USER SPACE – KERNEL SPACE

1. De ce nucleul sistemului de operare rulează, în general, într-un spațiu dedicat, numit *kernel space*?
   - **Răspuns**: Pentru că în *kernel space* au loc operații privilegiate. Spațiul kernel este un spațiu privilegiat la care doar nucleul sistemului de operare are acces. În felul acesta se păstrează securitatea sistemului, orice operație privilegiată necesitând trecerea în spațiul kernel și acordul nucleului sistemului de operare pentru execuție.
2. Ce este un apel de sistem?
   - **Răspuns**: Mecanism care asigură trecerea din user space în kernel space la solicitarea user space. Este folosit atunci când user space-ul nu are privilegiile de a realiza o operație și apelează la kernel space pentru acest lucru.
3. De ce este utilă separația user space / kernel space?
   - **Răspuns**: Spațiul kernel are privilegii complete la nivelul sistemului. Operațiile privilegiate nu pot fi realizate în user space din motive de securitate a sistemului. În aceste situații user space-ul apelează la kernel space prin intermediul unui apel de sistem.
4. De ce apelul de bibliotecă strcpy nu generează apeluri de sistem?
   - **Răspuns**: Un apel de sistem are loc în momentul în care este nevoie ca o operație privilegiată să fie realizată de kernel. Întrucât strcpy copiază octeți dintr-o zonă de memorie în altă zonă de memorie nu realizează operație privilegiată și, deci, nu necesită apel de sistem.
5. Precizați un rol al nucleului sistemului de operare.
   - **Răspuns**: Nucleul de operare gestionează memoria sistemului. Asigură separația între procesele sistemului la nivel de memorie pentru a preveni unul să scrie în spațiul de adresă al altuia.
6. De ce aduce un apel de sistem mai mult overhead decât un apel de funcție obișnuit?
   - **Răspuns**: Un apel de sistem aduce mai mult overhead datorită comutării în kernel-space, în timp ce un apel de funcție se execută în user-space.
7. Care dintre următoarele apeluri durează cel mai mult: `strcpy`, `strdup`, `strchr`?
   - **Răspuns**: strcpy copiază șirul iar strchr caută un caracter în șir; strdup realizează operație similară strcpy dar, în plus, alocă spațiu pentru noul șir, operație costisitoare ce poate însemna și efectuarea unui apel de sistem. În concluzie, strdup durează, în general, cel mai mult.
8. Un apel de bibliotecă (libc) poate invoca între `X` și `Y` apeluri de sistem. Ce valori au `X` și `Y`?
   - **Răspuns**: (X, Y) = (0, inf) poate să nu invoce nici un apel de sistem (vezi strcpy) sau mai multe apeluri de sistem (teoretic infinite); nu există o limitare pentru ca un apel de bibliotecă să apeleze mai multe apeluri de sistem (sau foarte multe), doar că nu este ceva comun.

SISTEMUL DE FISIERE

1. În ce situație practică este folosit apelul `dup()`?
   - **Răspuns**: Apelul `dup()` este folosit practic pentru redirectarea ieșirii, intrării sau erorii standard în fișier. Altă situație practică este pentru operatorul `|` (*pipe*) de comunicare între procese.
2. Ce conține tabela de descriptori de fișier a unui proces?
   - **Răspuns**: Tabela de descriptori de fișier a unui proces conține pointeri; ca structură de date este un vector de pointeri. Acești pointeri referă structuri de fișier deschis de proces. Când un proces deschide un fișier, se alocă o structură de fișier deschis, iar adresa acestei structuri este stocată într-un loc liber (indicat de descriptorul de fișier) din tabela de descriptori de fișier.
3. Care este un avantaj al apelurilor de tipul *buffered I/O* (precum `fread`, `fwrite`) și care este un avantaj al celor de tipul *system I/O* (precum `read`, `write`)?
   - **Răspuns**: Apelurile de tipul *buffered I/O* fac mai puține apeluri de sistem, deci overhead mai redus, întrucât informația este ținută în buffere până la nevoia de flush. Sunt, de asemenea, portabile. Apelurile de tipul *system I/O* au o latență mai redusă, informațiile ajung repede pe dispozitiv. De asemenea, apelurile de tipul `system I/O` nu alocă memorie suplimentară pentru buffering, sunt mai economice din acest punct de vedere.
4. Un descriptor de fișier gestionează/referă, în general, un fișier obișnuit (*regular file*). Ce altceva mai poate referi?
   - **Răspuns**: Un descriptor de fișier mai poate referi un director, un link simbolic, un pipe, un socket, un dispozitiv bloc sau caracter. Toate aceste entități sunt gestionate de un proces prin intermediul unui descriptor de fișier.
5. Dați exemplu de apel care modifică dimensiunea unui fișier.
   - **Răspuns**: Apeluri care pot modifica dimensiunea unui fișier sunt `write` (poate scrie dincolo de limita unui fișier), `ftruncate` (modifică chiar câmpul dimensiune) sau `open` cu argumentul `O_TRUNC` care reduce dimensiunea fișierului la 0.
6. Câte tabele de descriptori de fișier există la nivelul sistemului de operare?
   - **Răspuns**: Fiecare proces are o tabelă de descriptori de fișier, deci vor exista, la nivelul sistemului de operare, atâtea tabele de descriptori de fișier câte procese există în acel moment în sistem.
7. Dați un exemplu de informație care se găsește în structura de fișier deschis și un exemplu de informație care se găsește în structura de fișier pe disc (*inode*).
   - **Răspuns**: În structura de fișier deschis se găsesc cursorul de fișier, permisiunile de deschidere a fișierului, pointer către structura de fișier pe disc. În structura de fișier pe disc se găsesc permisiuni de acces, informații

despre utilizatorul deținător, grupul deținător, dimensiunea fișierului, timpi de acces, tipul fișierului, pointeri către blocurile de date.

8. Ce conține și când este populată o intrare din tabela de descriptori de fișier a unui proces?
   - **Răspuns**: Este un pointer la o structură de fișier deschis. Când se deschide un fișier (folosind fopen, open, CreateFile) se creează o nouă structură de fișier deschis iar adresa acesteia este reținută în cadrul intrării din tabela de descriptori de fișier.
9. Ce este un descriptor de fișier? Ce fel de operații folosesc descriptori de fișier?
   - **Răspuns**: Este un număr (întreg) ce referă o intrare în tabela de descriptori de fișier. Este folosit în operații de lucru cu fișiere, pentru a identifica un fișier deschis.
10. Ce rol are cursorul de fișier al unui fișier deschis? Când se modifică?
    - **Răspuns**: Stabilește care este poziția curentă de la care vor avea loc operații la nivelul fișierului. Dacă valoarea sa este 100 și un apel read citește 30 de octeți, valoarea sa va ajunge la 130 de octeți. Se modifică și la apeluri de scriere sau la apeluri specifice de poziționare (seek).
11. Care intrare din tabela de descriptori de fișier este modificată în cazul apelului cu redirectare "./run > out.txt" față de cazul rulării simple "./run"?
    - **Răspuns**: Se modifică intrarea aferentă ieșirii standard a procesului (standard output), în general cea cu indexul 1. Aceasta întrucât operatorul > este redirectarea ieșirii standard. Acum intrarea de la indexul 1 din tabela de descriptori de fișier va referi fișierul out.txt, nu ieșirea standard a sistemului.
12. Știind că apelul write(42, "X", 1), executat în procesul P, se întoarce cu succes, care este numărul minim de fișiere deschise **de** procesul P? De ce? Antetul apelului write este write(fd, *buf, count).
    - **Răspuns** Numărul minim de fișiere deschise **de** procesul P este 0, deoarece este posibil ca toate fișierele să fi fost deschise de părintele lui P. Numărul minim de fișiere deschise **în**procesul P este 1, și anume fișierul cu descriptorul 42, deoarece este posibil ca toți ceilalți descriptori de fișier să fie închiși.
13. Fie secvența de pseudocod:

```
for (i = 0; i < 42; i++)

      printf(...);
```

Care este numărul minim, respectiv numărul maxim de apeluri de sistem din secvența de mai sus?

- **Răspuns** Numărul minim de apeluri de sistem din secvența de mai sus este 0. Dacă printf scrie la terminal, este line buffered și nu se va executa apel de sistem dacă nu se umple buffer-ul sau nu a fost primit caracterul '\n'. Numărul maxim de apeluri de sistem este 42, dacă în fiecare iterație a for-ului se umple buffer-ul sau a fost primit caracterul '\n'.

14. De ce apelul fclose realizează în spate apel de sistem, dar apelul printf nu întotdeauna?
    - **Răspuns** Apelul fclose realizează în spate apel de sistem, deoarece închide un fișier, modificând tabela de descriptori din proces. Apelul fclose se mapează pe apelul de sistem close. Apelul printf scrie într-un buffer, iar apelul de sistem write se realizează dacă se umple buffer-ul sau a fost primit caracterul '\n'.
15. Fie P1 și P2 două procese diferite. Când este posibil ca modificarea cursorului de fișier pentru un descriptor din P1 să conducă la modificarea cursorului de fișier pentru un descriptor din P2?
    - **Răspuns** Această situație este posibilă dacă cele două procese au un proces "strămoș" comun și descriptorul de fișier nu a fost închis de niciunul dintre procese. Atunci, modificarea cursorului de fișier pentru un descriptor din P1 poate conduce la modificarea cursorului de fișier pentru același descriptor din P2.
16. Care este numărul minim de descriptori de fișier valizi în cadrul unui proces? În ce situație este posibilă această valoare?
    - **Răspuns** Numărul minim de descriptori de fișier valizi în cadrul unui proces este 0, în cazul în care un proces închide toți descriptori de fișier, inclusiv stdin, stdout, stderr. Un astfel de proces este numit daemon.
17. Unde este poziționat cursorul de fișier `fd1` în urma secvenței de mai jos? Presupuneți că toate apelurile se întorc cu succes.

```
18. fd1 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
19. fd2 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
20. write(fd2, "1", 1);

    dup2(fd2, fd1);
```

    - **Răspuns**: În urma apelului `open`, cursorul de fișier `fd2` va poziționat la început. După `write`, acesta va poziționat la 1 octet după începutul fișierului, iar după `dup2`, și cursorul de fișier `fd1` va poziționat la 1 octet după începutul fișierului.
21. Fie secvența de pseudocod de mai jos:

```
22. fd1 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
23. pid = fork();

24. switch (pid) {
25.     case 0:
26.         break;
27.     default:
28.         dup(fd1);

    }
```

Presupunând că toate apelurile se întorc cu succes, câți descriptori din fiecare proces vor referi fișierului `a.txt`?

- **Răspuns**: 3 descriptori; 2 descriptori în părinte (`fd1` și descriptorul rezultat în urma `dup`) și 1 descriptor în copil (`fd1`, moștenit de copil în urma `fork`).

29. Care este numărul minim de descriptori de fișier ai unui proces pot referi, la un moment dat, `stderr`(standard error)? De ce?
    - **Răspuns**: Numărul minim este 0, deoarece `stderr` poate fi închis prin apel `close`.
30. Fie secvența de pseudocod de mai jos:

```
31. fd1 = open("a.txt", O_RDONLY);
32. fd2 = open("b.txt", O_RDWR);
33. dup2(fd1, fd2);

    write(fd2, "X", 1);
```

Care sunt valorile posibile ce pot fi intoarse de apelul `write`?

- **Răspuns**:
  1. Dacă toate apelurile se întorc cu succes, în urma apelului `dup2`, `fd2` va puncta către `a.txt`deschis `O_RDONLY`, iar apelul `write` va întoarce -1 și va seta errno la valoarea `EBADF`, pentru a semnala eroarea.
  2. Dacă apelul `dup2` eșuează, `fd2` va puncta către `b.txt` deschis `O_RDWR`, iar apelul `write` va întoarce 1, dacă a scris caracterul, sau 0, dacă nu a scris caracterul.

34. De ce apelul `fopen` realizează în spate apel de sistem, dar apelul `memcpy` nu?
    - **Răspuns**:
    - `fopen` realizează apelul de sistem `open` pentru a putea deschide/crea un fișier sau dispozitiv, pentru acest lucru fiind necesară trecerea în kernel-space.
    - `memcpy` nu realizează apel de sistem deoarece scrie și citește memorie deja alocată în spațiul de adresă al procesului fără a trece în kernel-space.
35. Fie un fișier `a.txt` având dimensiunea de 1024 octeți și secvența de pseudocod de mai jos:

```
36. fd1 = open("a.txt", O_RDWR | O_TRUNC);
37. close(fd1);

    fd1 = open("a.txt", O_RDWR | O_APPEND);
```

Unde va fi poziționat cursorul de fișier ale descriptorului `fd1`? De ce?

- **Răspuns**: În urma primului apel `open`, flag-ul `O_TRUNC` reduce dimensiunea fișierului la 0. Al doilea apel `open` poziționează cursorul la sfârșitul unui fișier gol, adică pe poziția 0. Dacă fișierul `a.txt` nu există, toate apelurile vor întoarce -1.

38. Dați exemplu de două apeluri care modifică valoarea cursorului de fișier (file pointer).
    - **Răspuns**:
      1. `lseek`/`fseek`, apeluri al căror rol este de modificare a cursorului de fișier;
      2. read/fread/fgets – la fiecare citire cursorul de fișier este incrementat cu numărul de octeți citiți;
      3. write/fwrite/fputs/fprints – la fiecare scriere cursorul de fișier este incrementat cu numărul de octeți scriși;
      4. ftruncate – trunchiază fișierul (cursorul este plasat pe 0);
      5. apelurile echivalante Windows.
39. Unde este poziționat cursorul de fișier în urma apelului:

```
    open("a.txt", O_CREAT | O_RDWR, 0644);
```

Dar în urma apelului:

```
    open("a.txt", O_RDWR | O_TRUNC);
```

Se presupune că apelurile se întorc cu succes.

