Software Stack Q&A

1) Ce este un apel de sistem?

Apelul de sistem este un API expus de kernel pentru domain switch(trecerea din user space în kernel space) Asigură o metodă programatică prin care se cere un serviciu de la nucleu, precum interacțiunea cu I/O-ul, crearea de noi procese sau alocarea de memorie.

https://www.guru99.com/system-call-operating-system.html

2) De ce sunt necesare apeluri de sistem?

Apelurie de sistem sunt necesare din două puncte de vedere:

* pentru a permite folosirea mai ușoară a funcționalităților kernel-ului

* pentru separarea, protejarea, încapsularea kernel-ului de restul aplicațiilor, prin partiționarea pe ring-uri.

—pentru a permite programelor să obțină acces la resursele și serviciile sistemului de operare, pentru a interacționa cu mediul de execuție și pentru a beneficia de funcționalitatea și protecția oferite de acesta. Ele reprezintă puntea de legătură între aplicații și sistemul de operare, permitând programelor să ruleze într-un mediu controlat și să-și îndeplinească sarcinile într-un mod eficient și securizat.

https://www.ibm.com/docs/en/aix/7.2?topic=concepts-system-calls

Partitionarea pe ring-uri se refera la o structura de inel in care nodurile sunt conectate cu vecinii sai, formand un inel inchis. Astfel, datele sunt tinute in siguranta si se permite o foarte buna organizare a apelurilor de system.

3) Ce avantaj / dezavantaje au apelurile de sistem?

Avantaj:

— Mențin kernel-ul izolat de restul aplicațiilor, rulând în mod privilegiat, permințând în același

timp accesul optim la resursele hardware (pentru care există cerere mare din partea

aplicațiilor).

— Protecție și securitate: Prin intermediul apelurilor de sistem, sistemul de operare asigură

protecție și securitate pentru programele care rulează. Accesul la resursele de sistem poate

fi controlat prin intermediul privilegiilor și permisiunilor.

— permit programelor să acceseze și să utilizeze resursele și funcționalitățile oferite de

sistemul de operare.

Dezavantaj:

— Overhead temporal.

Overhead: Apelurile de sistem implică trecerea de la modul utilizator la modul kernel al sistemului de operare, ceea ce poate introduce un overhead de performanță. Trecerea între aceste moduri poate necesita schimbări de context și comutări între spațiul de adresare al utilizatorului și cel al kernelului.

—Dependență de sistemul de operare: Apelurile de sistem sunt specifice unui anumit sistem de operare. Acest lucru poate crea dependență și poate dificulta portabilitatea aplicațiilor pe alte platforme

https://stackoverflow.com/questions/6424725/why-should-i-minimize-the-use-of-system-call-in-my-code

4) Ce înseamnă user/application mode/space (mod neprivilegiat)? Ce înseamnă kernel/supervisor mode/space (mod privilegiat)?

Users space-ul este nivelul de privilegiu 3 (`CPL`/`CPSR` = 3), în care aplicațiile, procesele rulează.

Kernel space-ul este nivelul de privilegiu 0 (`CPL`/`CPSR` = 0), în care nucleul rulează, asigurându-se de împărțirea corectă a resurselor.

https://stackoverflow.com/a/57926248

Modul neprivilegiat, cunoscut și ca modul utilizator sau spațiul utilizatorului, este o stare de funcționare în care un program rulează cu un set limitat de permisiuni și acces la resursele sistemului. În acest mod, programul rulează într-un mediu izolat și nu are privilegii de a accesa direct sau de a controla resursele hardware sau alte componente critice ale sistemului. Programul este restricționat în a efectua operațiuni care ar putea compromite securitatea sau stabilitatea sistemului.

În modul neprivilegiat, aplicațiile utilizatorului rulează cu resursele și drepturile de acces alocate de sistemul de operare. Ele pot efectua operațiuni specifice, cum ar fi manipularea fișierelor, comunicarea în rețea sau interacțiunea cu interfața grafică, dar nu au acces direct la resursele hardware sau la funcționalități critice ale sistemului.

Pe de altă parte, modul privilegiat, cunoscut și ca modul kernel sau spațiul kernel, este o stare de funcționare în care sistemul de operare și modulele sale esențiale rulează. În acest mod, programul are privilegii de acces extinse și control complet asupra resurselor hardware și a altor componente critice ale sistemului.

Modul privilegiat permite sistemului de operare să execute operațiuni de bază și să controleze funcționalități cheie ale hardware-ului, cum ar fi gestionarea memoriei, planificarea proceselor, gestionarea dispozitivelor și accesul la resursele de sistem. Acest mod asigură protecția și izolarea între diferitele aplicații și permite sistemului de operare să gestioneze resursele într-un mod eficient și sigur.

Diferența fundamentală dintre modul privilegiat și modul neprivilegiat constă în nivelul de acces și control pe care îl are programul asupra resurselor și funcționalităților sistemului. Modul privilegiat este rezervat sistemului de operare și modulelor sale esențiale, în timp ce modul neprivilegiat este utilizat de aplicațiile utilizatorului. Această separare între moduri ajută la menținerea securității și stabilității sistemului.

5) Cum se realizează tranziția din mod neprivilegiat în mod privilegiat?

Tranziția se realizează prin API-ul pus la dispoziție de sistemul de operare (syscall API),

chemat de aplicații prin wrapper-e de bibliotecă sau direct, prin inline assembly.

https://open-education-hub.github.io/operating-systems/Lab/Software%20Stack/basic-syscal

6) Ce se întâmplă în momentul tranziției în mod privilegiat? Cum se/Cine asigură (enforcement) existența modului privilegiat?

Tranziția se realizează prin API-ul pus la dispoziție de sistemul de operare (syscall API),

chemat de aplicații prin wrapper-e de bibliotecă sau direct, prin inline asm.

* Se invocă un wrapper de bibliotecă;

* Funcția pregătește argumentele și mută în `eax` numărul syscall-ului;

* Funcția cheamă un trap;

* Procesorul este mutat în kernel mode;

* Kernel-ul invocă o rutină de syscall;

* Registrele sunt salvate pe stiva din kernel;

* Argumentele sunt verificate pentru a fi valide;

* Acțiunea dorită este lansată.

Astfel, după cum se observă, aplicația din user space trigger-uiește o acțiune care

conduce la implicarea kernel-ului.

Enforcement-ul distincției dintre kernel space și user space este asigurat de modurile de

privilegiu ale procesorului, marcate printr-un bit.

Acesta este salvat în registrul `CPL` pe `x86` sau `CPSR` pe `ARM`.

https://stackoverflow.com/questions/6424725/why-should-i-minimize-the-use-of-system-call-in-my-code

7) Ce este o bibliotecă?

O bibliotecă este o colecție de obiecte, colecție de cod reutilizabil, care oferă funcții, clase, metode sau alte componente software care pot fi utilizate de către alte aplicații. Bibliotecile sunt proiectate pentru a oferi funcționalități comune și sunt adesea integrate în aplicații pentru a extinde funcționalitatea acestora sau pentru a simplifica dezvoltarea software-ului.

8) Care este diferența dintre o aplicație și o bibliotecă?

O aplicație are un entry-point, pe când biblioteca nu. Cu alte cuvinte, biblioteca nu poate fi executată singură. O aplicație, da. diferența principală dintre o aplicație și o bibliotecă constă în scopul lor și modul în care sunt utilizate. O aplicație este un program independent utilizat direct de utilizatori, în timp ce o bibliotecă este o colecție de cod reutilizabil utilizată de dezvoltatori pentru a extinde funcționalitatea aplicațiilor.

9) Care este asocierea apel de bibliotecă / apel de sistem?

Asocierea dintre un apel de bibliotecă și un apel de sistem este aceea că un apel de bibliotecă poate rezulta într-un apel de sistem.

https://open-education-hub.github.io/operating-systems/Lab/Software-Stack/Libcall-Syscall/content/libcall-syscall

10) Ce este entry point-ul într-un executabil?

Entry-point-ul este prima instrucțiune care este pusă în IP (instruction pointer) la momentul lansării executabilului. Spre exemplu, pentru sintaxa NASM, entry-point-ul unui program este by default (pentru `ld` ca linker) simbolul `_start`. entry point-ul este punctul de intrare în codul executabil al unei aplicații sau biblioteci, de unde începe execuția programului. Este primul loc în care sistemul de operare sau mediul de execuție ajunge pentru a iniția execuția programului.

11) Care este rolul bibliotecii standard C (libc)?

`libc` oferă o suită de funcții, atât pentru folosirea integrală în user space (precum `strlen()`), cât și pentru interacțiunea cu kernel-ul (wrapper-e precum `printf()` sau `scanf()`).

12) Ce acțiuni se pot executa doar în mod privilegiat?

* Întreruperi

* Alocarea memoriei

* Operații cu cache-ul

* Interacțiunea cu I/O-ul

https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/

13) Ce operații / instrucțiuni low-level (ISA) se pot executa doar în mod privilegiat?

Un exemplu este instrucțiunea `INT` (cheamă o întrerupere software). Un alt exemplu este `HLT` (oprește procesorul până la apariția unei întreruperi sau a unui semnal).

htt O întrerupere software, cunoscută și sub denumirea de întrerupere generată de software sau întrerupere voluntară, este o formă de întrerupere a execuției programului care este inițiată explicit de către programul însuși, în loc de a fi declanșată de un eveniment extern, cum ar fi o întrerupere hardware sau o întrerupere de ceas (timer interrupt).

O întrerupere software permite programului să întrerupă execuția normală și să se ramifice către o rutină de tratare specifică pentru a realiza anumite operații sau a efectua anumite acțiuni. Această rutină de tratare este denumită și handler de întrerupere software sau serviciu software.

De obicei, întreruperile software sunt folosite pentru următoarele scopuri:

1. Comunicare între procese sau threaduri: Un program poate utiliza întreruperi software pentru a comunica cu alte procese sau threaduri din același sistem. Prin declanșarea unei întreruperi software, programul poate notifica și sincroniza alte entități software despre anumite evenimente sau schimbări de stare.

2. Sisteme de operare și servicii de sistem: Sistemele de operare și alte servicii software pot utiliza întreruperi software pentru a permite programelor să acceseze funcționalități sau resurse specifice oferite de acestea. Prin intermediul întreruperilor software, programul poate solicita servicii de sistem, cum ar fi operații de citire și scriere în fișiere, alocare de memorie, operare cu dispozitive etc.

3. Gestionarea excepțiilor și erorilor: Întreruperile software pot fi utilizate pentru a gestiona excepții și erori în cadrul programelor. Un program poate declanșa o întrerupere software pentru a trata o situație excepțională sau o eroare și pentru a realiza anumite acțiuni, cum ar fi închiderea curată a programului sau afișarea unui mesaj de eroare.

Este important de menționat că întreruperile software sunt diferite de întreruperile hardware. Întreruperile hardware sunt declanșate de evenimente hardware, cum ar fi întreruperi de ceas, întreruperi de dispozitive de intrare/ieșire etc., în timp ce întreruperile software sunt inițiate de

programul însuși.ps://sites.google.com/site/masumzh/articles/x86-architecture-basics/x86-architecture-basi cs?pli=1

14) Ce este un sistem de operare monolitic?

Un sistem de operare monolitic este un sistem de operare în care management-ul

sistemelor de fișiere, a memoriei, a device-urilor și a proceselor cade în atribuțiile kernel-ului.

Linux

https://en.wikipedia.org/wiki/Monolithic_kernel

15) Ce este un sistem de operare de tip microkernel?

Un sistem de operare microkernel este unul ale cărui componente (scheduler-e, sisteme de fișiere, stiva de rețea) rulează ca aplicații în user space.

https://en.wikipedia.org/wiki/Microkernel

16) Care sunt avantajele unui sistem de operare monolitic?

Performanța: toate componentele esențiale se află în același loc, iar comunicare cu

hardware-ul este mediată prin syscall API.

+++ ofera CPU scheduling, managementul memoriei, managementul sistemului de fisiere si

al altor functii ale sistemului de operare prin apeluri de sistem

+++ un singur proces mare ruleaza in intregime intr-un singur spatiu de adrese

+++ este un singur fișier binar static. Exemple de sisteme de operare bazate pe kernel

monolitic sunt Unix, Linux, Open VMS, XTS-400, z/TPF

—- dacă cineva eșuează serviciul duce la o defecțiune a întregului sistem

—- dacă utilizatorul trebuie să adauge vreun serviciu nou. Utilizatorul trebuie să modifice

întregul sistem de operare.

https://www.geeksforgeeks.org/monolithic-kernel-and-key-differences-from-microkernel/

17) Care sunt avantajele unui sistem de operare de tip microkernel?

