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.*