- **Răspuns**: De fiecare dată cursorul este plasat la începutul fișierului: în prima situație se va începe citirea/scrierea de la începutul fișierului; în a doua situație fișierul este trunchiat și cursorul se află la început. Singura situație în care cursorul este plasat altundeva în momentul deschiderii acestuia este aceea în care se folosește flag-ul `O_APPEND`.

40. Câți descriptori de fișier ai unui proces pot referi, la un moment dat, ieșirea standard (standard output)?
    - **Răspuns**: Oricâți, în limita dimensiunii tabelei de descriptori de fișier a procesului, prin intermediul folosirii apelului `dup`/`dup2`:

```
    for (i = 3; i < getdtablesize(); i++)

        dup2(1, i);
```

1. Care dintre următoarele apeluri întoarce un întreg: `open`, `read`, `malloc`, `fopen`?
   - **Răspuns**:
     - open întoarce un file descriptor (întreg) – **DA**
     - read întoarce numărul de octeți citiți (întreg) – **DA**
     - malloc întoarce adresa de memorie alocată (pointer) – **NU**
     - fopen întoarce FILE * (un pointer) – **NU**
2. În ce situație modificarea cursorului de fișier pentru un descriptor conduce la modificarea cursorului de fișier pentru alt descriptor?
   - **Răspuns**: în cazul în care unul dintre descriptori este un duplicat al altui descriptor; amândoi vor partaja descriptorul de fișier
3. Un descriptor de fișier pentru un proces dat poate referi între `X` și `Y` fișiere. Ce valori au `X` și `Y`?
   - **Răspuns**: X = 0 în cazul în care descriptorul este nevalid/nealocat; Y = 1 – un descriptor de fișier referă un singur fișier; nu poate să refere mai multe fișiere
4. Listați secvența de pseudocod prin care scrierea la descriptorul 1 al unui proces să realizeze afișarea la stderr iar scrierea la descriptorul 2 să realizeze afișarea la stdout.
   - **Răspuns**:

```
dup2(1, 3); /* descriptorul 3 indica stdout (salvare descriptor)*/

dup2(2, 1); /* descriptorul 1 indica stderr */
```

```
dup2(3, 2); /* descriptorul 2 indica stdout */
```

PROCESE & PLANIFICAREA EXECUTIEI

1. Apelul `wait()` este un apel blocant. Când are loc deblocarea procesului blocat în `wait()`?
   - **Răspuns**: Un proces este deblocat din apelul `wait()` atunci când unul dintre procesele sale copil își încheie execuția. În acel moment, apelul `wait()` se deblochează și întoarce informații despre modul în care și-a încheiat procesul copil execuția.
2. De ce spunem despre apelul `fork()` că este invocat o dată dar se întoarce de două ori?
   - **Răspuns**: Apelul *fork()* este invocat o dată de procesul părinte și se întoarce de două ori: o dată în procesul părinte pentru continuarea execuției acestuia și altă dată în procesul copil de unde va rula acesta.
3. Ce este un proces zombie?
   - **Răspuns**: Un proces zombie este un proces care și-a încheiat execuția dar care nu a fost încă așteptat de procesul său părinte.
4. Dați exemplu de situație care duce la trecerea unui proces din starea `RUNNING` în starea `READY`.
   - **Răspuns**: Un proces trece din starea `RUNNING` în starea `READY` atunci când îi expiră cuanta de rulare sau când există un proces cu prioritate mai mare în coada `READY` (care să îi ia locul).
5. Numiți o sursă de overhead care apare atunci când sistemul de operare schimbă contextul de execuție între două procese.
   - **Răspuns**: Surse de overhead pentru schimbarea de context între procese sunt schimbarea tabelei de pagini, care conduce la flush la TLB, algoritmul de alegere a următorului proces și schimbarea efectivă de context, cu salvarea registrelor procesului curent și restaurarea procesului ales.
6. Descrieți o problemă posibilă care poate apărea dacă un sistem de operare implementează un algoritm de planificare de tipul *Shortest Job First*.
   - **Răspuns**: În cazul unei planificări *Shortest Job First*, dacă sunt adăugate în sistem, în mod constant, procese noi și de durată scurtă, procesele de durată mai lungă nu vor apuca să ruleze. Va rezulta într-un timp de așteptare foarte mare pentru procesele de lungă durată sau chiar în `starvation` (așteptare nedefinită pentru ca un proces să poată rula pe procesor).
7. La ce se referă noțiunea de timp de așteptare (*waiting time*) în contextul planificării proceselor (*process scheduling*)?
   - **Răspuns**: Noțiunea de *waiting time* se referă la timpul de așteptare al unui proces în coada`READY` a planificatorului. Pentru un sistem interactiv/responsiv este de dorit ca timpul de așteptare să fie cât mai scurt.
8. Care este un avantaj și un dezavantaj al folosirii unei cuante de timp scurte în planificarea proceselor (*process scheduling*)?
   - **Răspuns**: Folosirea unei cuante de timp scurte înseamnă un sistem interactiv și responsiv. Dar înseamnă și schimbări dese de context adică un randament mai scăzut al sistemului în a rula procese, deci o productivitate (*throughput*) redusă.
9. În ce situație are loc tranziția din starea WAITING în starea READY a unui proces?
   - **Răspuns**: În cazul în care operația care a cauzat așteptarea (de exemplu, citirea de pe disc) s-a încheiat și acum procesul poate rula.
10. În ce situație are loc tranziția din starea RUNNING în starea WAITING a unui proces?
    - **Răspuns**: În momentul în care procesul execută o operație blocantă (operație de I/O, sleep), acesta trece din starea RUNNING în starea WAITING.
11. Ce este o schimbare de context? De ce este necesară?
    - **Răspuns**: Se referă la schimbarea unui proces care rulează pe un procesor (este în starea RUNNING) cu un alt proces (aflat în starea READY). Este necesară pentru a asigura folosirea optimă a procesorului (dacă un proces se blochează îi ia altul locul) și pentru asigurarea echității (fairness) a sistemului (procesele se schimbă cu altele pentru a permite câtor mai multe să ruleze în sistem).
12. Ce reprezintă spațiul de adrese al unui proces? De ce este util?
    - **Răspuns**: Spațiul de adrese al unui proces este spațiul de lucru cu memoria a unui proces. Procesul lucrează cu adrese de memorie iar spațiul de adresă îi definește zonele accesibile. Spațiul de adresă asigură separația, la nivelul memoriei, între un proces și alt proces
13. De ce sistemele cu planificare preemptivă au un nivel de interactivitate mai bun decât sistemele cu planificare cooperativă?
    - **Răspuns**: Planificarea preemptivă introduce noțiunea de cuantă de timp alocată unui proces. Când acestuia îi expiră cuanta, este preemptat și înlocuit pe procesor. În acest fel, fiecare proces va ajunge mai repede pe procesor; nu apare riscul ca un proces să ruleze mult timp pe procesor. Fiecare proces rulând destul de rapid pe procesor, vom avea un sistem mai responsiv și mai interactiv.
14. Dați exemplu de situație în care un proces este scos de pe procesor deși NU a efectuat o operație blocantă.
    - **Răspuns**: În momentul în care unui proces îi expiră cuanta, aceasta este scos de pe procesor și un alt proces aflat în starea READY este planificat. Același lucru se întâmplă dacă există un proces în coada READY cu prioriate mai bună decât cel ce rulează pe procesor.
15. De ce, în general, procesele I/O bound au prioritate mai bună decât procesele CPU bound?
    - **Răspuns**: În general, procesele I/O bound vor executa o operație de I/O rapid, adică se vor bloca. În acest caz, aceste procese vor trece în starea WAITING și vor elibera procesorul unui alt proces. Acordându-le prioritate mai bună, acestea vor rula mai repede dar vor elibera rapid procesorul lăsând loc altor procese. Un proces CPU bound va elibera mai târziu procesorul, motiv pentru care va avea o prioritate mai puțin bună. Preferăm să planificăm procesele I/O bound.
16. De ce sistemele care doresc productivitate ridicată au alocată o cuantă de timp mai mare alocată fiecărui proces?
    - **Răspuns**: Un sistem este productiv dacă cea mai mare parte din timp acesta execută acțiune utilă. Pentru aceasta trebuie ca procesele să ruleze cât mai mult timp și să existe cât mai puține schimbări de context. Prea multe schimbări de context înseamnă un overhead semnificativ asupra timpului util de lucru. De aceea, pentru a diminua numărul de schimbări de context un sistem productiv va aloca o cuantă de timp mare proceselor sale, procesele petrecând cât mai mult timp rulând.
17. Fie P0 procesul părinte al procesului P1, T0 momentul de timp la care P0 execută apelul wait() și T1 momentul de timp la care P1 execută apelul exit(). În ce stare vor fi cele două procese în intervalul (T0, T1) dacă T0<T1?
    - **Răspuns** Procesul P0 este în starea WAITING, în așteptarea semnalului de la copil. Procesul P1 poate fi în orice stare, în funcție de codul său, dar va trece, cu siguranță, prin starea RUNNING pentru a putea executa apelul exit().
18. Prezentați un avantaj al mapării spațiului de memorie al kernel-ului în spațiul de adresă al fiecărui proces.
    - **Răspuns** Prin maparea spațiului de memorie al kernel-ului în spațiul de adresă al fiecărui proces se evită schimbarea de context la fiecare apel de sistem, inclusiv apelul schedule().
19. Fie P0 procesul părinte al procesului P1, T0 momentul de timp la care P0 execută apelul wait() și T1 momentul de timp la care P1 execută apelul exit(). În ce stare vor fi cele două procese în intervalul (T1, T0) dacă T1<T0?
    - **Răspuns** Procesul P0 poate fi în orice stare, în funcție de codul său, dar va trece, cu siguranță, prin starea RUNNING pentru a putea executa apelul

wait(). Procesul P1 este în starea TERMINATED (zombie), deoarece și-a încheiat execuția și așteaptă să îi fie citită valoarea de ieșire de către părinte.

20. De ce un proces orfan nu poate deveni zombie?
    - **Răspuns** Deoarece un proces orfan este adoptat imediat de init, este imposibil ca el să devină zombie. Acesta execută wait pentru fiecare proces copil al său, care și-a încheiat execuția, împiedicând ca acesta să devină zombie.
21. Fie P un proces zombie. Ce procese îl pot elimina din sistem prin apelul wait()?
    - **Răspuns** Procesele care pot elimina din sistem un proces zombie prin apelul wait() sunt: părintele său (dacă nu și-a încheiat execuția) și procesul init (care adoptă procesele orfane și execută wait pentru fiecare proces copil al său, care și-a încheiat execuția.
22. În urma unui apel fork() pot rezulta între X și Y procese **noi**. Ce valori au X și Y?
    - **Răspuns** Dacă apelul fork() eșuează nu va fi creat niciun proces nou. Dacă apelul se execută cu succes, va fi creat un proces nou, copil al procesului care a executat fork(). Astfel, pot rezulta între 0 și 1 procese **noi**. X=0. Y=1.
23. Dați două exemple de resurse care pot aparține unui proces, dar nu pot aparține unui program.
    - **Răspuns** Procesul reprezintă o instanță activă activă a unui program. Resursele care pot aparține unui proces, dar nu pot aparține unui program sunt: memoria, CPU-ul, PCB-ul (PID, spațiul de adresă - zonele de date, cod, heap, stivă, tabela de descriptori, masca de semnale, etc.).
24. De ce un planificator echitabil (fair) nu este, în general, productiv (nu oferă un throughput mare)?
    - **Răspuns** Un planificator echitabil implică schimbări de context dese, astfel că este petrecut un timp relativ mare cu schimbările de context, scăzând productivitatea.
25. În ce situație este posibil ca un proces să treacă **direct** din starea WAITING în starea TERMINATED?
    - **Răspuns** Un proces va trece direct din starea WAITING în starea TERMINATED dacă primește un semnal care nu poate fi ignorat sau suprascris, precum SIGKILL sau SIGQUIT, care conduce la terminarea procesului, indiferent de context.
26. De ce nu mai este folosită planificarea cooperativă în sistemele desktop moderne?
    - **Răspuns** În sistemele desktop moderne accentul este pus pe interactivitate. Planificarea cooperativă se bazează pe cedarea voluntară a procesorului și nu poate oferi interactivitate.
27. De ce, pe un sistem desktop, de obicei, sunt mai multe procese în starea WAITING decât în starea READY?
    - **Răspuns** Pe un sistem desktop, majoritatea proceselor așteaptă inputul utilizatorului, deci sunt procese I/O bound.
28. Fie procesul P1. Câte procese copil va avea P1 în urma rulării secvenței de pseudocod de mai jos? De ce?

```
while(fork() == 0)
    ;
```