Securitatea: kernel-ul fiind mult mai mic, suprafața de atac scade; un serviciu de sistem poate fi repornit fără un reboot al întregului sistem de operare. +++ Modularitate: Deoarece nucleul și serverele pot fi dezvoltate și menținute independent, designul microkernel-ului permite o mai mare modularitate. Acest lucru poate facilita adăugarea și eliminarea funcțiilor și serviciilor din sistem. +++ Izolarea defectelor: Designul microkernel-ului ajută la izolarea defecțiunilor și prevenirea afectării întregului sistem. Dacă un server sau o altă componentă eșuează, acesta poate fi repornit sau înlocuit fără a provoca întreruperi restului sistemului.

18) . Care tip de sistem de operare are mai multe apeluri de sistem?

Un sistem de operare de tip monolitic, pentru că invocarea componentelor sale esențiale se face prin syscall; pe când în cazul unui sistem de tip microkernel, comunicarea se realizează în user space, fără a fi nevoie de syscall-urile specifice.

19) Care este avantajul folosirii mașinilor virtuale din perspectiva securității?

* Poți testa tot felul de malware în interiorul mașinii virtuale fără a afecta (ușor) sistemul de operare host. * Un attack de tip hyperjacking(cand vrei sa preiei controlul sistemului) este foarte greu de aplicat, poti lua controlul vm ului dar nu si controlul hostului care o ruleaza. +++MV-urile oferă posibilități extinse pentru analiza și monitorizarea securității. Prin utilizarea instrumentelor de monitorizare a mașinilor virtuale, se pot detecta și investiga comportamente suspecte sau activități anormale în cadrul unei MV, facilitând detectarea timpurie a unor eventuale amenințări sau atacuri. ++++MV-urile permit administrarea și restricționarea resurselor disponibile pentru fiecare mașină virtuală. Acest lucru poate fi folosit pentru a limita impactul unui atac sau a unei aplicații compromise, evitând suprautilizarea resurselor și protejând performanța și stabilitatea celorlalte MV-uri.

https://www.techtarget.com/searchsecurity/tip/Container-vs-VM-security-Which-is-better

20) Ce este o bibliotecă statică? Ce este o bibliotecă dinamică?

O bibliotecă statică este una ale cărei simboluri sunt copiate direct în executabilul programului, la link time. O bibliotecă dinamică este una ale cărei simboluri sunt aduse în executabil la load time, printr-un segment partajat.

O bibliotecă statică și o bibliotecă dinamică sunt două moduri de organizare și utilizare a codului reutilizabil în cadrul unui program.

1. Bibliotecă statică:

- O bibliotecă statică este o colecție de cod compilat și legat direct în executabilul final al unei aplicații. Aceasta conține funcții, rutine sau alte componente software care pot fi apelate de către programul principal.

- La momentul compilării, codul bibliotecii statice este inclus direct în fișierul executabil al aplicației. Astfel, aplicația devine independentă de biblioteca externă și poate fi distribuită ca un singur fișier.

- Avantajul unei biblioteci statice constă în faptul că nu necesită dependențe suplimentare sau instalări separate. Totuși, aceasta duce la o creștere a dimensiunii fișierului executabil, deoarece codul bibliotecii este inclus integral în fiecare aplicație care o utilizează.

2. Bibliotecă dinamică:

- O bibliotecă dinamică (sau bibliotecă partajată) este o colecție de cod compilat și legat separat față de executabilul aplicației. Aceasta este încărcată și utilizată în timpul rulării programului, la cerere.

- În loc să fie inclus în fișierul executabil, referințele către biblioteca dinamică sunt create în timpul compilării și apoi rezolvate în timpul rulării, utilizând un mecanism specific al sistemului de operare. Acest lucru permite ca mai multe aplicații să împărtășească aceeași bibliotecă și să aibă o dimensiune mai mică a fișierelor executabile.

- Avantajul unei biblioteci dinamice este că actualizările sau îmbunătățirile aduse bibliotecii se reflectă automat în toate aplicațiile care o utilizează, fără a fi necesară recompilarea acestora. De asemenea, bibliotecile dinamice permit economisirea resurselor de memorie, deoarece codul bibliotecii este încărcat în memorie doar atunci când este necesar.

Atât bibliotecile statice, cât și cele dinamice au rolul de a facilita reutilizarea codului și de a împărți funcționalități comune între mai multe aplicații. Alegerea dintre cele două depinde de nevoile specifice ale proiectului, considerând aspecte precum dimensiunea executabilului, eficiența în utilizarea resurselor și flexibilitatea în actualizări.

21) Când preferăm folosirea static linking, respectiv dynamic linking? Cu ce diferă un executabil dinamic de un executabil static?

Se preferă folosirea static linking pentru portabilitate, mai exact în cazul în care biblioteca nu există pe sistem, sau a fost modificată. Se preferă folosirea dynamic linking atunci când se dorește folosirea optimă a memoriei și a disk-ului, datorită mecanismului de shared library code și a executabilelor de mărime mică.

https://www.ibm.com/docs/en/aix/7.2?topic=techniques-when-use-dynamic-linking-static-linking

22) Dați exemplu de apel de sistem blocant

`read()` — utilizat pentru citirea datelor dintr-un fișier sau un descriptor de fișier. Acest apel

de sistem blochează execuția programului până când datele sunt citite sau până când apare

o eroare.

`recv()` — utilizat pentru a primi date prin socket-uri în aplicațiile de rețea. Acest apel de

sistem blochează execuția programului până când datele sunt primite sau până când apare

o eroare.

`accept()` — este utilizat pentru a accepta conexiuni pe un socket în aplicațiile de rețea.

Acest apel de sistem blochează execuția programului până când o conexiune este stabilită

sau până când apare o eroare.

23) Dați exemplu de apel de sistem neblocant.

`getpid()` — returnează ID-ul procesului curent și se execută în mod instantaneu, fără a

bloca execuția programului. Nu implică interacțiune cu dispozitive de intrare/ieșire sau

așteptarea unei condiții specifice.

`fcntl() — utilizată pentru a seta descriptorul de fișier în modul non-blocant, așa cum am menționat anterior. Prin utilizarea opțiunii "O_NONBLOCK" în apelul "fcntl()", operațiile de citire sau scriere ulterioare nu vor bloca execuția programului.

"epoll()": Funcția "epoll()" este specifică sistemelor de operare de tip Linux și oferă o alternativă eficientă la "select()" și "poll()". Aceasta utilizează mecanisme de semnalizare asincronă și permite monitorizarea eficientă a descriptorilor de fișiere în mod neblocant.

Funcția `epoll()` este o apel de sistem în Linux, utilizată pentru gestionarea evenimentelor de intrare/ieșire (I/O) asincronă pe un set de descriptori de fișiere. Este un mecanism eficient și scalabil care înlocuiește apelurile tradiționale de sistem, cum ar fi `select()` și `poll()`, oferind performanțe mai bune pentru manipularea unui număr mare de descriptori de fișiere.

`epoll()` este folosită în special în dezvoltarea de aplicații de rețea și în sistemele care necesită monitorizarea și gestionarea concurentă a multiple conexiuni I/O.

Funcția `epoll()` acceptă trei operațiuni principale:

1. `epoll_create()` sau `epoll_create1()`: Aceste apeluri de sistem sunt utilizate pentru a crea un nou obiect epoll, care este un descriptor de fișier special utilizat pentru a stoca informațiile despre descriptorii de fișiere monitorizați.

2. `epoll_ctl()`: Această funcție este utilizată pentru a adăuga, modifica sau șterge descriptori de fișiere din obiectul epoll creat anterior. Prin intermediul acestui apel, se specifică evenimentele de interes pe care programul dorește să le monitorizeze pentru fiecare descriptor de fișier adăugat. De exemplu, se pot specifica evenimente de citire, scriere sau erori.

3. `epoll_wait()`: Această funcție este utilizată pentru a aștepta evenimentele asociate descriptorilor de fișiere monitorizați în obiectul epoll. Apelul `epoll_wait()` este blocant și va fi deblocat numai atunci când unul sau mai multe evenimente de intrare/ieșire au loc pe descriptorii monitorizați. Astfel, aplicația poate să continue execuția și să efectueze alte operații în paralel până când sunt disponibile evenimente de procesat.

Prin utilizarea funcțiilor `epoll()`, programul poate să monitorizeze eficient multiple evenimente de I/O într-un mod asincron și fără a bloca execuția programului în așteptarea acestor evenimente. Acest lucru

permite o manipulare eficientă a resurselor de sistem și o gestionare scalabilă a operațiilor de intrare/ieșire.

24) De ce, în general, o aplicație trebuie să execute un apel de sistem pentru a accesa un dispozitiv hardware? De ce NU poate accesa direct dispozitivul hardware?

Pentru că în sistemele de operare care nu sunt unikernele, rulează în același timp mai multe aplicații. Majoritatea acestora doresc să interacționeze cu hardware-ul, apărând astfel probleme cu accesul concurent la resurse. Sistemul de operare, prin syscall (poate fi privit drept o cerere la un ghișeu), mediază accesul, forțând aplicațiile să nu se calce în picioare.

25) Ce înțelegem prin overhead spațial și overhead temporal?

Creșterea spațiului folosit, respectiv a timpului de execuție, în general, apărută o dată cu

apariția anumitor enforecement-uri.

Overhead-ul spațial și overhead-ul temporal sunt două concepte asociate cu costurile suplimentare

sau resursele necesare pentru a implementa o anumită funcționalitate sau abordare într-un sistem.

26) Dați exemplu de mecanism / funcție care reduce overhead-ul spațial și unul care reduce overhead-ul temporal.

Mecanismul de shared memory reduce overhead-ul spațial, prin accesarea directă a buffer-ului din kernel space.

Mecanismul de zero copy reduce overhead-ul temporal, prin evitarea domain switch-urilor

Mai avem si Memmove() care copiaza datele eficient chiar daca acestea se suprapun. El reduce overhead-ul temporal. Se evita pierderea timpului pe copier de date redundante

Mecanismul zero-copy (fără copiere) este o tehnică utilizată în sistemele de operare și în aplicațiile software pentru a eficientiza transferul de date între diferite componente sau straturi de software, fără a implica copierea inutilă a datelor între acestea. Mecanismul zero-copy vizează reducerea latenței și a utilizării CPU și a memoriei, oferind performanțe îmbunătățite.

În mod tradițional, când datele trebuie să fie transferate între diferite componente, cum ar fi între aplicații, între un driver de rețea și memoria principală sau între două straturi de protocol, datele sunt copiate de la sursă la destinație. Acest proces de copiere implică mișcarea datelor de la o zonă de memorie la alta, ceea ce consumă timp și resurse.

Cu mecanismul zero-copy, se evită copierea inutilă a datelor prin permiterea accesului direct la datele existente în memoria sursă, fără a fi necesară copierea lor într-o zonă intermediară. În loc să se copieze datele într-un buffer temporar, informațiile despre locația datelor sunt transferate direct între componentele implicate.

Există mai multe modalități de implementare a mecanismului zero-copy, inclusiv:

1. Direct Memory Access (DMA): În acest caz, componenta care deține datele poate utiliza DMA pentru a transfera datele direct într-o zonă specifică a memoriei destinatarului, evitând astfel copierea prin intermediul procesorului.

2. Mapare directă în memorie (Memory Mapping): Acesta implică partajarea unei regiuni de memorie între componentele implicate. Aceasta permite accesul direct la datele existente în memoria sursă de către destinatar, fără a implica copierea.

3. Pointeri și referințe: În unele cazuri, componentele pot utiliza pointeri sau referințe pentru a face referire directă la datele existente, evitând copierea lor într-un alt buffer.

Beneficiile mecanismului zero-copy includ:

- Reducerea utilizării CPU și a latenței: Deoarece nu este necesară copierea datelor, mecanismul zero-copy reduce sarcina CPU și permite transferul mai rapid al datelor, reducând latența.

- Economisirea resurselor de memorie: Prin evitarea copierii datelor în multiple zone de memorie, se economisește spațiul de memorie, permițând o utilizare mai eficientă a resurselor.

Este important de menționat că implementarea mecanismului zero-copy poate fi complexă și depinde de suportul hardware și de sistemul de operare utilizat. De asemenea, trebuie luate în considerare aspecte de securitate și sincronizare pentru a asigura integritatea și consistența datelor în timpul transferului.

27) Ce înseamnă double buffering? În ce situație concretă (mecanism/funcție) apare?

Double buffering se referă la prezența a două buffere: unul în user space și altul în kernel space. Apare adesea la funcții precum cele de read și write.

https://www.geeksforgeeks.org/double-buffering/

28) . Ce se întâmplă când există o eroare critică (de tip Segmentation fault) la nivelul sistemului de operare?

