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?

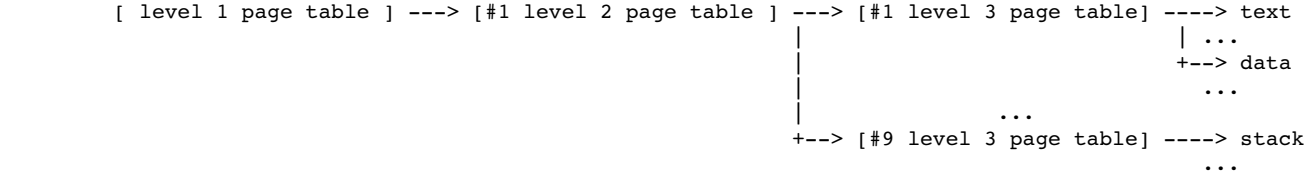
*Fiecare intrare în sistemul de fișiere (director sau fișier) conține cel puțin un dentry. Ignorând intrările speciale . și .. rezultă (1 +) 5 + 5*5 = 30 (31) intrări. Directorul rădăcină poate să nu aibă dentry. Considerând intrările speciale, rezultă un plus de 1 + 2*5 intrări = 11 intrări (directorul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