    - **Răspuns**: Apelul `fork` întoarce 0 în procesul copil și PID-ul procesului copil, diferit de 0, în părinte. De aceea, părintele va executa o singură dată `fork`, ieșind apoi din buclă. Copilul va mai executa o dată bucla, devenind la rândul lui părinte și ieșind din buclă. Fiecare proces va fi părinte pentru un singur copil. Astfel, P1 va avea un singur proces copil.
30. În ce zonă de memorie este plasată adresa de retur a unei funcții? De ce?
    - **Răspuns**: Adresa de retur a unei funcții este plasată pe stivă. La fiecare apel de funcție, un nou stack frame este creat pe stivă, care conține parametrii funcției, adresa de retur și variabile locale. La ieșirea din funcție, adresa de retur este preluată de pe stivă.
31. Dați două exemple în care:
    - a) un proces are mai multe procese părinte decât procese copil;
    - b) un proces are mai multe procese copil decât procese părinte.
    - **Răspuns**:
    - toate procesele, cu excepția procesului `init`, au un proces părinte;
        1. a) un proces care nu are procese copil (nu a executat `fork` niciodată);
        2. b) un proces care are cel puțin două procese copil (a executat `fork` de cel puțin două ori, cu succes) sau procesul `init` care nu are proces părinte.
32. De ce apelul `fork` întoarce 0 pentru succes în procesul copil și o valoare diferită de 0 pentru succes în procesul părinte?
    - **Răspuns**: În procesul copil întoarce 0 pentru că procesul poate folosi apelul `getppid` pentru a afla PID-ul procesul părinte. În procesul părinte, `fork` întoarce PID-ul procesului copil. Un proces poate avea mai multe procese copil și este comod ca `fork` să întoarcă PID-ul procesului copil proaspăt creat.
33. Care dintre următoarele apeluri pot modifica numărul de procese dintr-un sistem UNIX: `open`, `fork`,`dup2`, `wait`, `exec`?
    - **Răspuns**:
        1. `open` deschide și, posibil, creează un nou fișier sau dispozitiv - **NU**
        2. `fork` creează un nou proces - **DA**
        3. `dup2` duplică un descriptor de fișier - **NU**
        4. `wait` așteaptă schimbarea stării unui proces, în cazul unui proces copil care și-a încheiat execuția, permite sistemului să elibereze resursele asociate cu copilul, ducând la scăderea numărului de procese din sistem - **DA**
        5. `exec` înlocuiește imaginea procesului curent cu o nouă imagine de proces - **NU**
34. Dați exemplu de acțiune ce conduce la trecerea unui proces din starea `RUNNING` în `WAITING` și un exemplu de acțiune care conduce la o trecere inversă.
    - **Răspuns**
        1. Un proces trece din starea `RUNNING` în `WAITING` atunci când execută o acțiune blocantă.
        2. Un proces nu poate trece direct din starea `WAITING` în `RUNNING`. Procesul trece în starea `READY`atunci când a dispărut cauza blocării sale.
35. Cum ar trebui aleasă cuanta de timp pentru procese pentru un sistem în care se dorește interactivitate mare? De ce?
    - **Răspuns**: Cuanta de timp trebuie să fie mică pentru ca procesele să fie preemptate mai des și să crească timpul de răspuns al sistemului.
36. În ce mod este determinat numărul de procese care se pot afla, la un moment dat, în starea `RUNNING`de următoarele componente fizice ale sistemului: număr de procesoare, arhitectură pe 32/64 biți, memorie fizică, capacitate de stocare?
    - **Răspuns**: Numărul de procese care se pot afla, la un moment dat, în starea `RUNNING` depinde doar de numărul de procesoare, deoarece pot exista maxim *n* procese în starea `RUNNING` pe un sistem cu *n* procesoare. Celelalte componente nu influențează acest număr.
37. În ce mod este determinat numărul de procese care se pot afla, la un moment dat, în starea `READY`de următoarele componente fizice ale sistemului: număr de procesoare, arhitectură pe 32/64 biți, memorie fizică, capacitate de stocare?
    - **Răspuns**: Numărul de procese care se pot afla, la un moment dat, în starea `READY` depinde de memoria fizică, deoarece coada/cozile `READY` sunt reținute în liste în memoria fizică, iar dimensiunea acesteia poate limita dimensiunile acestora. Celelalte componente nu influențează, în general, acest număr.
        1. Observație: Au fost considerate valide următoarele argumente:
            1. pentru dimensiuni mari ale memoriei fizice (peste 4 GB) este relevantă arhitectura, pentru a putea adresa memoria disponibilă;
            2. existența mai multor procesoare duce la creșterea numărului de procese aflate simultan în starea `RUNNING`, astfel că scade numărul de procese care ar fi fost în starea `READY`, diferența fiind egală cu diferența dintre numărul de procesoare;
38. Care dintre următoarele apeluri au legătură directă cu procesele **zombie**: `fork`, `dup`, `wait`, `exec`,`malloc`?
    - **Răspuns**: Un proces zombie este un proces care a murit dar care nu a fost așteptat (wait) de procesul părinte; informațiile legate de încheiere ocupă spațiu în memorie până la un apel`wait`/`waitpid`/`WaitForSingleObject` al procesului părinte. Apelul `wait` are, așadar, legătură directă cu procesele zombie.
39. Ce valoare întoarce `fork` în procesul părinte? Dar în procesul copil? De ce?
    - **Răspuns**: În procesul copil întoarce 0 pentru că procesul poate folosi apelul `getppid` pentru a afla PID-ul procesul părinte. În procesul părinte, `fork` întoarce PID-ul procesului copil. Un proces poate avea mai multe procese copil și este comod ca `fork` să întoarcă PID-ul procesului copil proaspăt creat.
40. Un proces poate avea între `A` și `B` procese copil și între `C` și `D` procese părinte. Ce valori au (`A`, `B`) respectiv (`C`, `D`)?
    - **Răspuns**:
        1. (A, B) = (0, inf) - un proces poate avea oricâte procese copil, în limita resurselor disponibile
        2. (C, D) = (1, 1) - orice proces copil are un proces părinte, se poate considera init ca proces fără proces părinte; se permite soluția (C, D) = (0, 1) (dacă se precizează init)
41. Câte procese se pot găsi, la un moment dat, într-un sistem de operare, în stările `RUNNING`, `READY` și`WAITING`?
    - **Răspuns**:
        1. În starea `RUNNING` se găsesc procesele care execută cod pe procesor. Numărul maxim de procese în acea stare este dat de numărul de procesoare.
        2. În starea `READY`, respectiv `WAITING` se pot găsi oricâte procese în limita resurselor sistemului. În`READY` se vor găsi procese gata de rulare, neblocate, care așteaptă acordarea de timp pe procesor; în

starea `WAITING` se vor găsi procese blocate în așteptarea unei acțiuni de blocare (dispozitiv de I/O, semafoare cozi de mesaje). Nu există o limitare dată pentru procesele din starea `READY` sau `WAITING`.

42. Pentru un sistem se dorește productivitate (throughput) mare. De ce se preferă alegerea unei cuante de timp (timeslice) mare pentru procese?
    - **Răspuns**:
        1. Productivitate mare înseamnă un timp mare consumat pe lucru efectiv (rulare pe procesor) pe lângă timpi consumați pentru alte activități, cea mai importantă fiind schimabrea de context.
        2. În cazul unei cuante de timp mari, proceselor vor lucra timp îndelungat iar timpul consumat pe schimbarea de context (apărută la expirarea cuantei) va fi mai mic, relativ.
43. Dați exemplu de acțiune ce conduce la trecerea unui proces din starea `READY` în `RUNNING` și un exemplu care conduce la o trecere inversă.
    - **Răspuns**:
        1. Un proces trece din starea `RUNNING` în `READY` în momentul în care i-a expirat cuanta de timp sau când există un alt proces prioritar lui în coada `READY`.
            1. Observație: Blocarea procesului conduce la trecerea acestuia în coada `WAITING`, iar omorârea procesului înseamnă că acesta nu trece în nici o altă coadă.
        2. Un proces trece din `READY` în `RUNNING` în momentul în care se găsește în vârful cozii `READY` și procesului care rulează pe procesor (cel din starea `RUNNING`) îi expiră cuanta, sau are prioritate mai mică, sau acel proces se blochează sau acel proces moare.
44. Dați exemplu de acțiune ce conduce la trecerea unui proces din starea `READY` în `WAITING` și un exemplu care conduce la o trecere inversă.
    - **Răspuns**:
        1. Un proces nu poate trece direct din starea `READY` în `WAITING`. Pentru a ajunge în `WAITING` trebuie să execute o acțiune blocantă, adică trebuie să ruleze, adică trebuie să se găsească în stareaRUNNING.
        2. Un proces trece din starea `WAITING` în `READY` atunci când a dispărut cauza blocării sale: a apărut un eveniment de I/O pe care îl aștepta, a fost notificat, a fost făcut unlock pe mutex-ul la care aștepta etc. Procesul este, atunci, pregătit pentru execuție.

MEMORIA VIRTUALA

1. De ce este uzual și avantajos ca spațiul virtual de adrese al proceselor să cuprindă o zonă dedicată pentru kernel?
    - **Răspuns**: Prezența zonei dedicate pentru kernel în spațiul de adresă al fiecărui proces înseamnă că la fiecare apel de sistem, adică la trecerea din user space în kernel space, tabela de pagini rămâne aceeași și nu se face flush la TLB. În cazul în care kernel-ul ar avea o zonă dedicată, atunci ar avea și o tabelă de pagini dedicată și ar trebui schimbată tabela de pagini la fiecare apel de sistem și la fiecare revenire din apel de sistem.
2. Procesul P1 folosește `100MB` de memorie fizică (RAM) rezidentă. P1 execută `fork()` și rezultă procesul P2. Câtă memorie fizică (RAM) rezidentă folosesc împreună P1 și P2 imediat dupa`fork()`? De ce?
    - **Răspuns**: Apelul `fork()` folosește *copy-on-write* ceea ce înseamnă că nu se alocă memorie rezidentă nouă pentru noul proces. Se alocă, într-adevăr, o nouă tabelă de pagini, dar spațiul rezident al procesului P1 este acum partajat cu procesul P2 până la prima operație de scriere, când pagina aferentă va fi duplicată.
3. Care este utilitatea conceptului de *demand paging*?
    - **Răspuns**: Atunci când sistemul de operare folosește *demand paging* alocarea de memorie fizică este amânată până în momentul în care nevoie (adică la primul acces). Sistemul de operare doar rezervă memorie virtuală și nu alocă memorie fizică în spate, economisind memorie fizică. La primul acces se alocă și memorie fizică, la cerere (adică *on demand*) și se face maparea acesteia la spațiul virtual (*paging*).
4. De ce zonele `.text` și `.rodata` din cadrul bibliotecilor partajate (`shared libraries`) pot fi partajate între mai multe procese?
    - **Răspuns**: Zonele `.text` și `.rodata` sunt zone *read only*. Acest lucru înseamna că pot fi partajate în siguranță pentru că nici un proces care accesează zona nu o va putea modifica. Zonele conțin permanent aceleași informații indiferent de numărul de procese care le folosesc și pot fi, deci, partajate.
5. De ce este utilă paginarea ierarhică?
    - **Răspuns**: Dacă nu am folosi paginare ierarhică tabelele de pagini ar ocupa foarte mult spațiu; ar fi neovie de o intrare pentru fiecare pagină virtuală a unui proces. Paginarea ierarhică conduce la reducerea spațiului ocupat de tabela de pagini, profitând de faptul că o bună parte din spațiul virtual de adrese al procesului nu este folosit.
6. Care este o cauză sursă pentru evacuarea unei pagini din memoria fizică (RAM) pe disc (*swap out*)?
    - **Răspuns**: Cauze sursă pentru evacuarea unei pagini din RAM sunt:
        1. operația de *swap in*, care necesită o pagină liberă în RAM, conducând la o operația de *swap out*
        2. alocarea unei pagini fizice noi; nu există pagini libere, se execută *swap out*
        3. *demand paging*, la fel ca mai sus
        4. *copy on write* care necesită alocarea unei noi pagini fizice, posibil inexistente
7. Ce conține tabela de pagini a unui proces?
    - **Răspuns**: Tabela de pagini a unui proces conține pointeri de pagini fizice. Indexul în tabelă este pagina virtuală. În general tabela de pagini mai conține și informații legate de permisiuni, validatate, dacă pagina a fost sau nu modificată.
8. Dați exemplu de situație care cauzează *page fault* **fără** a rezulta în trimiterea unei excepții de acces la memorie (de tipul `SIGSEGV`, *Segmentation fault*) către procesul care a generat *page fault*-ul.
    - **Răspuns**: Dacă un proces are rezervat un spațiu virtual în modul *demand paging* atunci accesarea unei pagini virtuale din acel spațiu va conduce la *page fault*. În urma *page fault*-ului, se va aloca și mapa o pagină fizică, iar procesul își va continua execuția. Nu va fi generată excepție de acces la memorie. La fel se întâmplă și în cazul *copy-on-write*.
9. De ce NU avem fragmentare externă în cazul folosirii paginării?
    - **Răspuns**: Întreg spațiul fizic este împărțit în pagini de dimensiune fixă. Atunci când este nevoie de spațiu nou se alocă pagini noi indiferent de poziția lor în spațiul inițial. Dacă o pagină este liberă, este eligibilă pentru alocare. Nu ajungem să avem fragmentare externă, adică spațiu liber nealocabil între spații deja alocate.
10. Ce reprezintă "demand paging"? Ce rol are?
    - **Răspuns**: Demand paging este o formă de amânare a alocării de pagini fizice până în momentul în care este nevoie. Prin demand paging se alocă doar pagini virtuale iar paginile fizice aferente se vor aloca în momentul accesului la acele pagini. Rolul său este de a eficientiza consumul de memorie și de timpul de alocare. În momentul alocării se alocă doar memorie virtuală, nu și fizică, lucru care durează mai puțin. De asemenea, consumul de memorie fizică (RAM) la un moment dat este redus la strictul necesar în acel moment.