Dacă în nucleu se ajunge la Segmentation Fault, atunci sistemul de operare intră în kernel panic. (eroare de computer din care OS-ul sistemului nu se poate recupera rapid sau ușor si apare atunci când există o eroare fatală de nivel scăzut și kernel-ul sistemului de operare nu o poate remedia.)
https://en.wikipedia.org/wiki/Kernel_panic

29) Dați exemplu de un apel de bibliotecă standard C care nu cauzează apel de system

`strlen()` – calcula lungimea unui șir de caractere `abs()` — utilizată pentru a calcula valoarea absolută a unui număr întreg. Aceasta returnează valoarea absolută a numărului fără a avea nevoie să interacționeze cu sistemul de operare sau să efectueze operații de intrare/ieșire

30) Dați exemplu de un apel de bibliotecă standard C care cauzează apel de sistem.

`scanf()` `printf`

Read(), write()

31) Care este avantajul și dezavantajul existenței unei stive software la nivelul unui sistem de calcul?

Avantaj: izolare a aplicațiilor bună, portabilitate, modularitate

Dezavantaj: overhead temporal

O stivă software (sau stack) este o zonă de memorie utilizată în programare pentru a stoca informații despre execuția programului, cum ar fi variabile locale, parametri de funcție și adrese de retur. Este o structură de date de tip LIFO (Last-In, First-Out), ceea ce înseamnă că ultimul element adăugat pe stivă este primul element scos de pe stivă.

Rolul principal al stivei software este de a gestiona execuția programului prin stocarea și restabilirea contextului de execuție al funcțiilor și procedurilor. Pe măsură ce funcțiile sunt apelate în cadrul programului, informațiile specifice ale fiecărei funcții, cum ar fi variabilele locale, parametrii și adresa de retur, sunt adăugate pe stivă.

Când o funcție se termină de executat, informațiile sale sunt eliminate de pe stivă și controlul este returnat către funcția apelantă. Aceasta se realizează prin operația de "pop" (sau "unwind") care extrage informațiile de pe stivă și restabilește contextul funcției apelante.

Rolul stivei este crucial în realizarea execuției secvențiale și gestionarea apelurilor de funcții în programare. Principalele funcții ale stivei software includ:

1. Gestionarea contextului funcțiilor: Stiva este responsabilă de păstrarea și restaurarea contextului funcțiilor, inclusiv variabilele locale și adresele de retur, în timpul apelurilor și terminării funcțiilor.

2. Gestionarea apelurilor de funcții: Stiva permite apelul și returnarea secvențială a funcțiilor în cadrul programului, asigurând ordinea corectă a execuției.

3. Gestionarea stocării temporare: Stiva poate fi utilizată pentru a stoca temporar valori sau date în cadrul execuției programului, precum parametrii de funcție sau alte informații relevante pentru execuție.

4. Gestionarea depășirilor de stivă (stack overflow): O depășire de stivă apare atunci când stiva este umplută cu mai multe date decât poate gestiona. Aceasta poate duce la erori de execuție și întreruperea programului.

Stiva software este o componentă esențială în organizarea execuției programului și asigurarea corectitudinii și eficienței în apelurile de funcții și gestionarea contextului. Este gestionată automat de către compilatoare și de sistemul de operare în timpul execuției programului.

32) Care este avantajul și dezavantajul folosirii unui limbaj de programare high-level (precum Java, Python, Lua)?

Avantaj: management automat al memoriei (scapi de dureri de cap)

Dezavantaj: overhead temporal, pierderea controlului

33) Care este avantajul și dezavantajul folosirii unui limbaj de programare low-level (precum C, C++, Rust, D)?

Avantaj: control (faci ce vrea mușchiul tău), rapiditate

Dezavantaj: prone to mistakes, mai greu de portat

Data Q&A

1) Cum asigură sistemul de operare separația între procese?

Sistemul de operare separă procesele prin mecanismul de memorie virtuală (de paginare, acces granulat la RAM).

Sistemul de operare asigură separația între procese prin utilizarea unor mecanisme și tehnici de izolare și protecție. Aceste mecanisme sunt concepute pentru a preveni interferențele între procese și pentru a asigura că fiecare proces rulează într-un mediu izolat și protejat. Iată câteva mecanisme cheie utilizate în acest scop:

1. Spațiul de adresă virtual: Fiecare proces are propriul său spațiu de adresă virtual, care este o zonă de memorie izolată și privată în care acesta își stochează instrucțiunile și datele. Spațiul de adresă virtual permite fiecărui proces să opereze cu adrese de memorie virtuale care nu se suprapun cu cele ale altor procese, asigurând astfel izolarea și protecția datelor între procese.

2. Modele de acces la memorie: Sistemul de operare utilizează permisiuni și protecție bazată pe paginare pentru a controla accesul fiecărui proces la memoria fizică. Prin intermediul paginării, fiecare pagină de memorie poate fi marcată ca fiind accesibilă doar pentru citire, scriere sau execuție, și doar procesele autorizate pot accesa acele pagini.

3. Controlul accesului la resurse: Sistemul de operare gestionează accesul la resursele hardware și la alte resurse critice, cum ar fi dispozitivele de stocare sau interfețele de rețea. Accesul la aceste resurse este restricționat prin intermediul permisiunilor și drepturilor de acces atribuite fiecărui proces, astfel încât doar procesele autorizate să poată interacționa cu resursele respective.

4. Planificarea proceselor: Sistemul de operare utilizează planificatoare pentru a aloca resursele de procesor între diferitele procese. Prin intermediul planificării, fiecare proces primește o parte a timpului de procesor și nu poate interfera cu execuția altor procese. Astfel, procesele rulează în mod concurent și izolat, având impresia că au acces exclusiv la resursele de procesor.

5. Mecanisme de comunicare controlată: Sistemul de operare oferă mecanisme de comunicare între procese, cum ar fi pipe-uri, cozi de mesaje sau semafoare. Aceste mecanisme permit proceselor să comunice și să coopereze într-un mod controlat, fără a permite accesul neautorizat la datele sau resursele altor procese.

Aceste mecanisme combinate asigură separația între procese și protejează integritatea și confidențialitatea datelor. Ele sunt implementate de sistemul de operare pentru a asigura un mediu sigur și eficient pentru execuția proceselor într-un sistem informatic.

2) Ce înseamnă mecanismul de memorie virtuală?

Memoria virtuală este modalitatea prin care kernel-ul oferă proceselor impresia că dețin întreaga memorie, când lucrurile nu stau chiar așa. Adresele de memorie virtuală ajung să fie corelate cu unele fizice de abia la acces, ceea ce permite împărțirea corectă a RAM-ului.

Memoria virtuală este o tehnică utilizată de sistemele de operare pentru a gestiona eficient memoria fizică disponibilă într-un sistem informatic. Memoria virtuală extinde capacitatea de stocare a sistemului prin utilizarea unui spațiu de adresă virtual mai mare decât memoria fizică fizică disponibilă. Aceasta permite executarea simultană a unui număr mare de procese, depășind limitările memoriei fizice disponibile.

În esență, memoria virtuală separă spațiul de adresă vizibil pentru un proces, cunoscut sub numele de spațiu de adresă virtual, de spațiul de adresă fizic, care este memoria fizică reală. Astfel, fiecare proces are impresia că dispune de o memorie completă, dedicată și continuă, chiar dacă memoria fizică este limitată.

Pentru a implementa memoria virtuală, sistemul de operare utilizează un mecanism numit paginare. Memoria virtuală este împărțită în pagini mici de dimensiuni fixe. La nivel fizic, memoria fizică este împărțită în pagini echivalente. În timpul execuției, paginile virtuale sunt mapate pe paginile fizice corespunzătoare prin intermediul tabelelor de pagini. Aceste tabele conțin informații despre asocierea dintre adresele virtuale și adresele fizice.

Principalele avantaje ale memoriei virtuale sunt:

1. Abstracție a memoriei: Memoria virtuală oferă o abstracție pentru programe, permițându-le să opereze cu adrese de memorie virtuale. Aceasta facilitează dezvoltarea programelor, deoarece programatorii nu trebuie să gestioneze manual adresele de memorie fizică.

2. Izolare și protecție: Fiecare proces rulează în propriul său spațiu de adresă virtual, izolat de alte procese. Astfel, un proces nu poate accesa direct memoria altui proces, oferind protecție împotriva interferențelor și îmbunătățind securitatea sistemului.

3. Gestiune eficientă a memoriei: Memoria virtuală permite sistemei de operare să gestioneze eficient memoria fizică prin intermediul algoritmelor de înlocuire a paginilor. În cazul în care toate paginile necesare unui proces nu încap în memoria fizică, sistemul de operare poate transfera anumite pagini pe disc și le poate aduce înapoi în memorie când acestea sunt necesare. Acest lucru permite executarea simultană a unui număr mare de procese, chiar dacă memoria fizică este limitată.

Memoria virtuală este o caracteristică esențială în sistemele de operare moderne și joacă un rol crucial în gestionarea resurselor de memorie într-un mod eficient și flexibil.

3) Ce reprezintă spațiul virtual de adrese al unui proces?

Spatiul virtual de adrese este o înlănțuire de zone (segmente) folosită pentru a reține ce memorie procesul dorește să rezerve sau să elibereze.

Conține tabela de pagini a procesului, care reține corespondența dintre adresele virtuale și cele fizice.

https://en.wikipedia.org/wiki/Virtual_address_space

4) Cu ce diferă zona de date de zona de cod în spațiul virtual de adrese al unui proces?

Cele două zone diferă, practic, prin permisiunile lor: zona de date e RW-, pe când zona de cod e R-X, iar teoretic prin funcțiile lor: zona de date reține, în general, variabilele folosite de program, pe când zona de cod reține instrucțiunile care se execută.

5) Ce permisiuni are zona .text/.data/.rodata/.bss/de stivă/de heap?

.text: R-X

.data: RW-

.rodata: R--

.bss: RW-

stiva: RW-

heap: RW-

.text -> codul care este de executat

.data -> variabilelele unui program, date de intrare intr-un fisier, etc

.rodata -> doar date ce nu pot fi modificate, doar citite

.bss -> aici intra variabilele globale dintr-un program

.stiva -> parametrii unei functii, return address, variabilele locale

.heap -> alocarea cu malloc, calloc, etc

6) Ce este paginarea memoriei?

Paginarea memoriei este mecanismul prin care RAM-ul este împărțit în unități de o anumită dimensiune (ex: 4096 pentru Linux). Aceste unități sunt apoi folosite ca metrică pentru alocarea și rezervarea memoriei, menținând accesul granulat la aceasta.

7) Ce rol are tabela de pagini?

Tabela de pagini reține corespondența (map-area) dintre adresa virtuală (indexul din vectorul de pagini) și adresa fizică (valoarea reținută în vector).

8) Ce este și ce rol are MMU (Memory Management Unit)?

MMU-ul (sau MTU pe `ARM`) are rolul de a translata adresa de memorie virtuală în fizică. El consultă PFN(page frame number)-ul, permisiunile și bitul de validitate pentru a determina ce face mai departe: generează un page fault major sau minor.

https://developer.arm.com/documentation/101811/0102/The-Memory-Management-Unit--MM U-

MMU (Memory Management Unit) este o componentă hardware a unui sistem de calcul, responsabilă de gestionarea și traducerea adreselor virtuale generate de procesor în adrese fizice corespunzătoare

din memoria fizică. MMU funcționează în colaborare cu sistemul de operare pentru a realiza gestionarea eficientă și protecția memoriei într-un sistem informatic.

Iată câteva aspecte despre rolul și funcționalitatea MMU:

1. Traducerea adreselor virtuale în adrese fizice: Procesoarele utilizează adrese virtuale generate de programele utilizator, care sunt adrese logice sau simbolice utilizate în cadrul spațiului de adresă virtual al procesului. MMU preia aceste adrese virtuale și le traduce în adrese fizice corespunzătoare în memoria fizică, care reprezintă adresele reale ale celulelor de memorie.

2. Protecție și izolare a memoriei: MMU facilitează implementarea mecanismelor de protecție și izolare a memoriei. Prin intermediul tabelelor de pagini și a atributelor asociate, MMU permite sistemului de operare să stabilească permisiuni de acces specifice pentru diferite regiuni de memorie, protejând astfel un proces împotriva accesului neautorizat la alte regiuni de memorie sau la spațiul de adresă al altor procese.

3. Gestionarea memoriei virtuale: MMU permite implementarea memoriei virtuale, care extinde capacitatea sistemului de operare prin alocarea spațiului de adresă virtual al proceselor într-o manieră mai mare decât memoria fizică disponibilă. MMU gestionează încărcarea și descărcarea paginilor în și din memoria fizică, utilizând mecanisme de înlocuire a paginilor pentru a aduce în memorie doar paginile necesare pentru execuția curentă a proceselor.

