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