11. Precizați un avantaj al folosirii mecanismului de memorie virtuală.
    - **Răspuns**: Un avantaj este faptul că spațiul (virtual) de adrese al unui proces este continuu, independent de forma în care este realizată maparea pe spațiul fizic. Orice alocare se face în continuare spațiului virtual existent, iar mecanismul de memorie virtuală face maparea cu spațiul fizic. Un alt avantaj este posibilitatea folosirea spațiului de swap: mod prin care putem folosi discul pentru a reține pagini care nu încap în spațiul fizic (memoria RAM). Un alt avantaj este posibilitatea partajării memoriei, pagini virtuale din procese diferite (sau chiar din același proces) putând fi mapate peste aceleași pagini fizice.
12. Ce este spațiul de swap? Ce rol are?
    - **Răspuns**: Spațiul de swap este spațiul localizat pe disc folosit ca depozitar temporar al informațiilor din memorie RAM. În momentul în care spațiul fizic (memoria RAM) devine insuficient, se evacuează (swap out) anumite pagini fizice. În momentul în care aceste pagini sunt necesare sunt readuse în memoria RAM (swap in).
13. Care este avantajul folosirii paginării ierarhice?
    - **Răspuns**: Prin folosirea paginării ierarhice, spațiul ocupat de tabelele de pagini ale proceselor este diminuat. În loc să existe o intrare pentru fiecare pagină, vor exista intrări doar pentru paginile valide din spațiul virtual de adrese al proceselor.
14. Corespondența între pagini virtuale și pagini fizice este "mai multe la una". De ce se întâmplă și la ce este util acest lucru?
    - **Răspuns**: Tabela de pagini conține o mapare între pagini virtuale și pagini fizice. În acest fel mai multe pagini virtuale pot avea corespondent aceeași pagină fizică. Acest lucru este util pentru mecanismul de memorie partajată în care procese diferite au pagini virtuale din propriu spațiu de adresă mapate peste aceleași pagini fizice.
15. Ce este TLB? Ce rol are?
    - **Răspuns**: TLB (Translation Lookaside Buffer) este o memorie cache la nivelul sistemului care cache-uiește intrările din tabelele de pagini ale proceselor. Întrucât fiecare acces la memorie necesită de fapt două accese (unul la tabela de pagini, alta la datele efective), TLB-ul micșorează timpul de acces simplificând primul acces (la tabela de pagini). TLB îndeplinește astfel rolul eficientizării accesului la memorie.
16. Care este avantajul principal al folosirii mecanismului copy-on-write la crearea proceselor folosind fork()?
    - **Răspuns**: Prin folosirea mecanismului de copy-on-write la crearea proceselor folosind fork(), un proces nou este creat foarte rapid. Un proces nou nu va trebui să aloce spațiu fizic nou/separat ci va partaja spațiul fizic aferent procesului părinte. Acel spațiu este marcat read-only și va fi duplicat doar în momentul în care unul dintre procese va scrie (copy-on-write).
17. De ce numărul de pagini virtuale dintr-un sistem este mai mare decât numărul de pagini fizice?
    - **Răspuns** Numărul de pagini fizice este limitat de dimensiunea memoriei RAM, în timp ce numărul de pagini virtuale este determinat de numărul de procese. În cazul memoriei partajate de două sau mai multe procese, pot exista mai multe pagini virtuale mapate pe aceeași pagină fizică.
18. De ce paginarea ierarhică are un overhead de prelucrare mai mare decât paginarea neierarhică?
    - **Răspuns** În cazul paginării neierarhice, numărul paginii virtuale este și indexul în tabela de pagini, deci va exista un singur acces la memorie pentru aflarea paginii fizice. În cazul paginării ierarhice se vor face atâtea accese la memorie, cât numărul de niveluri ierarhice.
19. Care este principală sursă de overhead la schimbarea de context între două procese?
    - **Răspuns** Principală sursă de overhead la schimbarea de context între două procese este repopularea TLB-ului. În urma unei schimbări de context, intrările din TLB sunt invalidate și este necesară translatarea adreselor virtuale în adrese fizice pe baza tabelei de pagini a noului proces. Acest lucru implică numeroase accesări ale memoriei, la o viteză mult mai mică decât viteza TLB-ului.
20. De ce nu este necesară eliminarea paginilor de memorie ale kernel-ului din TLB în cazul unei schimbări de context?
    - **Răspuns** Eliminarea paginilor de memorie ale kernel-ului din TLB în cazul unei schimbări de context nu este necesară deoarece toate procesele au mapate pe aceleași pagini virtuale tot spațiul kernel.
21. Fie un sistem cu paginare ierarhică pe două niveluri, fără TLB. Pot două pagini de memorie virtuală referi aceeași pagină de memorie fizică?
    - **Răspuns** Da, două pagini de memorie virtuală pot referi aceeași pagină.
    - În această situație, în cadrul unui proces, în tabela de pagini, unor intrări diferite (indexate de pagina virtuală A și pagina virtuală B) le corespunde aceeași pagină fizică.
    - În cazul a două procese este vorba de implementarea mecanismului de shared memory.
    - Afirmațiile sunt valabile pentru orice tip de paginare: ierarhică, neierarhică, inversată, indiferent de prezența / absența TLB-ului.
22. Presupunând dimensiunea unei pagini de 4096, câte pagini fizice (frame-uri) noi vor fi alocate în urma apelului malloc(6000)? De ce?
    - **Răspuns** Apelul malloc folosește demand paging, alocând memorie pur virtuală, fără suport în memoria fizică. Totuși, pentru alocări de dimensiuni mici, este posibil să fie alocate și paginile fizice aferente. Deoarece 4096 < 6000 < 2 * 4096, se vor aloca maxim două pagini.
23. De ce este mecanismul de mapare a fișierelor esențial în rularea proceselor pe sisteme de operare moderne?
    - **Răspuns** Mecanismul de mapare a fișierelor este esențial deoarece rularea proceselor se face prin maparea executabilului în memorie și folosirea demand-paging pentru încărcarea zonelor de date și cod la nevoie.
24. Fie afirmația "Toate procesele vor genera page fault-uri în urma unui apel din familia exec()." Precizați și justificați valoarea de adevăr a afirmației.
    - **Răspuns** Afirmația este adevărată deoarece în urma apelului exec(), vor fi încărcate zonele de date și cod, folosind demand-paging.
25. Fie afirmația: "Un apel fork() modifică numărul de pagini virtuale și numărul de pagini fizice alocate într-un sistem." Precizați și justificați valoarea de adevăr a afirmației.
    - **Răspuns** Afirmația este adevărată. Numărul paginilor virtuale crește, deoarece apare un nou spațiu de adresă. Crește și numărul de pagini fizice, întrucât se alocă structuri interne nucleului, printre care noua tabelă de pagini pentru procesul copil.
26. În ce mod influențează dimensiunea TLB-ului numărul maxim de spații de adresă existente în sistem?
    - **Răspuns**: TLB-ul menține mapări de pagini fizice și pagini virtuale. Conține un subset al tabelei de pagini. Numărul maxim de spații de adresă existente în sistem depinde de numărul de procese existente în sistem. Acest număr nu este influențat de dimensiunea TLB-ului.
27. Fie un utilizator fără drepturi de administrator. În ce mod poate acesta modifica numărul de pagini fizice din sistem? Dar numărul de pagini virtuale?
    - **Răspuns**:
        1. Numărul de pagini fizice din sistem poate fi modificat prin adăugarea/scoaterea de memorie fizică din sistem sau prin modificarea dimensiunii paginilor fizice, având suport hardware. Pentru aceste operații este necesar un administrator, deci un utilizator fără drepturi de administrator nu poate modifica numărul de pagini fizice din sistem.
        2. Un utilizator fără drepturi de administrator poate modifica numărul de pagini virtuale din sistem prin crearea/omorârea de procese, deoarece fiecare proces ale paginile virtuale proprii.
28. Dați exemplu de avantaj, respectiv dezavantaj al paginării simple (neierarhice) în fața paginării ierarhice.
    - **Răspuns**:
        1. Avantaje:
            1. Overhead de prelucrare mic (este parcursă o singură tabelă de pagini);
            2. Complexitate mică de implementare.
        2. Dezavantaj: paginarea simplă ocupă mai mult spațiu, deoarece se ocupă spațiu și pentru zonele de memorie virtuale nevalide.
29. Dați exemplu de avantaj, respectiv dezavantaj al folosirii TLB.
    - **Răspuns**:
        1. Avantaj: TLB-ul este un cache rapid care menține mapări de pagini fizice și pagini virtuale. În cazul TLB hit, scade timpul de acces la memorie.
        2. Dezavantaje:
            1. În cazul TLB miss, timpul de acces la memorie este mai mare decât timpul de acces în absența TLB-ului.
            2. La fiecare schimbare de context se face TLB flush, astfel că vor exista mai multe miss-uri imediat după schimbarea de context.
            3. Dimensiune mică și cost ridicat.
30. Motivați utilizarea *copy-on-write* în cadrul apelului `fork()`.
    - **Răspuns**: Este utilă folosirea mecanismului de *copy-on-write* deoarece crearea unui nou proces se face mai rapid, evitând operațiile de copiere a memoriei. Foarte probabil, după `fork()` copilul va face un apel `exec()` pentru a lansa un program diferit. Astfel, dacă nu s-ar folosi *copy-on-write*, s-ar copia degeaba paginile din spațiul de adresă al părintelui.
31. Prezentați un avantaj și un dezavantaj al mapării fișierelor în memorie.
    - **Răspuns**:
        1. Avantaje:
            1. Se evită apelurile de sistem `read()` și `write()`, deci și *double-buffering-ul*.

2. Permite partajarea fișierelor între procese.
3. Nu mai este necesar apelul `lseek()`, căutarea se realizează prin manipularea pointer-ilor.

2. Dezavantaje:
1. Maparea fișierelor în memorie se face la nivel de pagină, astfel că apare fragmentare, mai ales în cazul fișierelor mici.
2. Maparea trebuie să se încadreze în spațiul de adresă al procesului. Pe sisteme pe 32 de biți, sunt dificil de mapat fișiere mari.
3. Accesul la memorie determină apariția page fault-urilor.
4. Trebuie apelat periodic `msync()`, deoarece modificările în memorie nu sunt scrise imediat pe disc.

32. În ce situație folosirea funcției `memcpy` generează *page fault* și în ce situație nu generează *page fault*?
- **Răspuns**: Funcția `memcpy` folosește două buffere: sursă și destinație. Fie `S` mulțimea paginilor în care se află bufferul sursă și `D` mulțimea paginilor în care se află bufferul destinație.
1. Folosirea funcției `memcpy` generează *page fault* în următoarele situații:
1. Cel puțin o pagină din `S` sau `D` este nevalidă. (nu a fost alocată în RAM, este pe swap sau un acces nevalid din partea programatorului)
2. Cel puțin o pagină din `D` este validă, dar nu sunt drepturi de scriere. (copy-on-write, mapare PROT_READ)
3. Cel puțin o pagină din `S` este validă, dar nu sunt drepturi de citire. (mapare PROT_NONE)
2. Folosirea funcției `memcpy` nu generează *page fault* dacă toate paginile din `S` sunt valide, cu drept de citire și dacă toate paginile din `D` sunt valide, cu drept de scriere.

33. Pe un sistem pe 32 de biți, cu 512MB RAM și 512MB de swap, un proces execută secvența următoare de cod:

```
int x = read_int_from_user();
void *a = malloc(x);

memset(a, 0, x);
```

La un curs de Sisteme de Operare, profesorul întreabă "Care este valoarea minimă a lui `x` pentru care apelul `malloc` întoarce un pointer valid, dar apelul `memset` duce la blocarea sistemului?". Un student răspunde "1GB + 1B". Profesorul răspunde "Cam pe acolo, dar valoarea reală este ceva mai mică". De ce a spus profesorul acest lucru?
- **Răspuns**: Apelul `malloc` folosește demand paging, alocând memorie pur virtuală, fără suport în memoria fizică. Apelul `memset` necesită alocarea de pagini fizice, astfel că sistemul se va bloca în momentul în care nu mai are pagini fizice disponibile. Sistemul are disponibil 1 GB memorie fizică (512MB RAM și 512MB de swap) astfel că studentul a considerat că se va bloca la ocuparea întregii memorii fizice (1GB + 1B). În realitate, sistemul se va bloca la o valoare "ceva mai mică", deoarece în memoria fizică se află pagini de memorie ale kernelului, cât și ale altor procese.

36. Dați exemplu de avantaj, respectiv dezavantaj al paginării ierarhice în fața paginării simple (neierarhice).
- **Răspuns**:
1. Avantaj: paginarea ierarhică ocupă mai puțin spațiu; se ocupă spațiu doar pentru zonele de memorie virtuale valide. În cazul paginării simple, tabela de pagini ocupă același spațiu indiferent de numărul de pagini virtuale valide.
2. Dezavataje:
1. overhead de prelucrare mai mare (trebuie parcurse mai multe tabele de pagini, mai multe referențieri);
2. complexitate mai mare de implementare (împărțire mai fină a unei adrese de memorie).

37. Fie un sistem cu paginare simplă (neierarhică). Pot două pagini de memorie virtuală referi aceeași pagină de memorie fizică? Dar invers?
- **Răspuns**:
1. Da, două pagini de memorie virtuală pot referi aceeași pagină.
1. În această situație, în cadrul unui proces, în tabela de pagini, unor intrări diferite (indexate de pagina virtuală A și pagina virtuală B) le corespunde aceeași pagină fizică.
2. În cazul a două procese este vorba de implementarea mecanismului de shared memory.
2. Nu, două pagini de memorie fizică nu pot referi aceeași pagină virtuală. Adresa virtuală este cea folosită de proces. O astfel de adresă nu poate indica spre două adrese fizice diferite. Este similar cu a spune că `f(X) = A` și `f(X) = B`, unde `A != B`.
3. Nu are relevanță folosirea paginării simple. Afirmațiile sunt valabile pentru orice tip de paginare: ierarhică, neierarhică, inversată.

38. Fie `struct tlb_entry` o structură aferentă unei intrări în TLB. Ce câmpuri conține o astfel de structură?
- **Răspuns**:
1. TLB-ul este folosit pentru translatarea rapidă a adreselor virtuale în adrese fizice. Structura conține adresa paginii virtuale și adresa paginii fizice aferente.

39. De ce este golit (flushed) TLB-ul la o schimbare de context?
- **Răspuns**:
1. TLB-ul conține un subset de intrări din tabela de pagini pentru acces rapid din partea procesorului. Tabela de pagini este proprie fiecărui proces (spațiului de adresă al acestuia). La o shimbare de context, procesul este schimbat și la fel și tabela de pagini. Schimbarea tabelei de pagini înseamnă invalidarea TLB-ului și este necesară golirea acestuia.