4. Paginare și cache: MMU lucrează în colaborare cu mecanismele de paginare ale sistemului de operare pentru a realiza împărțirea memoriei virtuale și fizice în pagini de dimensiuni fixe. De asemenea, MMU poate include și funcționalitatea de cache, permițând stocarea temporară a datelor accesate frecvent într-un cache de nivelul doi (L2 cache), ceea ce duce la îmbunătățirea performanței accesului la memorie.

5. Asigurarea coerenței cache-ului: În sistemele multiprocesor, MMU joacă un rol important în asigurarea coerenței cache-ului între diferitele nuclee de procesor. Acesta gestionează protocolul de coerență a cache-ului și asigură că datele modificate într-un cache sunt propagate corespunzător și reflectate în celelalte cache-uri.

MMU reprezintă un element cheie în sistemul de gestionare a memoriei al unui sistem de operare. Prin intermediul MMU, se realizează traducerea eficientă a adreselor virtuale în adrese fizice, se as

igură protecția și izolarea memoriei, se permite gestionarea memoriei virtuale și se optimizează accesul la memorie, contribuind la performanța și securitatea sistemului informatic.

9) Ce rol are TLB? — Translation-Lookaside-Buffer

TLB-ul este un cache pentru intrările din tabela de pagini a procesului curent. Acesta reține PTE (Page Table Entry) -urilefolosite recent, și pentru fiecare adresă virtuală, verifică dacă are adresa fizică memorată. Dacă da, atunci are loc un TLB hit, altfel rezultă un TLB miss.

https://developer.arm.com/documentation/101811/0102/Translation-Lookaside-Buffer-mainte nance

10) Care este ordinul de mărime al numărului de intrări ale TLB?

Intre 16 si 512 intrari

11) Ce conține o intrare în tabela de pagini?

PFN (page frame number)

permisiuni

bit de validitate

un dirty bit.

O intrare în tabela de pagini (Page Table Entry - PTE) conține informații specifice despre o pagină virtuală în cadrul unui sistem de gestionare a memoriei cu paginare. Această tabelă este utilizată de Memory Management Unit (MMU) pentru a realiza traducerea adresei virtuale în adresa fizică corespunzătoare.

Conținutul exact al unei intrări în tabela de pagini poate varia în funcție de arhitectura sistemului de operare și de implementarea specifică a MMU, dar în general, o intrare în tabela de pagini poate include următoarele informații:

1. Bitul de prezență (Present Bit): Acest bit indică dacă pagina virtuală este prezentă în memoria fizică sau nu. Dacă bitul de prezență este setat, înseamnă că pagina este prezentă în memoria fizică și poate fi accesată. Dacă este 0, poate indica faptul că pagina nu este în memorie și trebuie încărcată înainte de a fi accesată.

2. Adresa fizică (Physical Address): Aceasta reprezintă adresa fizică la care pagina corespunzătoare este stocată în memoria fizică. Atunci când o pagină este prezentă în memorie fizică, adresa fizică indică locația sa în memorie.

3. Bitul de scriere (Write Bit): Acest bit indică dacă pagina este protejată în mod read-only sau dacă poate fi accesată și modificată în mod read-write. Dacă bitul de scriere este setat, pagina poate fi modificată. În caz contrar, accesul la pagină este doar în mod citire.

4. Bitul de acces (Accessed Bit): Acest bit indică dacă pagina a fost accesată sau nu. MMU-ul poate seta acest bit automat atunci când se face un acces la pagină și poate fi utilizat pentru a implementa politici de înlocuire a paginilor (paging replacement policies).

5. Bitul de modificare (Dirty Bit): Acest bit indică dacă pagina a fost modificată (scrisă) de când a fost încărcată în memorie fizică. Acest bit este util pentru implementarea politicii de scriere în memorie fizică și poate fi utilizat pentru a gestiona operațiile de scriere.

6. Bitul de protecție (Protection Bit): Acest bit poate indica nivelul de protecție și permisiuni asociate paginii. Poate specifica dacă pagina poate fi accesată doar de procesul curent, dacă poate fi partajată între mai multe procese sau alte politici de acces.

Acestea sunt doar câteva exemple de informații comune găsite într-o intrare în tabela de pagini. Este important de menționat că conținutul exact al unei intrări poate varia în funcție de arhitectura și implementarea specifică a sistemului de operare și a MMU-ului.

12) Ce înseamnă tabelă de pagini multi-nivel (ierarhică)? De ce este utilă?

Este o tabelă de pagini care ca și informație reține adrese ale altor tabele de pagini. Intrările ultimului nivel de tabele de pagini rețin informația legată de frame-uri. Primul nivel conține o singură tabelă de pagini și adresa acesteia este reținută în PTBR.

https://www.geeksforgeeks.org/multilevel-paging-in-operating-system/

Existența tabelei de pagini ierarhice într-un sistem de gestionare a memoriei cu paginare este motivată de câteva avantaje și considerații de performanță. Iată câteva motive pentru utilizarea tabelei de pagini ierarhice în locul unei tabele de pagini plate (simplă):

1. Reducerea dimensiunii tabelei de pagini: O tabelă de pagini plată necesită alocarea unui spațiu continuu de memorie pentru a stoca toate intrările pentru fiecare pagină virtuală. Acest lucru poate deveni problematic în cazul în care spațiul de adrese virtuale este foarte mare. Prin utilizarea tabelei de pagini ierarhice, dimensiunea totală a tabelei de pagini poate fi redusă prin împărțirea acesteia în mai multe niveluri sau structuri ierarhice.

2. Eficiență în gestionarea memoriei fizice: O tabelă de pagini ierarhice permite o mai bună gestionare a memoriei fizice. În loc să aibă toate intrările într-o singură tabelă, paginile virtuale pot fi grupate într-un număr mai mic de tabele de nivel superior. Aceasta reduce numărul de intrări care trebuie examinate pentru a găsi o pagină fizică corespunzătoare. În acest fel, se reduce complexitatea și timpul de căutare în cazul înlocuirii paginilor și în gestionarea memoriei fizice.

3. Eficiență în gestionarea memoriei virtuale: O tabelă de pagini ierarhice permite o mai bună gestionare a memoriei virtuale, deoarece permite crearea de subtabele sau substructuri în funcție de nevoile efective de memorie ale procesului. Astfel, este posibilă alocarea și eliberarea memoriei virtuale în mod flexibil și eficient, fără alocarea inutilă a unei cantități mari de spațiu pentru paginile neutilizate.

4. Protecție și izolare: Prin utilizarea tabelei de pagini ierarhice, este posibilă aplicarea diferitelor niveluri de protecție și permisiuni pentru diferite subtabele sau niveluri. Aceasta permite izolarea și protecția datelor și codului între procese sau în interiorul unui proces.

5. Fragmentare redusă: Tabelele de pagini ierarhice pot ajuta la reducerea fragmentării memoriei virtuale și fizice. Prin organizarea paginilor în structuri ierarhice, este mai puțin probabil ca paginile alocate și eliberate să conducă la fragmentarea excesivă a memoriei.

În concluzie, utilizarea unei tabele de pagini ierarhice aduce avantaje în gestionarea memoriei virtuale și fizice, reducând dimensiunea și complexitatea tabelei de pagini și îmbunătățind eficiența în accesarea și gestionarea paginilor.

13) Când are loc un TLB miss?

Un TLB miss are loc atunci când pagina pe care se află adresa accesată nu a mai fost folosită, a fost folosită cu mult timp și a fost eliminată la update-ul TLB-ului sau când efectiv nu există o adresă fizică alocată pentru cea virtuală.

14) De ce se golește TLB-ul (TLB flush) la schimbare de context?

Pentru ca TLB-ul să rețină informația procesului curent, nu pe cea a procesului precendent. Nu este permis, astfel, accesul proceselor la informație ce nu ar trebui să fie partajată

15) De ce nu este nevoie de TLB flush la schimbarea de context între două thread-uri ale aceluiași proces?

Pentru că thread-urile aceluiași proces share-uiesc tabela sa de pagini, deci folosesc și au voie să acceseze aceleași adrese fizice.

16) Ce înseamnă mecanismul de copy-on-write?

Mecanismul de copy-on-write este modalitatea prin care procesele copil primesc corespondența pagini - frames. Inițial, spațiul virtual de adrese al procesului copil point-ează către același spațiu fizic de adrese. Permisiunile sunt schimbate pe read only, și de abia la scriere, se duplică frame-urile, iar permisiunile sunt updatate.

17) Dați exemple de situații în care are loc mecanismul de copy-on-write.

În management-ul memoriei virtuale: la `fork()`.

În biblioteci: clasa de string-uri din C++ (pentru standardul C++98)

18) Cu ce apel de sistem asociem copy-on-write?

`fork()`

19) Când se duplică o pagină marcată copy-on-write?

La scriere.

20) Cine detectează un acces de scriere într-o pagină marcată copy-on-write?

Kernel-ul interceptează încercarea de a scrie pe o pagină marcată COW.

Acesta alocă un nou frame (inițializat cu datele de pe pagina COW), face update la tabela

de pagini cu noua corespondență și decrementează numărul de referințe către pagina COW

care a cauzat intervenția sa.

COW -> Copy On Write

https://lwn.net/Articles/849638/

21) Ce înseamnă demand paging?

Demand paging este mecanismul prin care o pagină virtuală primește un corespondent fizic (un frame) de abia la momentul folosirii ei (la acces).

22) În ce situație apare page fault fără a cauza segmentation fault?

— la copy on write sau la demand paging

La primul acces al unei pagini ce nu a fost încă map-ată. Aceasta este rezervată (apare în VAS), dar nu are încă asociată un frame.

https://www.kernel.org/doc/gorman/html/understand/understand007.html

23) Ce rol are spațiul de swap?

Spațiul de swap are rolul de a reține paginile inactive din RAM, pentru a se putea face loc pentru altele noi, când memoria fizică este plină.

24) Când are loc swap in și swap out?

Swap out are loc atunci când spațiul fizic devine insuficient. Swap in are loc la un swap out.
https://www.geeksforgeeks.org/swapping-in-operating-system/

Swap in și swap out sunt două termeni utilizați în sistemele de operare pentru a descrie procesul de transfer al paginilor de memorie între memoria fizică (RAM) și spațiul de swap, care este o zonă de stocare pe disc utilizată pentru extinderea capacității de memorie a sistemului.

Swap out (sau page out) reprezintă procesul de transfer al paginilor de memorie din RAM către spațiul de swap. Acest proces are loc atunci când sistemul de operare are nevoie de mai multă memorie fizică pentru a gestiona toate procesele aflate în execuție. Paginile care nu sunt active sau care nu au fost utilizate recent sunt selectate și transferate în spațiul de swap pentru a face loc altor pagini necesare în RAM. Swap out reduce consumul de memorie fizică, dar poate introduce un cost suplimentar de performanță din cauza timpului necesar pentru transferul datelor între RAM și spațiul de swap pe disc.

Swap in (sau page in) reprezintă procesul invers, în care paginile de memorie care au fost transferate în spațiul de swap sunt aduse înapoi în memoria fizică (RAM) atunci când sunt necesare pentru a fi utilizate de către procese. Aceasta poate fi declanșată, de exemplu, atunci când un proces încearcă să acceseze o pagină care se află în spațiul de swap. În acest caz, acea pagină este adusă înapoi în RAM prin transferul ei din spațiul de swap. Procesul de swap in poate afecta performanța, deoarece implicațiile I/O (intrare/ieșire) sunt mai lente decât accesul direct la memoria RAM.

Swap in și swap out sunt mecanisme importante în gestionarea memoriei virtuale a sistemelor de operare. Ele permit extinderea capacității de memorie prin utilizarea spațiului de swap pe disc și asigură că procesele pot accesa datele necesare chiar și atunci când memoria fizică este limitată. Cu toate acestea, utilizarea excesivă a swap-ului poate afecta performanța, deoarece accesul la date de pe disc este mult mai lent decât accesul la datele din memoria RAM.

25) Care este rolul unui page fault? În ce condiții apare?

Page fault-ul este o excepție ridicată de MMU, care apare atunci când un proces încearcă să acceseze o pagină fără a împlini anumite condiții preliminarii. Are rolul de a semnala shared memory, demand paging-ul sau un posibil acces invalid, împărțind-se în minor, major sau invalid.
https://en.wikipedia.org/wiki/Page_fault

Un "page fault" (eroare de pagină) apare într-un sistem de operare atunci când o pagină de memorie solicitată nu este prezentă în memoria fizică (RAM) și trebuie să fie adusă din spațiul de swap sau să fie alocată o pagină nouă în memorie pentru acea referință.

Există două tipuri principale de page faults:

1. Page fault de tip "minor" (soft page fault): Acest tip de page fault apare atunci când pagina cerută este prezentă în memoria virtuală a procesului, dar nu este încă încărcată în memoria fizică. În acest caz, sistemul de operare încarcă pagina din spațiul de swap în memoria RAM și actualizează tabelele de pagini corespunzătoare. Acest tip de page fault are un impact relativ mic asupra performanței, deoarece nu implică schimbarea în contextul procesului și are un cost redus în timp.

