0);
            sem_init(&s_b, 0);

   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 nu este de dorit?

În cazul unei implementări de thread-uri la nivel user, blocarea unui thread conduce la blocarea întregului proces. Thread-ul care execută operații I/O va avea parte de situații dese de blocare (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. Acest 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 ar avea 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 moștenește descriptorii procesului părinte. Acest lucru atrage două dezavantaje importante:
** securitate: un proces poate citi, parcurge sau corupe datele din fișierele unui alt proces*
** resurse: menținerea descriptorilor deschiși duce la ocuparea unui număr mare de descriptori de fișier; în cazul în care se creează procese în continuare, tabela de descriptori de fișiere este ocupată în mare măsură de fișiere deschise de alte procese*

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

întorc cu succes)

```
acquire_mutex(&m);
write(fd, "abcd", 4);
free(p);
release_mutex(&m);
```

În cazul unei implementări de mutex-uri în user space (de tipul futex) și a unui mutex neocupat, funcțiile acquire_mutex și release_mutex nu generează apel de sistem.

Apelul de bibliotecă free poate să nu genereze apel de sistem (de obicei nu generează) depinzând de implementarea din biblioteca standard C. Atât apelul malloc cât și free alocă, respectiv eliberează, anumite dimensiuni de memorie din heap. În momentul în care se "cumulează" o zonă suficient de mare (de alocat sau eliberat), se realizează apel de sistem (brk).

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

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

11. 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. Indexul intrării în tabelă reprezintă frame-ul aferent. Partajarea unei pagini se poate realiza în măsura 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 alocarea unei liste de elemente de tip (PID, pagină virtuală) în cadrul unei intrări în 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).

Sisteme de operare

26 iunie 2010

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 oricâte procese copil, în limita resurselor sistemului. Procesul init poate fi considerat un proces particular care nu are un proces părinte.

2. Un proces execută secvența:

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

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

Întrucât secțiunea de mai sus nu este blocantă, procesul poate fi preemptat dacă îi expiră cuanta 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 aceluiași 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.

Avem o variabilă alocată pe stivă (adresa ei este o adresă din stivă), iar conținutul acelei variabile (pointer) este o adresă întoarsă de apelulul 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 primele. Presupunând un flux continuu de procese cu timp de rulare redus, procesele cu timp de rulare mare vor ajunge să se execute foarte rar 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 registrelor se schimbă. Fiecare thread dispune de valori proprii ale registrelor. Astfel, registrele generale și registrul de stivă se schimbă. Sistemele de operare moderne folosesc rar registrele de segment, astfel că în general acestea nu vor fi schimbate.

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

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

Bibliotecile partajate conține zone r-x (cod), r-- (read-only data) și rw- (date). Zonele read-write conțin variabile care pot fi scrise 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 nealocate. Sistemul dispune de o singură pagină fizică disponibilă. Care este numărul maxim de pagini virtuale care pot fi asociate cu pagina fizică?

*Oricâte pagini virtuale pot fi asociate cu o pagină fizică. Implementări de tipul mmap permit maparea unei zone de memorie virtuale peste o zonă de memorie fizică. În situația de mai sus, un proces poate mapa toate cele M pagini virtuale proprii peste acea pagină fizică. În total, pentru cele N procese, se pot mapa N*M pagini virtuale.*

10. Explicați de ce nu se poate implementa mecanismul 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. Mecanismul de memorie virtuală permite existența unui spațiu virtual de adrese care depășește spațiul fizic – spațiul suplimentar poate fi furnizat de disc, prin intermediul swap-ului. În cazul absenței mecanismului de memorie virtuală, nu se poate referi/folosi spațiul de swap, deci nu se poate implementa mecanismul 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 dentry ocupă 64 de octeți, câte intrări 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 așadar, că un inode poate referi 10240*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*4KB / 64 dentry-uri, adică 10240 * 64 = 655360.*