40. Dați exemplu de situație în care se produce *page fault* (la nivelul subsistemului de gestiune a memoriei), fără a rezulta în *segmentation fault*/excepție la nivelul procesului care a cauzat *page fault*-ul.
- În cazul *demand paging*, se alocă o pagină virtuală (*page*) fără suport fizic (*frame*). La apariția unui *page fault* se va aloca o pagină fizică, fără a rezulta o excepție.
- În cazul *copy-on-write*, două pagini virtuale (*page*) (din două procese diferite) sunt marcate read-only și referă aceeași pagină fizică (*frame*). În momentul în care unul dintre cele două procese efectuează o operație de scriere, se obține *page fault*, se duplică pagina fizică și se continuă execuția.
- În cazul unei pagini virtuale (*page*) valide, al cărei conținut se găsește în *swap*, un acces generează *page fault*. Conținutul este *swapped in* într-o pagină fizică (*frame*) si procesul își continuă execuția.

41. Care este rolul bitului *dirty* / *modified* pentru subsistemul de înlocuire a paginilor? Bitul este activat în momentul în care o pagină este modificată/scrisă.
- Dacă o pagină este modificată atunci o eventuală alegere a acestei pagini pentru a fi evacuată (*swapped out*) va înseamnă scrierea acesteia pe disc.
- Dacă o pagină nu este modificată (bitul *dirty* este dezactivat) nu va mai fi scrisă pe disc în momentul evacuării, rezultând într-un overhead redus al operației de înlocuire.
- În general, algoritmii de înlocuire de pagini din cadrul subsistemelor aferente, vor ține cont de ultimul acces (fie de scriere, fie de citire al unei pagini) – LRU, NRU (*Least/Not Recently Used*). Se vor prefera paginile nereferite recent, iar dintre acestea, cele care nu au fost scrise (bitul *dirty*dezactivat) – vezi NRU.

42. Dați exemplu de situație/scenariu în care apariția unui *page fault* determină *swap out*.
- În cazul apariției unui *page fault*, este posibil ca pagina fizică (*frame*) aferentă să nu fie alocată (*demand paging* sau *copy-on-write*) sau să fie pe disc (*swapped*). În acest caz pagina trebuie adusă în RAM.
- Dacă memoria fizică (RAM) este ocupată, va trebui aleasă o pagină fizică pentru a fi înlocuită. Se aplică un algoritm de înlocuire a paginii.
- În momentul aplicării algoritmului de înlocuire a paginii, pagina fizică este evacuată pe disc (*swapped out*); conținutul de pe disc aferent paginii ce a cauzat *page fault*-ul este adus în memorie (*swapped in*) în locul paginii fizice proaspăt evacuate.

SECURITATEA MEMORIEI

1. De ce este relevant, în contextul securității memoriei, faptul că adresa de retur a unei funcții se reține pe stivă?
   - **Răspuns**: Adresa de retur stocată în memorie oferă unui atacator posibilitatea suprascrierii acesteia și alterarea fluxului normal de execuție al programului. Pentru aceasta este nevoie de o vulnerabilitate într-un buffer la nivelul stivei. În general folosirea de adrese pe stive oferă această posibilitate si e de evitat, dar nu putem face asta în privința adresei de retur; este nevoie de stocarea pe stivă pentru a putea reveni în stack frame-ul anterior.
2. De ce folosirea DEP (*Data Execution Prevention*) **nu** previne atacurile de tipul *return-to-libc*?
   - **Răspuns**: DEP previne existența simultană a permisiunilor de scriere și execuție. Adică nu se poate scrie într-o zonă un shellcode (sau ceva similar) care apoi să se execute. Un atac de tipul *return-to-libc* presupune suprascrierea unei adrese (de retur, pointer de funcție) ca să pointeze către o funcție din biblioteca standard C. Întrucât un atac de tipul *return-to-libc* nu presupune scriere și execuție a aceleiași zone, nu poate fi prevenit de DEP.
3. De ce trebuie avut grijă la construcțiile precum cea de mai jos în cadrul unei funcții?

```
4.          int (*fn_ptr)(int, int);  /* fn_ptr is a function pointer */
```

```
          char buffer[128];    /* buffer for storing strings */
```

   - **Răspuns**: În construcția din exercițiu dacă nu se ține cont de dimensiunea buffer-ului se poate obține un *buffer overflow*. În urma overflow-ului, se suprascrie pointer-ul de funcție `fn ptr`. Probabil acest pointer va fi folosit la un moment dat rezultând în execuția arbitrară și alterând fluxul normal de execuție al programului.
5. De ce, în general, un shellcode se încheie cu invocarea apelului de sistem `execve`?
   - **Răspuns**: Un shellcode încearcă, în general, rularea unui program nou, de exemplu a unui shell în forma echivalentă a unui apel `execve("/bin/sh")`. Întrucât un apel de bibliotecă este mai dificil de realizat, se preferă o instrucțiune simplă de apel de sistem (precum `int 0x80`. Se face un apel de sistem `execve` cu un argument de forma unui șir `/bin/sh` într-un registru rezultând în crearea unui shell nou.
6. Care este utilitatea ASLR (Address Space Layout Randomization) din perspectiva securității memoriei?
   - **Răspuns**: În cazul unui atac ce exploatează o vulnerabilitate de securitate a memoriei, atacatorul dorește să deturneze fluxul normal de execuție spre o funcție/adresă injectată de el (shellcode) sau către una existentă (return to libc). Pentru aceasta are nevoie de adresa precisă a acelei funcții; dacă procesul folosește ASLR este foarte dificil (în special pe sistemele pe 64 de biți) de identificat adresa funcției.
7. Ce înseamnă "stack buffer overflow"?
   - **Răspuns**: Stack buffer overflow este depășirea unui buffer local unei funcții (alocat pe stivă). Adică în cazul unui buffer cu 10 elemente, accesăm al 15-lea sau al 20-lea element. Putem suprascrie pointeri sau adresa de retur a funcției și dând naștere, astfel, unor atacuri de securitate.
8. Ce este un shellcode?
   - **Răspuns**: Un shellcode este o secvență de cod binar care se dorește a fi injectat, prin intermediul unei vulnerabilități de securitate, în codul unui proces care rulează. Apoi procesului îi este deturnat fluxul de execuție pentru a executa shellcode-ul. De regulă shellcode-ul urmărește obținerea unui shell prin execuția unei instrucțiuni de forma exec("/bin/bash").
9. Ce înseamnă un atac de tipul "return-to-libc"?
   - **Răspuns**: Un atac de tipul "return-to-libc" presupune suprascrierea unui pointer sau a adresei de retur a unei funcții cu adresa unei funcții din biblioteca standard C (de obicei funcția system). În acest fel se deturnează fluxul normal de execuție al programului către o altă adresă încercându-se obținerea unui shell.
10. De ce NU previne flag-ul NX (No eXecute) atacurile de tipul return-to-libc?
    - **Răspuns** Flag-ul NX marchează regiuni specifice (stiva, de exemplu) ca neexecutabile. Un atac de tipul return-to-libc forțează un jump nevalid la o adresă din zona de cod (text) care este executabilă și nu este afectată de prezența sau nu a flag-ului NX.
11. În ce mod protejează chroot împotriva atacurilor de tip shellcode?
    - **Răspuns** În chroot jail nu este adăugat executabil de shell (/bin/bash sau /bin/sh). În acest caz, nu se poate executa un shell dintr-un program chroot-at și, deci, nici un shellcode.
12. De ce NU este stocată valoarea "salt" și în fișierul /etc/passwd?
    - **Răspuns** Deoarece valoarea este corelată cu hash-ul, care se stochează în /etc/shadow. De asemenea, din considerente de securitate, nu ar trebui să fie stocată într-un fișier care poate fi citit de orice user.
13. De ce are tehnica ASLR impact redus pe un sistem pe 32 de biți?
    - **Răspuns** Tehnica ASLR împiedică atacatorul să cunoască adresa funcției dorite din cadrul zonei de cod prin maparea acestora în puncte aleatoare din spațiul de adresă. Pentru sisteme pe 32 de biți, spațiul virtual nu este suficient de mare, atacatorii putând "căuta" adresa dorită pentru a realiza atacuri de tipul return-to-libc.
14. De ce este considerată funcția `memcpy` mai "sigură" decât funcția `strcpy`?
    - **Răspuns**: Deoarece funcția `memcpy` specifică dimensiunea buffer-ului care trebuie copiat, în timp ce `strcpy` presupune că buffer-ele sunt alocate cum trebuie, astfel încât `strlen(sursă) < sizeof(dest)`.
15. Completați zona punctată a următoarei secvențe de cod cu un apel de bibliotecă, astfel încât:
    - a) programul să fie vulnerabil la stack smashing;
    - b) programul să nu fie vulnerabil la stack smashing.
    - Motivați alegerile făcute.

```
#define INPUT "1234"
int main(void)
{
        char a[2];
        memcpy(a, INPUT, ...);
        return 0;
}
```

    - **Răspuns**: Pentru ca programul să fie vulnerabil la stack smashing, trebuie să permită copierea unui număr de octeți independent de dimensiunea buffer-ului destinație. Pentru ca programul să nu fie vulnerabil, trebuie să permită copierea unui număr de octeți mai mare decât dimensiunea buffer-ului destinație.
      1. a) Apeluri de bibliotecă: `strlen(INPUT)`, `sizeof(INPUT)`, etc.
      2. b) Apeluri de bibliotecă: `sizeof(a)`, etc.
16. Cum previne tehnica ASLR (*Address Space Layout Randomization*) atacuri de tipul *return-to-libc*?
    - Un atac de tipul *return-to-libc* presupune suprascrierea adresei de retur din cadrul stack frame-ului unei funcții cu o adresă din zona de cod a procesului (cel mai probabil codul aferent pentru`system` sau `exec`).
    - Tehnica ASLR împiedică atacatorul să cunoască adresa funcției dorite din cadrul zonei de cod prin maparea acestora în puncte aleatoare din spațiul de adresă. În cazul unui spațiu virtual suficient de mare (spre exemplu în cazul sistemelor pe 64 de biți), timpul de "căutare" a adresei dorite face impractice atacuri de tipul *return-to-libc*.
17. Cum previne flag-ul NX (*No eXecute*) atacurile de tipul *stack buffer overflow*?
    - Atacurile de tipul *stack buffer overflow* presupun suprascrierea adresei de retur din cadrul stack frame-ului unei funcții prin trecerea peste limita unui buffer alocat pe stivă. În general, adresa este suprascrisă chiar cu o adresă de pe stivă din cadrul buffer-ului.
      1. Atacatorul completează în cadrul buffer-ului de pe stivă un shell code și apoi suprascrie adresa de retur cu adresa din cadrul buffer-ului unde începe shell code-ul.
    - Flag-ul NX este un flag/bit hardware ce marchează anumite pagini ca fiind neexecutabile. Exemple de zone care sunt candidați pentru acest bit sunt stiva, heap-ul și zonele de date. Marcarea stivei ca fiind neexecutabilă, prin folosirea flag-ului NX, înseamnă imposibilitatea realizării unui atac de tipul *stack buffer overflow*.
18. Cum sunt prevenite atacurile de tipul *stack buffer overflow* folosind soluții de tipul *stack smashing protection*?
    - Vezi descriere *stack buffer overflow* mai sus.
    - Soluțiile de tipul *stack smashing protection* presupun plasarea unei valori speciale (*canary value*) înaintea adresei de retur dintr-un stack frame.
    - Întrucât majoritatea atacurilor presupun operații pe șiruri, suprascrierea adresei de retur va însemna și suprascrierea valorii speciale. Înainte de revenirea de funcție se verifică această valoare. Dacă a fost modificată atunci se generează eroare.
19. De ce executabilul aferent comenzii *su* (*/bin/su*) are bitul *setuid* activat?
    - Un proces care ia naștere din cadul executabilului `/bin/su` execută o serie de operații privilegiate precum:
      1. parcurgerea fișierului `/etc/shadow` pentru a citi parola;
      2. schimbarea user id-ului curent (practic schimbarea utilizatorului).
    - Pentru executarea operațiilor privilegiate este nevoie de drept de superuser. Prezența bituluisetuid permite efectuarea acestora.