2. Page fault de tip "major" (hard page fault): Acest tip de page fault apare atunci când pagina cerută nu este disponibilă nici în memoria virtuală, nici în memoria fizică. Acesta implică o operație de I/O (intrare/ieșire) pentru a citi pagina de pe disc din spațiul de swap sau de alocare a unei pagini noi în memoria fizică. Page fault-urile majore pot avea un impact semnificativ asupra performanței, deoarece implică un timp mai mare de acces la disc și de transfer de date.

Gravitatea fiecărui caz de page fault depinde de mai mulți factori, cum ar fi frecvența și tipul page fault-urilor, dimensiunea și timpul de acces la spațiul de swap, cantitatea de memorie disponibilă, designul algoritmului de înlocuire a paginilor etc. În general, page fault-urile minore sunt considerate mai puțin grave, deoarece implică doar transferul paginii din memoria virtuală în memoria fizică, în timp ce page fault-urile majore pot introduce o penalizare semnificativă de performanță din cauza timpului mai lung de acces la disc.

O cantitate excesivă de page fault-uri majore sau o gestionare ineficientă a acestora poate duce la fenomenul de "thrashing", în care sistemul de operare petrece o cantitate semnificativă de timp în transferul continuu al paginilor între memoria fizică și spațiul de swap, afectând în mod negativ performanța generală a sistemului.

Pentru a gestiona și minimiza impactul page fault-urilor, sistemele de operare utilizează diferite tehnici, cum ar fi algoritmi de înlocuire a paginilor (precum LRU - Least Recently Used), algoritmi de prefetching, utilizarea unui cache de pagini, strategii de alocare a resurselor și optimizări ale algoritmilor de planificare a proceselor.

26) Care sunt secțiunile/zonele din spațiul de adrese al unui proces?

.text

.data

.rodata

.bss

stiva

heap

biblioteci dinamice

kernel

27) Ce secțiuni ale unui executabil se pot inspecta doar în timpul rulării?

Stiva, heap-ul și bibliotecile dinamice.

28) Care sunt zonele writable din spațiul de adrese al unui proces?

.data, .bss, stiva șî heap-ul

29) De ce sunt avantajoase bibliotecile dinamice pentru spațiul de adrese al unui proces?

Pentru că sunt încărcate frame cu frame o singură dată, fiind folosite de mai multe procese în același timp.

30) Două procese sunt pornite din același executabil, ce zone din spațiul de adrese vor partaja?

Bibliotecile dinamice, kernel space, .text și .rodata.

31) Se alocă un buffer a[100]. De ce a[105] NU va rezulta, în general, în Segmentation fault?

Pentru că, în general, se alocă o pagină întreagă, iar elementul 105 se află în ea.

32) În ce situație a[300] rezultă în Segmentation fault?

Atunci când lui a[100] îi este alocat spațiu la finalul unei pagini, iar a[300] ar intra în zona unei pagini nealocate încă.

33) Câte pagini ocupa char a[100] pentru sistem standard cu pagini de 4096 de octeți?

Cel puțin una, cel mult două.

34) Câte pagini fizice alocă un apel mmap() care alocă 1MB? O pagină ocupă 4KB

0, `mmap()` nu alocă, ci rezervă spațiu virtual

35) Câte pagini fizice alocă un apel calloc() care alocă 1MB? O pagină ocupă 4KB.

Cel puțin 250, cel mult 251.

36) Ce înseamnă maparea unui fișier în memorie? De ce este avantajos să mapăm fișiere față de folosirea read/write?

Fișierele map-ate în memorie prezintă un mecanism prin care procesele acesează fișiere prin incorporarea lor directă în VAS. Astfel, se reduce în mod semnificativ I/O data movement (și overhead-ul temporal), întrucât datele fișierului nu mai trebuie copiate în buffer-ele procesului așa cum s-ar întâmpla în cazul funcțiilor `read()` și `write()`.

https://www.ibm.com/docs/en/aix/7.2?topic=memory-understanding-mapping

37) Câte page fault-uri se pot obține în cazul operației *a = b?

4: unul pentru dereferențiere, unul pentru a, unul pentru operația în sine, dacă nu a fost map-ată pagina de .text aferentă și unul pentru b.

38) Care este numărul maxim de page fault-uri pe care îl poate genera expresia a = b + c?

5: unul pentru a, unul pentru pentru operația de atribuire în sine, dacă nu a fost map-ată pagina de .text aferentă, unul pentru b, unul pentru c, unul pentru operația de adunare, dacă nu a fost map-ată pagina de .text aferentă.

39) Ce informații sunt reținute în stivă? Ce variabile C?

variabilele locale funcției, adresa de return, stack-frame-ul, parametrii funcției.

40) Ce se întâmplă la faza de loading (încărcarea unui executabil în memorie și crearea unui proces)?

EXEC genereaza loading

În timpul fazei de loading, sistemul de operare citește executabilul de pe disk și map-ează

părți din el în RAM.

Tot acum se translatează adresele logice în adrese fizice.

https://en.wikipedia.org/wiki/Loader_(computing)

Se face si shimbare de context, se incarca bibliotecile statice

41) Ce înseamnă deturnarea fluxului de execuție a unui program (control flow hijack)? De ce este acest lucru relevant pentru un atacator?

Cotrol-flow hijack este un tip de atac care urmărește schimbarea flow-ului normal al aplicației (injectarea IP/PC), și executarea de cod fraudulos prin vulnerabilități precum buffer overflow, index out of bounds, integer overflow sau string formatting. Este relevant pentru că scopul final al unui atac de tip control-flow hijack este preluarea controlului (pornirea unui nou shell, preluarea controlului unui server web, etc).

42) Ce înseamnă memory leak / memory disclosure? De ce este acest lucru relevant pentru un atacator?

Memory disclosure este un tip de atac care urmărește descoperirea unor informații sensibile, neautorizate. Mecanisme hardware precum branch prediction sau speculative execution și vulnerabilități ca memory dump, memory reuse sau cross-use au condus la atacuri celebre (Spectre, Meltdown) care au afectat miliarde de procesoare.

43) De ce în general, preferăm o împărțire a spațiului virtual de adrese între kernel space și user space? Și nu un spațiu dedicat pentru kernel space?

Pentru că kernel space-ul este la rândul lui specific fiecărui proces, iar separarea celor două spații asigură protecția memoriei și a hardware-ului împotriva software-ului malițios sau eronat.
https://unix.stackexchange.com/questions/472223/whats-the-use-of-having-a-kernel-part-in-t he-virtual-memory-space-of-linux-proc

44) Ce secvență de cod C va duce la o excepție de acces la memorie (de tip Segmentation fault)? De ce?

Dereferențierea unui pointer null. Acesta este reprezentat ca o referință către adresa 0 din adress space, iar, prin urmare, MMU-ul indică ca nefăcând parte din memorie pagina care o conține. Pagina nu este, astfel, inclusă în VAS, iar o posibilă scriere sau citire pe aceasta rezultă într-un acces invalid și un SEGFAULT.

45) Cu ce diferă o funcție de o variabilă într-un executabil și/sau în cadrul spațiului de adrese al unui proces?

O variabilă nu este executabilă, ea se află în zona .data, .rodata, .bss sau stivă. O funcție este un simbol ce poate fi executat. Ea se află în zona .text și are permisiuni de citire și execuție.

46) Două procese partajează o zonă de memorie. Cum se manifestă acest lucru în tabelele de pagini ale celor două procese?

Cele două tabele de pagini vor avea anumite intrări comune, reprezentând adresele fzice pe care cele două procese le folosesc concomitent.

47) Putem avea mai multă memorie fizică decât dimensiunea maximă a spațiului virtual de adrese al unui proces? Dar invers?

În general, memoria virtuală este mai mare sau egală cu cea fizică. Totuși, în anumite scenarii, memoria fizică poate fi mai mare decât cea virtuală (Intel 32-bit cu PAE)

https://stackoverflow.com/a/26424819

48) Ce zone de memorie se aloca static? Dar dinamic?

Stiva și heap-ul sunt alocate dinamic; text, data, rodata și bss sunt alocate static, înainte de începerea execuției.

49) Ce se întâmplă cu o variabilă modificată într-un proces copil din perspectiva procesului părinte?

Procesele nu share-uiesc variabile, deci nu s-ar întâmpla nimic cu "variabila modificată într-un proces copil".

50) Ce reprezintă un loader? Ce rol are acesta?

Loader-ul este o componentă a sistemului de operare care se ocupă cu încărcarea programelor și a bibliotecilor. Are rolul de a valida permisiuni, de map-are a executabilului de pe disk în memorie, de a copia argumentele în memoria virtuală, inițializează regiștrii (spre exemplu, `SP`) și face jump la entry-point (`_start`). Pe sistemele Unix, loader-ul este handler-ul apelului de sistem `execve()`. – creaza mediul in care sa se ruleze aplicatia.

51) La ce folosim apelul mprotect()? La ce mecanism de securitate putem face bypass folosind acest apel?

Apelul `mprotect()` este folosit la schimbarea permisiunilor unuei zone de memorie. Cu ajutorul său se poate face bypass la `W^X`.

Apelul de sistem mprotect() este utilizat pentru a schimba permisiunile de acces la pagini de memorie existente. Acesta poate fi folosit pentru a controla și gestiona nivelul de acces și protecția paginilor de memorie într-un proces.

Principalul scop al mprotect() este de a permite programatorilor să specifice politici de securitate și protecție pentru paginile de memorie utilizate de aplicații. Utilizând mprotect(), se pot seta permisiuni diferite pentru paginile de memorie, cum ar fi permisiuni de citire, scriere sau executare.

Mecanismele de securitate care pot fi bypassate (sau circumvenite) folosind apelul mprotect() depind de contextul și permisiunile de securitate existente în cadrul unui sistem. Câteva exemple de circumvenire a mecanismelor de securitate prin mprotect() includ:

1. Protecție împotriva execuției (DEP - Data Execution Prevention): mprotect() poate fi utilizat pentru a modifica permisiunile unei pagini de memorie pentru a permite executarea codului în zone care în mod normal ar fi marcate ca fiind non-executabile. Acest lucru poate fi folosit pentru a evita mecanismele DEP și pentru a permite rularea de cod malicios în pagini de memorie marcate ca date.

2. Protecție împotriva scrierii (W^X): mprotect() poate fi utilizat pentru a permite scrierea în zone de memorie care în mod normal sunt marcate ca fiind doar pentru citire și execuție. Aceasta poate fi utilizată pentru a bypassa politici de securitate care impun restricții stricte asupra scrierii în pagini de memorie care conțin cod executabil.

Este important de menționat că circumvenirea mecanismelor de securitate prin mprotect() poate implica acces privilegiat la nivelul sistemului de operare și poate fi o acțiune malicioasă. În plus, reamintesc că discutarea sau promovarea activităților și metodelor care vizează bypass-ul sau exploatarea mecanismelor de securitate sunt împotriva politicilor de utilizare a sistemelor informatice și a securității cibernetice.

52) n ce zonă de memorie se află o variabilă globală, inițializată cu valoarea 0?

.bss —> globale neinit, sau init cu 0

Compute Q&A

1) De ce este nevoie de procese?

Procesul reprezintă modalitatea prin care o acțiune primește un identificator și posibilitatea de a putea fi executată.

El este important pentru că propune o interfață comună pentru realizarea de sarcini, iar CPU-ul (prin management-ul sistemului de operare), se ocupă de punerea în practică a acestor sarcini, împlinindu-și la rândul său scopul fundamental.

Nevoia de procese mai e dată de faptul că pe un sistem se dorește executarea mai multor acțiuni, consolidarea mai multor aplicații.

https://docs.google.com/document/d/1ZneG6oNqbbzEWLA6QwODLyYvHzdFj2OJ79Xka7QkkSs/edit#heading=h.v2p2vab8hzh2

2) De ce este nevoie de thread-uri?

Thread-ul reprezintă unitatea fundamentală planificabilă, mai multe thread-uri putând face parte din același proces. El apare ca urmare a necesității de a paraleliza, în general, fiind utilizat omogen, pentru acțiuni identice, realizate simultan.

https://www.ibm.com/docs/sv/aix/7.1?topic=programming-benefits-threads

Omogen -> Mai multe task-uri identice aproape de care avem nevoie realizate in parallel. Sincronizarea este usor de facut deoarece threadurile sunt in mare parte independente.

Eterogen -> Task-uri diferite de care avem nevoie sa fie rulate in parallel. Sincronizarea este mai dificila, overhead-ul este mai mare deoarece threadurile comunica mult intre ele si trebuie destul de des sa asteptam dupa alt thread pentru a ne putea continua activitatea

3) Ce este un proces?

