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).