THREAD-URI & MECANISME DE SINCRONIZARE

1. Care este un avantaj și un dezavantaj al folosirii unei implementări de thread-uri în user space (*user-level threads*)?
   - **Răspuns**:
     - Avantaje pot fi:
       - timp de creare mai mic decât thread-urile kernel level
       - schimbări de context mai rapide
       - control mai bun asupra aspectelor de planificare (totul se întâmplă în user space, sub controlul programatorului)
     - Dezavantaje pot fi:
       - blocarea unui thread duce la blocarea întregului proces
       - nu poate fi folosit suportul multiprocesor
2. De ce este importantă o instrucțiune de tip `TSL` (*test and set lock*) la nivelul procesorului?
   - **Răspuns**: Implementările de mecanisme de sincronizare se bazează pe instrucțiuni hardware. Fără suportul procesorului pentru operații atomice (precum `TSL` sau `cmpxchg`) nu ar fi posibilă implementarea unor mecanisme precum spinlock-uri. Astfel de instrucțiuni vor fi disponibile pentru orice procesor pentru a permite implementarea mecanismelor de sincronizare.
3. Care este un avantaj și un dezavantaj al folosirii unei implementări de thread-uri cu suport în kernel (*kernel-level threads*)?
   - **Răspuns**:
     - Avantaje pot fi:
       - dacă un thread se blochează celelalte thread-uri pot rula
       - se folosește suportul multiprocesor al sistemului
       - planificator robust asigurat de sistemul de operare, preemptiv
     - Dezavantaje pot fi:
       - timp de creare mai mare (necesită apel de sistem)
       - schimbare de context mai lentă (overhead datorat trecerii în kernel space pentru invocarea planificatorului)
4. Precizați un dezavantaj al folosirii primitivelor de sincronizare.
   - **Răspuns**: Dezavantaje sunt:
     - lock contention: mai multe thread-uri așteaptă la un lock (un singur thread poate accesa regiunea critică protejată de lock) → ineficiență
     - lock overhead: apelul de lock/unlock produce overhead, de multe ori însemnând apel de sistem
     - serializarea codului: codul protejat de un lock este cod serial, accesibil unui singur thread; nu avem paralelism
     - deadlock: o folosire necorespunzătoare a primitivelor de sincronizare duce la deadlock sau livelock
5. De ce dimensiunea spațiului virtual de adresă al unui proces crește în momentul creării unui thread (chiar dacă thread-ul nu ajuns încă să se execute)?
   - **Răspuns**: În momentul creării unui thread se alocă o stivă nouă acelui thread. În mod implicit, pe sistemele Linux, dimensiunea stivei este de 8 MB de memorie, observând o creștere semnificativă a spațiului virtual de adresă al procesului.
6. Când este recomandat să folosim un spinlock în locul unui mutex?
   - **Răspuns**: Spinlock-ul folosește busy-waiting și are operații de `lock()` și `unlock()` ieftine prin comparație cu mutex-ul. Operațiilor de `lock()` și `unlock()` pe mutex sunt de obicei costisitoare întrucât pot ajunge să invoce planificatorul. Având operații rapide, spinlock-ul este potrivit pe secțiuni critice de mici dimensiuni în care nu se fac operații blocante; în aceste cazuri faptul că face busy-waiting nu contează așa de mult pentru că va intra rapid în regiunea critică. Dacă am folosi un mutex pentru o regiune critică mică, atunci overhead-ul cauzat de operațiile pe mutex ar fi relativ semnificativ față de timpul scurt petrecut în regiunea critică, rezultând în ineficiența folosirii timpului pe procesor.
7. De ce schimbarea de context între două thread-uri ale aceluiași proces este, în general, mai rapidă decât schimbarea de context între două procese?
   - **Răspuns**: Schimbarea între două thread-uri ale aceluiași proces este mai rapidă decât schimbarea de context între două procese pentru că nu este nevoie de schimbarea spațiului de adresă. Schimbarea spațiului de adresă este relativ costisitoare pentrucă presupune schimbarea tabelei de pagini și golirea multor intrări din TLB.
8. Care sunt cele două tipuri de operații aferente mecanismelor de sincronizare prin secvențiere/ordonare?
   - **Răspuns**: Cele două operații aferente mecanismelor de sincronizare prin secvențiere/ordonare sunt:
     - `wait()` pentru așteptarea îndeplinirii unei condiții după care thread-ul curent va rula;
     - `notify()` sau `signal()` pentru a anunța thread-ul/thread-urile blocate în operația `wait()` de îndeplinirea condiției.
9. Precizați un avantaj al folosirii thread-urilor în locul proceselor
   - **Răspuns**: Avantaje pot fi:
     - timp de creare mai mic
     - timp de schimbare de context mai rapid pentru thread-urile aceluiași proces
     - partajare facilă a datelor între thread-urile aceluiași proces
     - în cazul implementărilor cu suport la nivelul nucleului (kernel-level threads), blocarea unui thread nu blochează întregul proces, ducând la o productivitate sporită
10. Ce înseamnă "lock contention"?
    - **Răspuns**: Lock contention se referă la accesul concurent la un lock/mutex din partea multor thread-uri. În cazul în care multe thread-uri așteaptă la un lock, eficiența este scăzută, doar un singur thread putând accesa la un moment dat regiunea critică protejată de lock.
11. Ce efect are apelul exit() în cadrul unei codului rulat de un thread?
    - **Răspuns**: Apelul exit() încheie execuția procesului curent, indiferent de punctul în care este apelat. Dacă în cadrul funcției unui thread se apelează exit() atunci procesul aferent thread-ului își încheie execuția (împreună cu toate thread-urile procesului).
12. Indicați un dezavantaj/neajuns al primitivelor de acces exclusiv chiar și în cazul folosirii corespunzătoare (în care se asigură coerența datelor).
    - **Răspuns**: Dezavantaje sunt:
      - lock contention: mai multe thread-uri așteaptă la un lock (un singur thread poate accesa regiunea critică protejată de lock) → ineficiență
      - lock overhead: apelul de lock/unlock produce overhead, de multe ori însemnând apel de sistem
      - serializarea codului: codul protejat de un lock este cod serial, accesibil unui singur thread; nu avem paralelism
13. Ce se întâmplă în cazul unui acces nevalid la memorie în cadrul codului rulat de un thread?
    - **Răspuns**: În cazul unui acces nevalid la memorie, sistemul de operare generează o excepție (semnalul SIGSEGV pe Linux) al cărei efect este încheierea execuției procesului curent. Indiferent de modul în care este generat accesul (din cadrul funcției unui thread), procesul își încheie execuția, împreună cu toate thread-urile aferente.
14. Care este dezavantajul folosirii de regiuni critice de mici dimensiuni (granularitate fină)?
    - **Răspuns**: Într-o regiune critică mică, overhead-ul cauzat de apelurile lock și unlock este semnificativ față de acțiunea efectivă realizată în regiunea critică.

Dacă regiunea critică este accesată foarte des atunci acest overhead devine semnificativ la nivelul întregului set de acțiuni executate de thread.

1. Precizați un dezavantaj al folosirii thread-urilor în locul proceselor.
   - **Răspuns**: Dezavantaje sunt:
     - dacă un thread execută un apel nevalid la memorie atunci se generează excepție și întreg procesul își încheie execuția
     - zonele de memorie folosite la comun impun folosirea mecanismelor de sincronizare care pot produce probleme dificil de depanat
     - un număr semnificativ de thread-uri duce la penalizări de performanță față de alte abordări care nu folosesc mai multe thread-uri sau procese (de exemplu event-based I/O sau operații neblocante/asincrone)

1. Două thread-uri folosesc două mutex-uri. Cum se poate ajunge la deadlock?
   - **Răspuns**: Fie T1, T2 cele două thread-uri și mutex_a și mutex_b cele două mutex-uri. Situația în care se poate ajunge la deadlock presupune ca thread-ul T1 să execute un cod de forma lock(mutex_a); lock(mutex_b); iar thread-ul T2 să execute un cod de forma lock(mutex_b); lock(mutex_a); Dacă thread-ul T2 rulează între cele două apeluri lock ale thread-ului T1 atunci acesta va achiziționat mutex-ul mutex_b. În acea situație T1 va avea achiziționat mutex-ul mutex_a și va aștepta după eliberarea mutex-ului mutex_b, iar T2 invers. În această situație nici un thread nu poate trece mai departe, ambele rămânând blocate: deadlock.
2. De ce este necesară folosirea mutex-urilor, și nu a spinlock-urilor, pentru regiunile critice cu operații de I/O?
   - **Răspuns** Regiunile critice cu operații de I/O sunt, de obicei, lungi. De asemenea, operațiile I/O pot duce la blocarea thread-ului, caz în care nu poate fi folosit spinlock-ul.
3. În ce situație este posibilă apariția unui deadlock pe o singură resursă critică?
   - **Răspuns** Fie procesul P1 care a acaparat resursa critică și procesele P2, P3, … , Pn care așteaptă eliberarea resursei respective. Un deadlock pe resursa respectivă va apărea dacă procesul P1 nu va elibera resursa critică, fie datorită codului său (nu există instrucțiunea de release/unlock, intră într-un ciclu infinit, etc.), fie deoarece a fost terminat prin semnal SIGKILL.
4. Descrieți două diferențe între un mutex și un semafor binar.
   - **Răspuns**
   - Mutexul este folosit pentru acces exclusiv, semaforul binar este folosit pentru sincronizare
   - Mutexul poate fi eliberat doar de procesul care l-a ocupat, în timp ce orice proces poate incrementa semaforul.
   - Mutexul este mereu inițializat unlocked, semaforul binar poate fi inițializat la valoarea 0.
   - Mutexul are un overhead mai mic decât semaforul binar.
5. De ce overhead-ul creării unui nou thread este independent de utilizarea mecanismului copy-on-write?
   - **Răspuns** Copy-on-write are sens doar între două tabele de pagini diferite, adică între două procese diferite, întrucât se partajează paginile fizice între pagini virtuale din procese diferite. Thread-urile din același proces partajează tabela de pagină, astfel că overhead-ul creării unui nou thread este independent de copy-on-write.
6. Câte fire de execuție va avea un proces multithreaded în urma executării unui apel din familia exec()?
   - **Răspuns** În urma executării unui apel din familia exec(), spațiul de adresă al procesului existent va fi înlocuit cu spațiul de adresă al noului proces. Acesta va avea inițial n singur fir de execuție.
7. Fie un program multithreaded care rulează pe un sistem uniprocesor cu sistem de operare cu suport multithreaded. În ce situație este mai eficientă folosirea user-level threads în fața kernel-level threads?
   - **Răspuns** Folosirea user-level threads este mai eficientă în cazul în care procesul face multe operații CPU intensive, deoarece este mai rapid context switch-ul.
8. Fie f o funcție care nu este reentrantă. Cum trebuie aceasta apelată pentru a fi thread-safe?
   - **Răspuns** Pentru a fi thread-safe, trebuie apelată folosind un mecanism de acces exclusiv. Dacă se pune lock, un singur thread va putea accesa funcția, care devine, astfel, thread-safe.
9. Dați exemplu de avantaj, respectiv dezavantaj al sincronizării folosind semafoare binare în fața spinlock-urilor.
   - **Răspuns**:
     1. Avantaj: Deoarece semafoarele binare nu folosesc busy waiting, pot fi folosite pentru secțiuni critice de orice dimensiune.
     2. Dezavantaj: Un proces va ceda procesorul dacă nu poate lua semaforul. Operația de `down` are un overhead ridicat, datorită schimbării de context.
10. Dați exemplu de avantaj, respectiv dezavantaj al sincronizării folosind spinlock-uri în fața mutex-urilor.
    - **Răspuns**:
      1. Avantaj: Un proces nu va ceda procesorul dacă nu poate lua spinlock-ul. Operația de `lock` are un overhead scăzut.
      2. Dezavantaj: Deoarece spinlock-urile folosesc busy waiting, sunt recomandate doar pentru secțiuni critice mici, ce se execută rapid.
11. Care metodă de sincronizare (mutex sau spinlock) este mai avantajoasă pentru sincronizarea accesului la următoarea regiune critică și de ce? Prezentați cel puțin un motiv.

```
12....

13.// start regiune critica
14.c=a+b;
15.a=b;
16.b=c;
17.// stop regiune critica

...
```

    - **Răspuns**: Deoarece regiunea critică este mică, este mai avantajoasă folosirea unui spinlock pentru sincronizare, deoarece se evită overhead-ul unei schimbări de context.
18. Care dintre următoarele operații pot genera schimbare de context cu o cauză diferită de expirarea cuantei? De ce? `lock(&mutex)`, `unlock(&mutex)`, `spin_lock(&spinlock)`, `spin_unlock(&spinlock)`, `down(&semaphore)`, `up(&semaphore)`.
    - **Răspuns**:
      1. `lock(&mutex)` - încercarea de a obține un mutex inaccesibil trece procesul în starea `WAITING` -**DA**
      2. `unlock(&mutex)` - la ieșirea din zona critică, procesele care așteptau la mutex vor trece din starea `WAITING` în starea `READY` - **DA**
      3. `spin_lock(&spinlock)` - încercarea de a obține un spinlock inaccesibil nu trece procesul în stare`WAITING` - **NU**
      4. `spin_unlock(&spinlock)` - la ieșirea din zona critică, procesul care aștepta la spinlock se va debloca, dar va rămâne în starea `RUNNING` - **NU**
      5. `down(&semaphore)` - încercarea de a obține un semafor inaccesibil trece procesul în stare`WAITING` - **DA**
      6. `up(&semaphore)` - la ieșirea din zona critică, procesele care așteptau la semafor vor trece din starea `WAITING` în starea `READY` - **DA**
19. Dați 2 exemple de resurse partajate între *thread-urile* aceluiași proces.
    - **Răspuns**: *Thread-urile* aceluiași proces partajează descriptorii de fișier, spațiul de adrese (memoria), masca de semnale, deoarece acestea sunt resurse la nivel de proces, nu de thread.
      1. Ele folosesc aceleași segmente de memorie `.heap`, `.data` și `.bss`. (deci și variabilele stocate în ele)
20. Dați 2 exemple de resurse care **NU** sunt partajate între *thread-urile* aceluiași proces.
    - **Răspuns**: Fiecare *thread* al unui proces are un context de execuție propriu, format din stivă și set de regiștri (deci și un contor de program - registrul (E)IP). De asemenea, TLS/TSD reprezintă variabile specifice unui *thread*, invizibile pentru celelalte *thread-uri*.
21. Dați exemplu de situație/aplicație în care este mai avantajos să se folosească un număr `X` de *kernel-level threads* într-un proces și o situație/aplicație în care este mai avantajos să se folosească un număr `Y` de *kernel-level threads*, cu `X < Y`. Ambele situații se vor raporta la același sistem dat.
    - **Răspuns**:
      1. Este mai avantajos să se folosească un număr `X` (mai mic) de *kernel-level threads* într-un proces care execută operații CPU-intensive care folosesc puține apeluri blocante, deoarece utilizarea mai multor *kernel-level threads* ar însemna mai multe schimbări de context, care ar scădea performanța.
      2. Este mai avantajos să se folosească un număr `Y` (mai mare) de *kernel-level threads* într-un proces care este I/O-intensive, având multe apeluri blocante, permițând unui număr mai mare de thread-uri să și continue execuția în cazul blocării unui thread.
22. Prezentați un avantaj și un dezavantaj al folosirii *user-level threads* față de *kernel-level threads*.
    - **Răspuns**:
      1. Avantaje :
         1. schimbarea de context nu implică kernelul, deci vom avea comutare rapidă
         2. planificarea poate fi aleasă de aplicație; aplicația poate folosi acea planificare care favorizează creșterea performanțelor
         3. firele de execuție pot rula pe orice sistem de operare, inclusiv pe sisteme de operare care nu suportă fire de execuție la nivel kernel.
      2. Dezavantaje :

1. kernel-ul nu știe de fire de execuție, astfel că dacă un fir de execuție face un apel blocant toate firele de execuție planificate de aplicație vor fi blocate
2. nu se pot utiliza la maximum resursele hardware: kernelul va vedea un singur fir de execuție și va planifica procesul respectiv pe maximum un procesor, chiar dacă aplicația ar avea mai multe fire de execuție planificabile în același timp.