Procesul este entitatea activă folosită pentru a executa acțiuni (nu neapărat omogene), utilizând în mod echitabil resursele sistemului.

Reprezinta interfata de realizare a sarcinilor intr-un mod controlat, fiind identificate unic si manageruite de catre kernel. Acestea primesc resurse individuale, sunt monitorizate independent si putem observa usor cand un process are crash (Spre exemplu o aplicatie, Elden Ring da fail, stim rapid ca el a produs eroarea).

4) Ce este un thread?

Thread-ul este cea mai mică unitate de procesare programabilă de sistemul de operare pentru execuție. El poate fi privit ca o combinație unică de (IP (instruction pointer), SP (stack pointer))

5) Cu ce diferă un thread de un proces?

1. Relație de incluziune: Procesele pot avea thread-uri, thread-urile nu pot avea procese.

2. Memoria partajată: Procesele nu partajează resurse, pe când thread-urile da (întreg

VAS-ul procesului care le naște, mai puțin TLS-ul (thread local storage), stiva și regiștrii).

3. Erori: Dacă un proces își încheie execuția necontrolat, din cauza unei erori, programul

poate continua să ruleze normal, pe când dacă un thread crapă, tot programul crapă.

4. Overhead: Thread-urile sunt mai lightweight și pot fi planificate mai ușor, de asemenea,

ele sunt mai rapide prin timpul de creare și cel de context switch.

https://www.geeksforgeeks.org/difference-between-process-and-thread/

6) Cum este afectat spațiul virtual de adrese al unui proces în momentul creării unui thread?

Spațiul virtual de adrese al procesului primește o nouă zonă de memorie, pentru stiva noului thread.

https://stackoverflow.com/questions/38555287/how-memory-management-happens-for-proc ess-threads-in-one-virtual-address-space

7) Ce zone de memorie au comune thread-urile unui proces și ce zone au specifice?

Au la comun .text, .data, .rodata, .bss, heap-ul, bibliotecile dinamice și kernel space-ul. Au specifice doar TLS-ul () și stiva, iar stiva contine tls ul

TLS stands for Thread-Local Storage. It is a mechanism provided by operating systems and programming languages to allocate and manage memory that is specific to each individual thread in a multi-threaded program.

In a multi-threaded application, multiple threads share the same memory space, including global variables and static variables. However, there are situations where each thread needs its own private memory space that is not shared with other threads. This is where TLS comes into play.

TLS allows each thread to have its own separate area of memory called thread-local storage. This storage is unique to each thread and is not shared with other threads. Each thread can store and access its own set of variables within its TLS without interfering with the variables of other threads.

TLS is useful in scenarios where multiple threads need to maintain separate copies of certain data. For example, in a server application, each thread may need to store its own thread-specific data such as a user session context or thread-specific configuration settings. TLS provides a convenient and efficient way to manage this thread-specific data.

The operating system and programming languages typically provide APIs and mechanisms to allocate and access TLS. For example, in C/C++, the `thread_local` keyword can be used to declare thread-local variables. The operating system manages the allocation and cleanup of TLS memory for each thread.

It's important to note that TLS is specific to individual threads within a process. It should not be confused with process-local storage or global variables, which are accessible by all threads within the process.

8) Ce conține PCB (Process Control Block) ?

SP-ul (stack pointer ul)

starea procesului

PID-ul

IP-ul (instruction pointer)

regiștrii

PT (page table)

FDT (file descriptor table)

https://www.cs.auckland.ac.nz/courses/compsci340s2c/lectures/lecture06.pdf

PCB (Process Control Block) este o structură de date utilizată de un sistem de operare pentru a păstra și gestiona informațiile despre un proces în timpul execuției acestuia. PCB, cunoscut și sub denumirea de TCB (Task Control Block), conține o serie de informații importante despre starea și contextul unui proces. Conținutul PCB poate varia în funcție de sistemul de operare specific, dar în general, acesta poate include următoarele informații:

1. Identificatorul procesului (PID): Un identificator unic atribuit procesului de către sistemul de operare. PID-ul servește la identificarea unică a procesului în cadrul sistemului.

2. Starea procesului: Informație despre starea procesului, cum ar fi "pregătit pentru execuție" (ready), "în execuție" (running), "așteptând" (waiting), "suspendat" (suspended), "terminat" (terminated) etc.

3. Contextul procesului: Informații despre contextul de execuție al procesului, inclusiv valorile registrelor de procesor (cum ar fi registrele de bază, de stivă, de instrucțiuni și de stare), conținutul registrelor de

programe (PC - Program Counter) și alte informații relevante necesare pentru a restabili și continua execuția procesului într-un moment ulterior.

4. Prioritatea procesului: Nivelul de prioritate atribuit procesului pentru planificarea și asignarea resurselor de sistem.

5. Informații despre resurse: Informații despre resursele alocate procesului, cum ar fi descriptorii de fișiere deschiși, spațiul de adresă virtual, zona de stivă, resursele de memorie și alte resurse asociate.

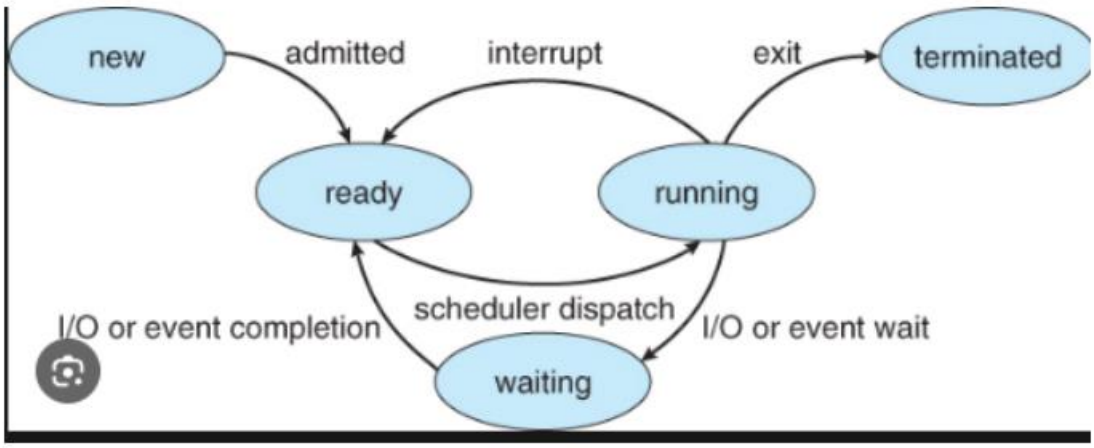
6. Informații despre planificare: Informații despre politica de planificare și informații necesare pentru algoritmul de planificare, cum ar fi timpul total de execuție al procesului, timpul de execuție rămas, timpul de așteptare, numărul de întreruperi și altele.

7. Informații despre securitate: Informații despre privilegiile și permisiunile asociate procesului, cum ar fi drepturile de acces la resurse și informații despre proprietatea procesului.

PCB reprezintă o structură esențială pentru gestionarea și controlul proceselor într-un sistem de operare. Aceasta permite sistemului de operare să monitorizeze și să gestioneze resursele și execuția proceselor, să realizeze planificarea și sincronizarea acestora și să asigure izolarea și protecția între procese.

9) Care sunt stările în care se poate găsi un thread?

* NEW
* READY
* RUNNING
* BLOCKED / WAITING
* TERMINATED



https://docs.google.com/document/d/1Um_iOiKEeKAPP0xjr8fXFbpPx3_wBdJZtjqWinqmbqs/edit#heading=h.7sfnwfm7vsop

10) Ce efect are apelul fork()?

`fork()` creează un proces copil, se întoarce de două ori (o dată în părinte cu PID-ul copilului, și o dată în copil cu valoarea 0) și este asociat cu mecanismul de copy-on-write. !!!!

Apelul de sistem `fork()` este folosit în sistemele de operare pentru a crea un proces nou prin duplicarea procesului apelant. Efectul apelului `fork()` este că procesul apelant se bifurcă în două procese separate, care rulează în paralel, cunoscute sub denumirile de proces părinte și proces copil.

Iată câteva efecte ale apelului `fork()`:

1. Duplicarea procesului: Apelul `fork()` creează o copie exactă a procesului apelant, inclusiv starea procesului, contextul de execuție, descriptorii de fișiere, zona de memorie și alte resurse asociate. Procesul copil este o copie identică a procesului părinte.

2. Identificatori de proces: După apelul `fork()`, atât procesul părinte, cât și procesul copil primesc un identificator unic de proces (PID), dar au PID-uri diferite. Procesul părinte primește PID-ul procesului copilului prin valoarea returnată de apelul `fork()`, în timp ce procesul copil primește valoarea 0 ca rezultat al apelului `fork()`.

3. Execuție paralelă: După apelul `fork()`, procesul părinte și procesul copil rulează în paralel și au fluxuri de execuție separate. Ambele procese continuă execuția de la punctul în care a fost efectuat apelul `fork()`, dar în spații de adrese virtuale separate.

4. Returnarea valorii din `fork()`: Apelul `fork()` returnează valoarea PID-ului procesului copil în procesul părinte. Procesul copil primește valoarea 0 ca rezultat al apelului `fork()`, indicând că este copilul.

5. Procese separate: Procesul părinte și procesul copil au resurse separate și spații de adrese virtuale separate. Modificările făcute într-un proces nu vor afecta celălalt proces, cu excepția schimbărilor care ar putea fi vizibile într-un sistem de fișiere partajat sau prin mecanisme de comunicare între procese, cum ar fi pipe-uri sau memorie partajată.

Apelul `fork()` este adesea folosit ca bază pentru crearea de procese în sistemele de operare, deoarece permite crearea de procese noi cu ușurință și permite paralelismul în execuție. De obicei, într-un program după apelul `fork()`, procesul părinte și procesul copil pot executa diferite secțiuni de cod, putând realiza sarcini diferite sau comunicând între ele pentru a împărți rezultate sau resurse.

11) Ce resurse partajează/nu partajează procesul părinte și procesul copil în cazul apelului fork()?

După apelul funcției `fork()`, VAS-ul părintelui este copiat și atribuit copilului. Inițial, cele două VAS-uri vor conține aceleași corespondențe adrese virtuale - adrese fizice; de abia la write acestea vor fi modificate, atât pentru părinte, cât și pentru copil. Procesul copil mai primește, de asemenea, și o copie a FDT părintelui, cu descriptori de fișiere ce point-ează către aceleași open file structures ca file descriptor-ii părintelui.

12) Cum modifica fork() si exec() spațiul virtual de adrese?

Atât `fork()` cât și `exec()` creează un spațiu nou de adrese, diferența dintre cele fiind aceea că `fork()` creează un nou spațiu de adrese pentru procesul copil, pe când `exec()` golește spațiul virtual de adrese al procesului curent (mai puțin partea de kernel space) și regiștrii și pregătește pentru a rula noul program.

https://stackoverflow.com/a/1653415

13) Ce efect are apelul exec()?

De cele mai multe ori, în sistemele Unix moderne apelul `exec()` este folosit împreună cu apelul `fork()` (modelul fork-and-exec) pentru a încărca un nou program în memorie, după crearea unui nou VAS. Un exemplu de suprascriere utilizând `exec()` este porinirea oricărui executabil din shell: shell-ul realizează un copil al său, a cărui imagine va fi înlocuită cu imaginea executabilului, care apoi va fi lăsat să ruleze. Shell-ul așteaptă copilul să-și încheie execuția, și apoi, îi dezalocă resursele.

14) Câte threaduri se pot găsi în starea RUNNING, READY și WAITING?

În RUNNING se găsesc atâtea thread-uri câte procesoare avem. În READY și în WAITING se pot găsi oricâte thread-uri.

15) Ce este o schimbare de context? Ce se întâmplă la o schimbare de context?

când planificatorul decide că un proces părăsește procesorul, are loc o schimbare de context schimbarea de context înseamnă salvarea informațiilor procesului anterior într-o zonă din sistemul de operare și restaurarea informațiilor noului proces schimbarea de context înseamnă overhead

O schimbare de context reprezintă schimbarea stării unui thread: marchează trecerea

către și de la starea de RUNNING.

