# 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);`<br>`write(fd, "P1", 2);`<br>`--- schedule ---` | |
| | `--- schedule ---`<br>`fd = open("/tmp/a.txt", O_CREAT \| O_RDWR, 0644);`<br>`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.*