23. Se poate implementa un semafor folosind **doar** mutex-uri? Dar un mutex folosind **doar** semafoare?

- **Răspuns**:
  - Un mutex se poate implementa ca un semafor binar. Răspuns afirmativ la a doua întrebare.
  - (Mulțumim lui Răzvan Pistolea pentru observație) Pentru implementarea unui semafor cu mutex-uri, este nevoie de un mutex care să îndeplinească rolul unei variabile condiție (pentru notificare). În acest caz, trebuie ca acel mutex să nu poată fi incrementat de două ori (adică să se apeleze release de două ori). În plus, trebuie asigurată sincronizarea corespunzătoare, pentru a preveni condițiile de cursă, precum două thread-uri care trec de semafor în momentul în care valoarea acestuia este 1.

24. Fie un sistem cu două procesoare. Câte procese pot *"aștepta"* simultan eliberarea unui spinlock? Dar a unui mutex?
    - **Răspuns**:
      1. La un mutex pot aștepta oricâte procese. Dacă mutexul este achiziționat, procesele se blochează și trec în starea `WAITING`. Pot exista oricâte procese în starea `WAITING`.
      2. Dacă un proces a achiziționat un spinlock, cel mult un alt proces poate *"aștepta"* (busy waiting) așteptarea acestuia pe un alt procesor. Dacă cele două procese (ce rulează pe procesor) au ajuns la un livelock (ambele așteaptă la spinlock) sistemul este "agățat"; ambele așteaptă în acel moment la spinlock.
         1. Un proces care deține un spinlock nu va fi planificat pentru că atunci alte procese ar aștepta nedefinit și sistemul devine instabil. Un spinlock protejează o zonă scurtă și rapidă. De asemenea, un proces care *"așteaptă"* la un spinlock nu va fi preemptat pentru că așteptarea este că va aștepta puțin la procesor (prin busy waiting).
      2. În concluzie, pe un sistem cu două procesoare, eliberarea unui spinlock poate fi așteptată de cel mult două procese.

25. Este nevoie de folosirea unui mecanism de sincronizare la folosirea memoriei partajate? Dar la folosirea cozilor de mesaje?
    - **Răspuns**:
      1. Da, la memoria partajată. Două (sau mai multe procese) pot decide să scrie în memorie partajată și pot rezulta date incorecte. Este nevoie de protejarea prin folosirea unui mecanism de locking.
      2. În cazul folosirii cozilor de mesaje nu este nevoie de folosirea unui mecanism de sincronizare. Aceasta deoarece operațiile pe cozile de mesaje sunt atomice și serializate.
         1. Scrierea unui mesaj se face după altă scriere, iar citirea unui mesaj se realizează în momentul în care un mesaj există deja în coadă (altfel se blochează în așteptare). Nu este necesară folosirea unei forme de utilizare de genul `lock(); send(); unlock();`.

26. Dați exemplu de situație în care trei procese care interacționează ajung în deadlock.
    - **Răspuns**:
      1. Cea mai simplă situație este aceea a unei așteptări circulare a proceselor. Procesele, respectiv, `P1`, `P2`, `P3` dețin resursele `R1`, `R2`, `R3` fără a le elibera. Apoi procesul `P1` solicită accesul la resursa `R2`, `P2` la `R3` iar `P3` la `R1`. Fiecare proces așteaptă la o resursă deținută de alt proces, fără ca vreunul din ele să o fi eliberat. Toate sunt blocate – deadlock.

27. Fie un program multithreaded care efectuază multe operații I/O per thread. Este mai eficientă folosirea *user-level threads* sau *kernel-level threads*?
    - Operațiile I/O sunt, în general, operații blocante. Efectuarea unei operații de I/O în cadrul unei implementări de thread-uri *user-level* va bloca întreg procesul, nu doar thread-ul curent.
    - În cadrul unei implementări de thread-uri *kernel-level*, doar thread-ul care a efectuat operația I/O, celelalte thread-uri putând fi planificate pe procesor.
    - Implementarea *kernel-level threads* este, în acest caz mai eficientă, prin folosirea procesorului/procesoarelor de mai multe thread-uri ale proceselor.

28. Fie un program multithreaded care rulează pe un sistem multiprocesor. Este mai eficientă folosirea*user-level threads* sau *kernel-level threads*?
    - În cadrul unei implementări *user-level*, un singur thread rulează pe procesor; planificatorul de la nivelul nucleului "vede" o singură entitate planificabilă – procesul corespunzător.
    - În cadrul unei implementări *kernel-level*, fiecare thread al procesului poate rula pe un procesor. Se poate întâmpla ca, pe un sistem cu număr suficient de procesoare, toate thread-urile unui proces să ruleze pe procesoare.
    - În concluzie, este mai eficientă folosirea unei implementări *kernel-level* prin utilizarea mai bună a procesoarelor, rezultând, astfel, într-o productivitate mai bună.

29. Fie următoarea secvență de cod:

```
30. int main(void)
31. {
32.     int a = 0;
33.     pthread_create(...);
34.     a++;
}
```

Care va fi valoarea lui `a` la finalul secvenței?

- În urma apelului `pthread create` thread-ul principal (*main thread*, *master thread*) își continuă execuția; thread-ul nou creat execută funcția transmisă ca argument funcției `pthread create`.
- Variabila `a` va fi incrementată doar de thread-ul principal, astfel că valoarea finală a acesteia va fi 1.
- O situație diferită este aceea în care `a` este transmisă într-o formă ca argument funcției aferente thread-ului nou creat și acesta o modifică; de asemenea, o valoare diferită rezultă în momentul în care thread-ul nou creat accesează variabila a pe stiva thread-ului principal (foarte puțin probabil, dar posibil).

35. În ce situație pot două thread-uri din două procese diferite să partajeze o pagină de memorie?
    - Două thread-uri din două procese diferite nu pot partaja o pagină de memorie decât în cazul în care cele două procese o partajează la rândul lor.
    - Întrebarea se transformă, așadar, în "când pot două procese diferite să partajeze o pagină de memorie?"
      1. când cele două procese referă au mapată aceeași pagină fizică (folosint apeluri de forma`mmap`);
      2. când cele două procese sunt înrudite și referă o pagină prin copy-on-write;
      3. când cele două procese au mapat același executabil/aceeași bibliotecă (codul acestora este read-only și paginile fizice aferente sunt partajate).

I/O – DISPOZITIVE DE INTRARE/IESIRE

1. Indicați două obiective ale algoritmilor de planificare a cererilor pentru hard disk, și dați un exemplu de algoritm care le îndeplinește.
   - **Răspuns**: Un obiectiv este performanță ridicată. Un alt obiectiv este fairness: asigurarea că toate procesele au acces echitabil la resurse și că un proces nu așteaptă mai mult ca altul accesul la disc. Un algoritm care colectează mai multe cereri și apoi le sortează și agregă, independent de procesul care le cauzează va atinge obiectivele. Algoritmi precum C-SCAN sau C-LOOK sau altele satisfac aceste obiective.
2. Precizați două diferențe între un *symbolic link* și un *hard link*.
   - **Răspuns**: Un symbolic link are un inode al său, pe când un hard link este un dentry (un nume și un index de inode). Un symbolic link poate referi directoare în timp ce un hard link nu; un symbolic link poate fereri un fișier de pe altă partiție/alt sistem de fișiere, în timp ce un hard link nu.
1. Ce limitează performanța hard disk-ului în cazul accesului aleator la date din diverse zone ale discului?
   - **Răspuns**: Mutarea capului de citire pe sectoarele/zonele necesare. Dacă există acces aleator, atunci datele vor fi plasate în diverse zone iar o operație va consta în două suboperații:
     a. plasarea capului de citire
     b. citirea sau scrierea datelor respective
     - Dacă datele sunt plasate aleator, operația de plasare va dura mult și va limita performanța; putem optimiza prin ordonarea cererilor și limitarea timpului de accesare. Operația de citire și scriere este standard, ține de mecanica discului, nu o putem optimiza.

   Un director conține `N` subdirectoare. Câte hard link-uri pointează la acest director?

   Răspuns: Directorul va avea `N+2` hard link-uri. `N` hard link-uri sunt date de intrarea `..` (*dot dot*) a fiecărui subdirector (link către directorul părinte). Celelalte două hard link-uri sunt numele directorului și intrarea `.` (*dot*) care referă directorul însuși.

De ce la plăcile de rețea de mare viteză are sens folosirea polling în locul întreruperilor pentru partea de intrare/ieșire?

Răspuns: Având viteze mari, vor veni pachete foarte des și vor fi generate întreruperi foarte des. În această situație, procesorul va fi ocupat foarte mult timp rulând rutine de tratare a întreruperilor. Prin trecere la polling, procesorul interoghează placa de rețea și, dacă are date, le citește repede, fără întreruperi. În restul timpului face și alte lucruri, fără a mai consuma timp în rutina de tratare a întreruperilor.

Care este un avantaj și un dezavantaj al alocării indexate (cu *i-node*) pentru blocuri de date pentru fișiere?

Răspuns: Principalul dezavantaj al alocării indexate este limitarea dimensiunii fișierului la numărul de intrări din lista de indecși (pointeri către blocuri). Avantajele este accesul rapid la blocuri (se citește indexul) și absența fragmentării externe: blocurile se pot găsi oriunde și pot fi referite din lista de indecși. Dezavantajul este compensat prin folosirea indirectării (simple, duble, triple) ducând la o mai mare dimensiune a fișierului, dar introducând un alt dezavantaj: timp mai mare de acces pentru blocurile din partea finală a fișierului; întrucât se trece prin blocurile de indirectare. Un dezavantaj aici poate fi și ocuparea de blocuri doar cu indecși, în loc să conțină date efective.

De ce operația `lseek()` nu are sens pe dispozitive de tip caracter, ci doar pe dispozitive de tip bloc?

Răspuns: Pentru că pe dispozitivele de tip caracter datele vin și sunt citite/scrise octet cu octet, ca într-o țeavă. Nu putem anticipa date și ne putem plasa mai sus sau mai jos pe banda de date. În cazul dispozitivelor de tip bloc însă, datele se găsesc pe un spațiu de stocare pe care ne putem plimba/glisa; putem "căuta" date prin plasarea pe un sector/bloc al dispozitivului de stocare și atunci operația `lseek()` are sens.

Ce conțin blocurile de date aferente unui inode de tip director?

Răspuns: Conțin un vector de *dentry*-uri. Un *dentry* este o structură ce conține numele fișierului și indexul inode-ului aferent. Fiecare intrare din director (indiferent de tipul acesteia: fișier, director, link symbolic) are un *dentry*.

La ce este util buffer/page cache-ul?

Răspuns: Datele accesate recent de pe disc/sistemul de fișiere sunt reținute în memorie pentru acces rapid. Șansele sunt mari ca acele date să fie reaccesate în viitor. Memoria fiind mult mai rapidă ca discul, se mărește viteza de lucru a sistemului.

Care este un avantaj al alocării indexate față de alocarea contiguă la nivelul sistemului de fișiere?

Răspuns: Alocarea indexată reduce fragmentarea externă: un fișier poate folosi blocuri din poziții aleatoare de pe disc.

De ce în cadrul unei plăci de rețea de mare viteză (10Gbit) este problematic să se folosească un model bazat DOAR pe întreruperi? (în general se folosește un model hibrid de întreruperi și polling)

Răspuns: Pentru că pachetele vin cu vitează foarte mare și generează multe întreruperi. În cazul în care s-ar folosi doar un sistem bazat pe întreruperi, ar exista riscul ca procesorul să fie ocupat doar de rularea de rutine de tratare a întreruperilor.

Ce este un hard link?

Răspuns: Un hard link se referă la situația în care avem mai multe intrari in directoare (dentry-uri) care pointeaza catre acelasi fișier pe disc (inode). Un nume sau un dentry denotă un hard link.

Care este o caracteristică a unui dispozitiv de tip bloc?

Răspuns: Un dispozitiv de tip bloc permite acces aleator la date, nu secvențial ca în cazul unui dispozitiv de tip caracter. De asemenea, un dispozitiv de tip bloc lucrează cu blocuri de date, nu cu câte un caracter/byte așa cum este cazul unui dispozitiv de tip caracter.

De ce este importantă ordonarea cererilor în cadrul unui planificator de disk (disk scheduler)?

Răspuns: Dacă cererile nu sunt ordonate, se fac multe operații de căutare (seek) pe disk pentru fiecare cerere, ceea ce înseamnă timp consumat. Prin ordonarea cererilor timpul de căutare (seek) este minimizat: se trece, în ordine, de la un bloc la alt bloc.

Care este un avantaj al folosirii operațiilor I/O asincrone?

Răspuns: După momentul lansării unei operații I/O asincrone, sistemul/procesul poate executa alte operații, nu trebuie să se blocheze în așteptarea încheierii acesteia. Acest lucru conduce la o eficiență sporită a sistemului.

Ce este un inode? Ce informații conține (în linii mari)?

Răspuns: Este o structură/tip de date care identifică un fișier pe disc. Un inode identifică orice fișier (fișier obișnuit, director, link simbolic) și conține metadate despre un fișier: permisiune de acces, deținător, timestamp-uri, dimensiune, contor de link-uri, pointeri la blocurile de date etc.