O schimbare de context implică:

* salvarea stării thread-ului: punerea regiștrilor săi pe stivă, salvare IP-ului și SP-ului în

TCB (thread control block)

* introducerea handle-ului pentru TCB-ul thread-ului care pleacă de pe procesor în coada

de READY sau BLOCKED, de la caz la caz

* pregătirea următorului thread care va rula: retragerea IP-ului său din TCB-ul aflat în

coada de priorități, restaurarea stării regiștrilor săi

https://stackoverflow.com/questions/7439608/steps-in-context-switching

16) Ce cauzează schimbări de context?

schimbarea de context se poate produce:

* voluntar: un proces decide sau cauzează schimbarea de context

** un proces cedează de bună voie procesorul: yielding

** un proces execută o operație blocantă (de exemplu citire de la un dispozitiv care nu are

încă date)

** un proces își încheie execuția

* nevoluntar: planificatorul sistemului de operare decide forțarea unui proces de pe procesor

** un proces a stat prea mult timp pe procesor (i-a expirant cuanta)

** apare în sistem un proces mai important

—------

Schimbările de context apar la plecare voluntară/nevoluntară a thread-urilor de pe

procesor:

* `yield()`: thread-ul care rulează îl lasă voluntar pe următorul din coadă să îi ia locul

* expirarea quantei de timp: thread-ul a rulat atât cât îi fusese indicat de scheduler

* apel blocant: thread-ul așteaptă după I/O

* apare un thread cu o prioritate mai mare

* exit: thread-ul și-a încheiat treaba înainte de expirarea quantei

17) Ce este o schimbare de context voluntară și o schimbare de context nevoluntară?

O schimbare de context voluntară este una în care thread-ul face o acțiune despre care știe sigur că va duce la înlocuirea sa de pe procesor (o acțiune, care, programabil, îl dă afară): apelurile blocante sau cele de `yield()`.

O schimbare de context nevoluntară nu este dată de o acțiune directă, ci de ceea ce dictează scheduler-ul, pentru a menține echitatea timpului petrecut rulând: expirarea quantei, thread prioritar.

18) Ce sunt threadurile cu implementare user-level și thread-urile cu implementare kernel-level?

Thread-urile user-level (sau green threads) sunt thread-uri ce rulează exclusiv în user space, fiind controlate de biblioteci; pentru sistemul de operare, ele sunt privite ca un singur thread, putând folosi la un moment dat un singur procesor. Thread-urile de sistem sunt thread-uri ce rulează în kernel space (produc context switch-uri) și sunt privite ca mai multe acțiuni ce pot fi paralelizate (ocupă mai mult de un core). Green thread-urile sunt utile atunci când context switch-urie dese produc mult mai mult overhead față de acțiunea propriu-zisă pe care o pun în aplicare acestea. Cu toate astea, kernel level threads sunt ușor paralelizabile și au implementări de bibliotecă în limbaje precum C (pthread).

19) În ce situație este utilă zona TLS (thread local storage)?

TLS este utilă atunci când dorim să avem variabile statice/globale vizibile pentru toate funcțiile thread-ului, dar locale lui. Cu alte cuvinte, aceste variabile sunt accesabile de thread-ul de care aparțin și doar de el, pentru thread având scope global/static. TLS ul e in stiva

20) De ce este necesară sincronizarea proceselor/thread-urilor?

Sincronizarea este folosită pentru a asigura două comportamente din partea thread-urilor: serializare și acces exclusiv. Prin acestea, este menținut caracterul determinist al acțiunilor care pot fi deturnate de la cursul lor natural din cauza thread-urilor sau proceselor modificând zone de memorie partajată.

21) Ce înseamnă race condition?

Race-condition se referă la rezultatul nedeterminist pe care o are acțiunea făcută două sau mai multe thread-uri simultan, dorind să modifice o resursă comună.

22) Ce înseamnă deadlock?

Deadlock-ul este rezultatul acțiunii prin care două thread-uri așteaptă ceva din partea unui altuia. Cu alte cuvinte, thread-urile depind unul de altul și nu își pot continua execuția deoarece niciunul dintre ele nu poate ieși din loop.

https://www.baeldung.com/cs/deadlock-livelock-starvation

23) Care sunt dezavantajele sincronizării?

* greu de implementat dacă programatorul nu este experimentat

* implementare ineficientă: regiuni critice mari, deadlock-uri, race conditions

* overhead temporal mare:

* la instrucțiuni atomice: lock pe magistrală, un singur procesor accesează memoria

* la spinlock: busy waiting ocupând loc pe procesor

* la mutex/semafor: apel de sistem, context switch

Un spinlock este o formă simplă de lock (blocaj) utilizată în programare concurentă pentru a asigura accesul exclusiv la o resursă partajată între mai multe fire de execuție (thread-uri) într-un mod sincronizat. Acesta se bazează pe conceptul de așteptare activă (busy-waiting), în care un fir de execuție se blochează (spin) într-un ciclu repetitiv până când poate obține accesul exclusiv la resursă.

Caracteristicile unui spinlock includ:

1. Așteptare activă: În loc să elibereze procesorul în timpul așteptării, un fir de execuție care nu poate obține accesul la resursă intră într-un ciclu repetitiv de verificare a stării lock-ului (blocat sau eliberat). Acest ciclu de verificare continuă până când lock-ul este eliberat și firul de execuție poate obține accesul la resursă.

2. Eficient pentru perioade scurte de așteptare: Spinlock-urile sunt eficiente atunci când așteptarea este de scurtă durată, deoarece așteptarea activă nu implică schimbarea contextului de execuție sau trecerea în modul inactiv.

3. Consum de resurse: Un spinlock poate utiliza resurse de procesor în timpul așteptării active, deoarece firul de execuție care spintă este activ și utilizează procesorul într-un ciclu repetitiv. Acest lucru poate reduce eficiența generală a sistemului, în special în cazurile în care așteptarea este prelungită.

Spinlock-urile sunt utilizate în situații în care resursa partajată este utilizată într-un mod scurt și rapid și se așteaptă ca accesul la resursă să fie eliberat rapid de către firul de execuție care o deține. De exemplu, în cadrul unui algoritm de sincronizare sau în manipularea atomică a datelor, spinlock-urile pot oferi o soluție simplă și eficientă.

Este important de menționat că utilizarea spinlock-urilor necesită atenție în gestionarea corectă a sincronizării și evitarea situațiilor de interblocare (deadlock) sau așteptarea nesfârșită (livelock). Utilizarea spinlock-urilor trebuie făcută cu grijă și adaptată în funcție de cerințele specifice ale aplicației și de caracteristicile sistemului.

24) Ce înseamnă TOCTOU (time of check to time of use)?

Se referă la un race condition care apare imediat după un compare, înainte de a se face instrucțiunea următoare (de folosire), când un alt thread apare și modifică valoarea considerată "safe". https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use

25) Când se blochează un producător în problema producator-consumator? Dar un consumator?

Un producător se blochează dacă consumatorul nu a citit informația pe care el a produs-o

(când buffer-ul comun e plin).

Un consumator se blochează dacă producătorul nu i-a oferit nicio informație pe care el să

o poată folosi (când buffer-ul comun e gol).

26) De ce este necesară prezența unei instrucțiuni de tipul atomic_compare_and_swap în fiecare ISA?

Pentru că acest tip de instrucțiune stă la baza tuturor mecanismelor de sincronizare. Dacă compare and swap nu ar fi garantat atomică, nici mutex-urile, nici semafoarele sau spinlock-urile nu ar putea exista.

Prezența unei instrucțiuni de tipul atomic_compare_and_swap (atomic compara și interschimbă) în fiecare ISA (arhitectură de set de instrucțiuni) este importantă și necesară pentru a permite implementarea eficientă a operațiilor atomice și a sincronizării în programare concurentă.

Instrucția atomic_compare_and_swap (adesea abreviată ca CAS) este utilizată pentru a efectua o operație atomică în care o valoare este comparată cu o valoare stocată în memorie și, în cazul unei potriviri, o nouă valoare este interschimbată cu valoarea stocată în mod atomic. Această operație se efectuează într-o singură instrucțiune, garantind că nu există interferențe de la alte fire de execuție care ar putea să modifice valoarea în timpul execuției.

Iată câteva motive pentru care prezența instrucțiunii CAS în fiecare ISA este importantă:

1. Sincronizare atomică: Instrucția CAS permite implementarea operațiilor atomice, ceea ce înseamnă că operațiunile de citire, modificare și scriere a unei valori pot fi realizate în mod atomic, fără a fi întrerupte de alte fire de execuție. Aceasta asigură coerența și corectitudinea operațiunilor în contextul programării concurente.

2. Evitarea condițiilor de curse (race conditions): CAS oferă un mecanism de sincronizare sigur împotriva condițiilor de curse, în care două sau mai multe fire de execuție încearcă să acceseze și să modifice în mod concurent aceeași zonă de memorie. Utilizarea CAS poate asigura că o operație atomică este efectuată de un singur fir de execuție într-un mod ordonat.

3. Implementarea structurilor de date sincronizate: Instrucția CAS este esențială în implementarea structurilor de date sincronizate, cum ar fi lock-uri, semafoare, cozi și alte algoritmi de sincronizare. Aceasta permite gestionarea sigură a accesului concurent la aceste structuri de date și evitarea condițiilor de curse.

4. Performanță și eficiență: Instrucția CAS este implementată în mod direct în hardware și este optimizată pentru a oferi performanță și eficiență în sincronizarea atomică. Prin utilizarea unei singure instrucțiuni, evită nevoia de sincronizare externă sau de întoarcere la nivel de sistem de operare pentru a realiza operații atomice, ceea ce poate duce la o performanță mai bună și un consum mai mic de resurse.

Astfel, prezența instrucțiunii CAS în fiecare ISA este crucială pentru a asigura sincronizarea atomică și implementarea eficientă a operațiilor atomice în programare concurentă.

27) Cu ce diferă un spinlock de un mutex? Când folosim spinlock-uri? Când folosim mutex-uri?

Spinlock-urile sunt o formă de busy waiting (rulare in mod nedefinit) în starea de

RUNNING.

Ele nu determină ieșirea de pe procesor a thread-ului care așteaptă.

Sunt folosite atunci când timpul de așteptare nu este mare (zone critice mici, fără apeluri

blocante), iar un context switch este mai costisitor în comparație.

Mutex-urile sunt o formă de sincronizare care funcționează prin context switch, și

ocazional apel de sistem (chemarea scheduler-ului la `lock()` și `unlock`).

Sunt folosite în toate celelalte cazuri în care spinlock-ul nu este viabil: apeluri blocante,

zone critice mari

https://www.geeksforgeeks.org/mutex-vs-semaphore/

Practic, spinlock-urile nu provoaca schimbare de context si eliberare a procesorului pe care ruleaza un thread. Astfel ca desi salveaza overhead temporal deoarece evita context switch, pentru durate mai lungi de timp sau zone critice mari nu este eficient.

28) Ce efect are folosirea operatorului & din shell în crearea unui proces?

Shell-ul nu mai face waiting blocant pentru procesul ce primește &. Acesta este trecut în background, își continuă execuția, iar la final, când dă exit, părintele (shell-ul) primește semnalul `SIGCHLD` și face `wait()`.

29) Ce este un proces zombie? Cum apare un proces zombie? Care este problema proceselor zombie?

Un proces zombie este un proces care și-a încheiat execuția și căruia nu i s-a făcut wait de către părintele său. Problema cu procesele zombie este că ocupă resurse utile (PID, intrare în tabela de procese) degeaba, deoarece acestea nu mai au nimic de executat, dar nu sunt eliberate.

Atunci când un proces execută funcția wait() într-un sistem de operare, acesta suspendă execuția sa și așteaptă ca unul sau mai mulți dintre copiii săi să se termine. Funcția wait() este utilizată pentru sincronizarea proceselor părinte și copil într-un mediu multitasking.

30) Ce este un proces orfan? De ce un proces este orfan foarte puțin timp?

Un proces orfan este un proces al cărui părinte a murit înainte lui. Procesele orfane sunt adoptate rapid de către `init`.

31) Poate fi un proces zombie orfan? Ce se întâmplă cu un proces zombie orfan?

Da. Un proces zombie orfan este tratat mai întâi ca unul orfan, fiind adoptat de `init`. Acesta îl va aștepta și îi va elibera resursele pe care părintele inițial nu a apucat să le trateze.

32) Ce se întâmplă dacă un thread realizează un acces invalid la o zonă de memorie?

Thread-ul, împreună cu întreg procesul de care ține, va primi SEGFAULT, iar programul se va încheia cu o eroare.

33) Poate un thread să acceseze stiva altui thread? Cum? Da, ca sunt in acelasi spatiu de adresa

Da. Există mai multe variante:

* se reține într-o variabilă globală o adresă de stivă a altui thread (pentru thread-uri ale

aceluiași proces)

* se iau adrese random, dacă adresa este validă totul merge bine, dacă nu, `SEGFAULT`

(se merge la ghicit)

https://leetcode.com/discuss/interview-question/operating-system/124628/Can-a-posix-threa

d-modify-another-thread's-stack-variable-within-the-same-program/236079

34) De ce schimbarea de context între două thread-uri ale aceluiași proces este mai rapidă decât schimbarea de context între două thread-uri din procese diferite?

Pentru că thread-urile aceluiași proces share-uiesc majoritatea VAS-ului procesului, iar flush-ul TLB-ului nu este necesar.