Cu ce diferă o operație I/O sincronă de o operație I/O blocantă?

Răspuns O operație sincronă întoarce rezultatele prezente în acel moment, indiferent dacă operația s-a încheiat sau nu. O operație blocantă blochează procesul curent.

Care sunt cele două argumente ale unei instrucțiuni de tipul IN sau OUT (pentru lucrul cu port-mapped I/O)?

Răspuns Cele două argumente sunt registrul procesorului și portul dispozitivului I/O.

De ce nu are sens sortarea operațiilor I/O pentru dispozitive caracter (char devices)?

Răspuns Dispozitivele de tip caracter obțin datele în format secvențial, astfel că operațiile I/O nu pot fi sortate.

Prezentați un avantaj al folosirii întreruperilor în fața polling-ului.

Răspuns Polling-ul este un mecanism de tip busy-waiting, deci procesorul va fi ținut ocupat în așteptarea datelor. În schimb, întreruperile nu țin procesorul ocupat, oferind o utilizare mai bună a acestuia.

Cum se modifică numărul de inode-uri ocupate, respectiv numărul de dentry-uri de pe o partiție în cazul creării unui fișier nou obișnuit (regular file)? Explicați.

Răspuns În cazul creării unui fișier nou obișnuit (regular file), se va crea un dentry nou, în cadrul directorului părinte, și va fi ocupat un inode.

Fie afirmația "Spațiul ocupat pe disc de un director este constant." Precizați și justificați valoarea de adevăr a afirmației.

Răspuns Afirmația este falsă. Spațiului unui director pe disc crește pe măsură ce apar intrări noi (dentry-uri) în cadrul directorului.

Cum se modifică numărul de inode-uri, respectiv dentry-uri de pe o partiție în cazul creării unui director nou? Explicați.

Răspuns În cazul creării unui director nou, se vor crea trei dentry-uri noi (unul în cadrul directorului părinte și două: "." și ".." în cadrul directorului nou creat), și va fi ocupat un inode.

Cum se modifică numărul de inode-uri, respectiv dentry-uri de pe o partiție în cazul creării unui hard-link? Explicați.

Răspuns În cazul creării unui hard-link, se va crea un dentry nou, în cadrul directorului părinte, fără a se modifica numărul de inode-uri.

Fie un program multithreaded cu *user-level threads* care efectuează multe operații I/O per thread. Este mai eficientă folosirea operațiilor I/O blocante sau non-blocante?

Răspuns: Efectuarea unei operații blocante în cadrul unei implementări cu *user-level threads* va bloca întreg procesul, nu doar thread-ul curent. De aceea este mai eficientă folosirea operațiilor I/O non-blocante.

Fie afirmația "Întreruperile pot fi folosite la fel de bine și pentru dispozitive care folosesc *memory-mapped I/O* și pe sisteme care folosesc *port-mapped I/O*". Precizați și justificați valoarea de adevăr a afirmației.

Răspuns: Întreruperile sunt semnale folosite de către dispozitive pentru a semnala procesorului finalizarea unei operații de I/O. Acesta va salva starea curentă și va rula rutina de tratare a întreruperii. Utilizarea *memory-mapped I/O* sau a *port-mapped I/O* determină doar modul de adresare a dispozitivelor în cadrul rutinei de tratare, fără a afecta eficiența utilizării întreruperilor. Astfel, afirmația este adevărată.

Care este principala caracteristică a unui dispozitiv de tip caracter? Dați 2 exemple de astfel de dispozitive.

Răspuns: Dispozitivele de tip caracter oferă acces secvențial și transfer de date la nivel de caracter, astfel au viteză redusă.

- Exemple: tastatură, mouse, game controller, port serial, terminal,

Care sunt cele două caracteristici importante ale unui dispozitiv de tip bloc? Dați 2 exemple de astfel de dispozitive.

Răspuns: Dispozitivele de tip bloc oferă acces aleator și transfer de date la nivel de bloc, astfel au o viteză ridicată.

- Exemple: discuri (hard-disk, floppy, unități optice, unități flash), sisteme de fișiere, memoria RAM.

Care sunt *dentry-urile* existente în orice director pe un sistem de fișiere *ext2/3* și ce reprezintă acestea?

Răspuns: Orice director are intrările `.` și `..` care sunt *dentry-uri* către directorul curent, respectiv directorul părinte.

Prezentați un avantaj și un dezavantaj al utilizării sistemelor de fișiere jurnalizate.

Răspuns:

- Avantaj: menținerea unui log al acțiunilor făcute asupra sistemului de fișiere, log ce poate fi consultat în cazul unei defecțiuni și poate ajuta la recuperarea datelor afectate în timpul defecțiunii.
- Dezavantaj: este overhead-ul adus în procesare și spațiul pe suportul fizic consumat pentru menținerea jurnalului

Fie o implementare de *inode* care reține următoarele valori: permisiuni, număr de link-uri, uid, gid, dimensiune și zone directe. Fie fișierul "`a`" și un hard link către acesta, "`b`". Prin ce diferă *inode-urile*celor două fișiere?

Răspuns: Un hard link reprezintă un nou dentry care referă același *inode*, deci *inode-urile* celor două fișiere sunt identice.

Pe un sistem se dorește conservarea numărului de *inode-uri*. Care din cele două tipuri de link-uri,*hard* sau *sym*, trebuie folosite în acest caz și de ce?

Răspuns: Un hard link nu generează *inode-uri* noi, pe când un sym link este un *inode* nou, astfel că trebuie folosite hard link-uri.

Cine/ce generează întreruperi și cine/ce le tratează?

Întreruperile sunt generate de dispozitivele hardware la apariția unui eveniment specific (primirea de date/eliberarea unui buffer local, caz de eroare).

Tratarea întreruperilor este realizează de procesor în cadrul unei rutine de tratare a întreruperii (*IRQ handler* sau *ISR – Interrupt Service Routine*).

Dați exemplu de funcție de API care efectuează operații I/O sincrone și o funcție care efectuează operații I/O asincrone.

Operații I/O sincrone: `read`, `write`. Sunt funcții la apelul cărora se execută operația de I/O.

Operații I/O asincrone: `aio_read`, `aio_write`, `Overlapped I/O`, `select`.

- Primele 3 sunt operații asincrone, neblocante. Asincrone – declanșează o operația I/O fără a urmări "sincron" execuția acesteia; neblocante – nu se blochează în așteptarea încheierii operației.
- `select` – nu este cu adevărat o operație I/O, ci un apel de control a acestora. Poate fi considerată asincronă pentru că nu urmărește execuția unui set de operații; fie sincrone sau asincrone, `select` este un apel blocant care notifică încheierea unei operații I/O.

Care este scopul sortării cererilor de I/E pe care le face un sistem de operare atunci când lucrează cu discul?

Sortarea cererilor de I/E rezultă în organizarea acestora după sectorul de pe disc din care fac parte.

Prin sortarea acestora se minimizează timpul de căutare și poziționarea acului mecanic pe sector corespunzător. În loc să se rotească discul înainte și înapoi pentru poziționare pe capătul/sectorul/platanul corespunzător, se parcurg liniar/secvențial sectoarele descrise în cereri.

Rezultă, astfel, un overhead redus al căutării (*seek*) pe disc în cadrul cererilor de I/E – performanță îmbunătățită.

Cu ce diferă o operație I/O asincronă de o operație I/O neblocantă?

O operație I/O asincronă este pornită dar nu se așteaptă încheierea acesteia. În momentul încheierii operației, se trimite o notificare.

O operație I/O neblocantă se întoarce imediat. Dacă este sincronă atunci va citi/scrie cât îi oferă dispozitivul. Dacă este asincronă, va primi notificare la încheierea operației.

Diferența constă, în general, în primirea sau nu a unei notificări, la sfârșitul încheierii operației de I/O. Acest lucru se întâmplă tot timpul la o operație I/O asincronă; la o operație I/O neblocantă se întâmplă doar dacă aceasta este asincronă – dacă este sincronă (`read` cu `O_NONBLOCK`) atunci nu se primește notificare, doar se întoarce după ce a scris/citit cât i-a oferit dispozitivul.

Care este forma de asociere între *dentry* și *inode*? (unu la unu, mai multe la mai multe etc.)

Un dentry conține un nume și un inode number. Două sau mai multe dentry-uri pot referi același inode, prin intermediul inode number-ului (denumite și hard link-uri). Asocierea este **mai multe dentry-uri la un inode**.

De ce este mai eficientă folosirea unei tabele FAT în locul alocării cu liste pentru gestiunea blocurilor libere ale unui sistem de fișiere?

Alocarea cu liste presupune existența, la sfârșitul fiecărui bloc, a unui pointer către următorul bloc de date. Acest lucru înseamnă că, pentru a ajunge la al N-lea bloc de date, trebuie parcurse (citite de pe disc și aduse în memorie) N-1 blocuri de date.

În cazul tabelei FAT toți pointerii sunt ținuți localizat în cadrul tabelei. La o adresă specifică unui bloc se găsește un pointer către o altă adresă (specifică altui bloc). Overhead-ul/timpul de citire a unui bloc de date este, astfel, mult redus.

Tabela FAT, fiind localizată, poate fi stocată în cache-ul de memorie, mărind astfel viteza de acces la resursele aferente.

De ce nu se poate crea un hard link către un inode de pe alt sistem de fișiere?

Un hard link este un dentry; conține un nume și un număr de inode.

Întrucât două sisteme de fișiere diferite pot avea alocat același inode, este ambiguă prezența unui număr de inode ce referă al sistem de fișiere. Numărul de inode din cadrul dentry-ul aferent va referi inode-ul de pe sistemul curent de fișiere.

În cazul alocării indexate, inode-ul conține un număr limitat de pointeri către blocuri de date, limitând astfel dimensiunea maximă a unui fișier. Cum este rezolvată această problemă?

Problema este rezolvată prin indirectare. O parte din pointerii din cadrul inode-ului nu vor referi blocuri de date, ci vor referi blocuri cu pointeri către blocuri de date (indirectare simplă) sau blocuri cu pointeri către blocuri cu pointeri către blocuri de date (indirectare dublă) etc. Dezavantajul este o latență mai mare de acces pentru blocurile finale.

NETWORKING IN SISTEMUL DE OPERARE

1. O aplicație execută un apel `send()` cu `1024` de octeți de date, iar apelul `send` întoarce `1024`. Alegeți varianta corectă de mai jos și argumentați: În acest moment, aplicația sender poate fi sigură că datele au fost livrate cu succes către
   - nucleul SO de pe sistemul destinație
   - aplicația destinație
   - datele au fost salvate în send-buffer-ul de pe sistemul transmițătorului
   - **Răspuns**: Datele au fost salvate în buffer-ul de send al transmițătorului. În momentul în care datele au fost scrise acolo, apelul se întoarce. Este posibil ca datele să nu fi părăsit sistemul, dar apelul se va întoarce. Stiva TCP se va ocupa de transmiterea datelor din buffer-ul de send către destinație.
2. Explicați motivul pentru care este indicat să folosim pentru apelurile `send()` și `recv()` buffere de dimensiuni mari (de exemplu mai mari decât `100KB`).
   - **Răspuns**: Ca să transmitem mai multe date o dată si să evităm apelurile de sistem generate de apelul send și recv. Un apel de sistem va însemna overhead de timp (intare în kernel mode și apoi revenire în user mode) și overhead de copiere (transfer de date din buffer-ul din user space în buffer-ul din kernel space sau invers).
3. În ce situație operația `send()` pe un socket se blochează?
   - **Răspuns**: Apelul `send()` pe socket se blochează în situația în care buffer-ul din kernel (*send buffer*) nu dispune de loc pentru copierea datelor din buffer-ul de user space. Sau, în anumite cazuri, precum în cazul socketților non-blocanți, dacă nu există nici măcar 1 octet liber în buffer-ul de kernel (*send buffer*).
4. În cazul unui apel `recv()` comandat pentru citirea a `789` de octeți, se citesc `123` de octeți. Cum se explică citirea unui număr mai mic de octeți decât cel comandat?
   - **Răspuns**: În momentul citirii datelor, doar `123` de octeți erau disponibili în buffer-ul din kernel aferent socket-ului (*receive buffer*). În această situație apelul `recv()` se întoarce cu numărul de octeți disponibili (`123`) deși exista spațiu mai mare (`789`) în buffer-ul din user space.
5. În ce situație se poate bloca un apel send pe un socket?
   - **Răspuns**: Apelul send pe socket se blochează dacă buffer-ul de send (transmit, TX) al socketului este plin, adică dacă nu are nici un slot de un octet disponibil. Buffer-ul este plin pentru că nu au apucat să fie transmise

pachetele pe rețea (placă de rețea lentă, congestie sau receiver-ul are și el buffer-ul plin).

6. De ce este considerat sendfile un mecanism de tip zero-copy?
   - **Răspuns**: sendfile transmite un fișier (sau parte a unui fișier) pe un socket. Întrucât nu există copieri între user space și kernel space, așa cum ar fi cazul unor operații de tipul read și send, sendfile este un mecanism de tip zero-copy.
7. Ce valoare (numărul de octeți transmiși) poate întoarce un apel de forma send(s, buffer, 1000, 0)? Apelul urmărește transmiterea unui buffer de 1000 de octeți pe socketul s.
   - **Răspuns**: Apelul send poate întoarce între 1 și 1000 de octeți. Întoarce numărul de octeți disponibili (între 1 și 1000) când are date disponibile. Dacă nu are date disponibile și celălalt capăt nu a închis conexiunea, se blochează. La eroare sau când celălalt capăt a închis conexiunea, se întoarce cu eroare (-1).
8. accept este un apel blocant pe partea server-ului. Ce apel din partea clientului deblochează apelul accept? De ce?
   - **Răspuns**: Apelul connect din partea clientul este cel care stabilește o conexiune la server. În acest caz apelul accept se întoarce și creează un nou socket care va fi folosit pentru transmisia datelor.