35) Ce limitează numărul maxim de threaduri care pot fi create în cadrul unui proces?

Dimensiunea memoriei virtuale, deoarece pentru fiecare nou thread trebuie creată o nouă stivă în cadrul VAS-ului procesului.

36) Ce limitează numărul maxim de procese care pot fi create în cadrul unui sistem de calcul?

Numărul maxim de intrări în tabela de procese. De obicei un număr fix setat de sistemul de operare. https://unix.stackexchange.com/questions/124040/how-to-determine-the-max-user-processvalue

37) Ce se întâmplă când toate procesele sistemului sunt blocate?

Apare un proces IDLE care rulează în mod nedefinit (prin busy waiting) până un alt proces este pregătit să intre pe procesor.

Un proces IDLE (sau proces Idle) este un proces special care rulează pe un sistem de operare și care se ocupă de gestionarea timpului de procesor nefolosit. Scopul principal al unui proces IDLE este de a menține procesorul ocupat chiar și atunci când nu există alte procese care să fie în execuție activă.

Iată câteva activități specifice pe care un proces IDLE le poate efectua:

1. Bucla de așteptare (Busy-waiting): Un proces IDLE poate executa o buclă de așteptare infinită, care consumă timpul de procesor nefolosit. Aceasta asigură că procesorul nu intră într-o stare pasivă sau nu trece într-un mod de economisire a energiei atunci când nu există alte sarcini de executat.

2. Execuție de instrucțiuni minime: Un proces IDLE poate executa un set minim de instrucțiuni repetitive și inofensive pentru a menține procesorul ocupat. Aceste instrucțiuni pot fi de exemplu instrucțiuni simple de calcul sau operații aritmetice.

3. Economisirea energiei: În anumite cazuri, procesul IDLE poate fi optimizat pentru a permite procesorului să intre în moduri de economisire a energiei, cum ar fi oprirea sau reducerea frecvenței de lucru a procesorului. Acest lucru ajută la economisirea energiei și la reducerea consumului de energie al sistemului.

4. Sincronizarea evenimentelor: În unele sisteme, procesul IDLE poate fi utilizat pentru a efectua sincronizarea sau semnalarea anumitor evenimente sau acțiuni în sistem. De exemplu, poate fi utilizat pentru a declanșa anumite acțiuni atunci când sistemul se află într-o stare inactivă sau pentru a răspunde la evenimente specifice.

Procesul IDLE este de obicei executat cu prioritate scăzută și este planificat să ruleze atunci când nu există alte sarcini de priorități mai mari care să fie executate de procesor. Acesta ajută la menținerea

stabilității și performanței sistemului de operare, precum și la gestionarea eficientă a resurselor de procesor nefolosite.

38) Ce înseamnă waiting time (timp de așteptare) în planificarea proceselor?

Waiting time este timpul pe care îl petrec procesele în starea READY. Ideal, acesta este cât mai mic.

39) Putem avea un sistem multi-core cu un singur proces aflat în starea RUNNING și mai multe procese în READY?

Nu, dacă prin proces se înțelege un singur thread. Da, dacă prin proces se înțelege un proces cu mai multe thread-uri; în acest caz, toate thread-urile procesului pot rula în același timp, ocupând toate procesoarele și nepermițând altor task-uri să ruleze.

40) Ce înseamnă starea READY/RUNNING/TERMINATED/WAITING(BLOCKED)?

READY → thread-ul este gata pentru a intra pe procesor, așteaptă să i se facă loc, să

fie planificat

RUNNING → thread-ul rulează, are o cuantă stabilită și poate executa oricâte acțiuni

pe parcursul ei

TERMINATED → thread-ul și-a încheiat execuția și face `exit()`

BLOCKED → thread-ul a făcut un apel blocant și așteaptă după I/O

41) Ce este un proces I/O intensive?

Un proces care face multe apeluri blocante, comunică des cu diskul sau placa de rețea; trece des din RUNNING în BLOCKED.

42) Ce este un proces CPU intensive?

Un proces care face multe calcule aritmetice; adeseori ajunge CPU hog și este trimis nevoluntar (la expirarea quantei) în READY.

43) Cum tratează planificatorul procesele I/O intensive și procesele CPU intensive?

Procesele I/O intensive sunt prioritare: ele, în mod garantat, fac multe apeluri blocante și

ajung voluntar în BLOCKED, nu au tendința de a acapara CPU-ul.

Pe de altă parte, procesele CPU intensive nu sunt favoritele scheduler-ului: au tendința de

a ajunge CPU hogs, de aceea, ele primesc o cuantă mai mică, pentru a menține wainting

time-ul mic și eficiența împărțirii resurselor.

44) Două thread-uri ale unui proces execută aceeași funcție. Care sunt diferențele între cele două thread-uri?

Poate diferi atât SP-ul, cât și IP-ul.

Totul depinde de rapiditatea cu care thread-urile sunt planificate; nimic nu poate garanta

sau ghici ordinea lor, tocmai de aceia, sincronizarea este importantă.

Pentru că oferă o garanție a ordinii și serializării, atunci când scheduler-ul nu poate fi

ghicit.

45) Cu ce diferă procesul copil (creat prin fork) de procesul părinte?

Prin PID, prin VAS-uri (după write-uri multiple, inițial sunt identice, dar cu permisiunea de write scoasă).

46) Cu ce diferă un proces zombie de un proces orfan?

Procesul zombie poate avea un părinte în viață.

47) Ce parametri ai planificatorului trebuie să modificăm pentru a avea un sistem cu productivitate mai mare?

Procesele CPU intensive primesc o cuantă de timp mai mare.

48) Ce parametri ai planificatorului trebuie să modificăm pentru a avea un sistem cât mai interactiv (responsive)?

Procesele interactive, de I/O primesc prioritate mare.

49) Am avea nevoie de folosirea unui apel de sistem pentru crearea unui thread în cazul unei implementări de tip user-level threads? Dar în cazul deschiderii unui fișier în același scenariu?

Nu este necesar un syscall pentru crearea unui thread în user space, deoarece există conceptul de green threads (fire de execuție ce rulează la nivel de bibliotecă). Este necesar un syscall pentru deschiderea unui fișier, deoarece, în acest caz, interacționăm cu hardware-ul (disk-ul) și avem nevoie de permisunea kernel-ului pentru a face asta.

50) Ce se întâmplă dacă folosim apeluri de sistem blocante în interiorul unui spinlock?

Deadlock.

Asta se intampla deoarece spinlock-ul nu elibereaza procesorul, astfel nu are cine sa vina in ajutor si sa deblocheze actiunea blocanta

51) De ce este necesară folosirea prefixului LOCK pentru realizarea operațiilor atomice?

Prefixul LOCK anunță sistemul de operare să facă un lock hardware pe magistrală. Un singur core mai are acces, astfel, la memorie, iar operația atomică poate avea loc.

https://stackoverflow.com/a/8891781

52) Care este avantajul folosirii mutexurilor în defavoarea spinlockurilor și invers?

Mutex-urile sunt folosibile pentru critical regions mari, cu apeluri blocante, pe când spinlock-urile sunt ieftine când vine vorba de `lock()` și `unlock()`, pentru că nu fac context switch și nici nu invocă scheduler-ul.

53) Care este echivalentul apelului wait() pentru threaduri?

`join()`

54) De ce este nevoie de apelul wait() / pthread_join()?

Apelurile `wait()` / `pthread_join()` realizează eliberarea resurselor folosite de proces / thread după terminarea acestora.

55) Ce tranziție între stările unui thread nu este posibilă?

READY → BLOCKED: thread-ul trebuie mai întâi să fie capabil să facă o acțiune pentru a putea face un apel blocant, iar asta se întâmplă numai dacă este running.

I/O Q&A

1) Ce conține un FCB (File Control Block)?

Un FCB conține:

un identificator (ino),

permisiuni,

ownership,

size,

timestamps,

pointeri la DB-uri

numărul de link-uri.

Un File Control Block (FCB) este o structură de date utilizată de sistemele de operare pentru a stoca informații despre un fișier specific. FCB-ul conține detalii și metadate referitoare la fișier, cum ar fi numele fișierului, dimensiunea, calea, atributele, informații despre permisiuni și acces,

timestamp-uri și altele. FCB-ul este creat și gestionat de sistemul de operare pentru fiecare fișier deschis sau în curs de utilizare în sistem.

Structura și conținutul unui FCB pot varia în funcție de sistemul de operare și de implementare, dar în general, un FCB poate conține următoarele informații:

1. Numele fișierului: Numele unic al fișierului și extensia acestuia, care identifică fișierul în sistemul de fișiere.

2. Calea fișierului: Adresa sau calea completă către locația fișierului în sistemul de fișiere.

3. Dimensiunea fișierului: Numărul de octeți sau de blocuri de memorie alocate pentru stocarea fișierului.

4. Atributele fișierului: Informații despre atributul fișierului, cum ar fi permisiunile de citire, scriere și execuție, proprietarul fișierului, data și ora creării, modificării și accesului la fișier.

5. Descriptor de fișier: Un descriptor sau un identificator unic utilizat de sistemul de operare pentru a face referire la fișier în timpul operațiilor de citire, scriere sau alte operațiuni.

6. Pointeri la blocurile de date: Unul sau mai mulți pointeri către blocurile de date care conțin conținutul real al fișierului.

7. Informații despre deschidere și acces: Informații despre deschiderea fișierului, cum ar fi numărul de referințe deschise, modul de acces curent (citire, scriere, etc.), indicatori de blocare sau deținere a fișierului.

8. Alte metadate: Informații suplimentare despre fișier, cum ar fi timestamp-uri pentru creare, modificare și acces, informații despre proprietar, informații despre permisiuni, etichete, atribute speciale etc.

FCB-ul este utilizat de sistemul de operare pentru a gestiona operațiunile de citire, scriere, deschidere, închidere și alte operațiuni pe fișiere. Acesta reprezintă o structură internă utilizată de sistemul de operare pentru a urmări și controla informațiile asociate cu un fișier și pentru a asigura un acces corect și eficient la acesta.

2) Ce reprezintă un descriptor de fișier?

Descriptorul de fișier este un indice în tabela de file descriptors. Ca intrare pentru un descriptor de fișiere este reținut un pointer către un open channel structure.

Un descriptor de fișier (file descriptor) este un număr întreg utilizat de un sistem de operare pentru a identifica și a gestiona un fișier deschis sau o resursă de intrare/ieșire (I/O). Descriptorul de fișier reprezintă o referință la un canal de comunicare între programul utilizator și sistemul de operare pentru a accesa și manipula fișiere, socket-uri, dispozitive și alte resurse I/O.

Fiecare proces care rulează în sistemul de operare are o tabelă de descriptori de fișiere asociată, cunoscută și sub numele de "tabel de fișiere deschise" (open file table). Această tabelă conține o intrare pentru fiecare fișier deschis de proces. Descriptorul de fișier este un indice în această tabelă, care permite accesul rapid la informațiile relevante despre fișierul deschis.

Descriptorii de fișiere sunt întregi non-negative și pot fi utilizate pentru a efectua diverse operațiuni asupra fișierelor, cum ar fi citirea, scrierea, căutarea, modificarea atributelor și altele. Sistemul de operare oferă funcții sau interfețe de programare a aplicațiilor (API) pentru a lucra cu descriptorii de fișiere, precum open(), read(), write(), close(), care permit programului utilizator să deschidă, să acceseze și să gestioneze fișierele.

Este important de menționat că valorile descriptorilor de fișiere pot varia în funcție de sistemul de operare și de implementare. De exemplu, în sistemele de operare bazate pe UNIX, descriptorii de fișiere întregi non-negative încep de obicei cu valoarea 0 și primele câteva (de obicei 0, 1 și 2) sunt predefinite pentru standardele de intrare (stdin), ieșire (stdout) și eroare (stderr). Celelalte descriptori sunt atribuite prin deschiderea explicită a fișierelor sau prin operarea cu alte resurse I/O.

3) Ce reprezintă tabele de descriptori de fișiere?

Tabelele de descriptori de fișiere reţin toate fişierele pe care un proces le foloseşte la un

moment dat.

Acestea sunt vectori de pointeri către open file structures și sunt populate prin apeluri

precum `open()`, `pipe()`, `dup()`, `create()`, `socket()` sau `accept()`.

4) Câte tabele de descriptori de fișiere se găsesc într-un sistem de operare?

Câte procese.

Asta pentru ca fiecare proces are o tabela de descriptori unica.

5) Ce efect are apelul dup()?

Duplică un file descriptor deja existent și ocupă o poziție nouă în FDT

Unul dintre principalele motive pentru a utiliza dup() este pentru a permite manipularea aceluiași fișier de către mai multe fire de execuție sau procese. Duplicarea unui descriptor de fișier permite fiecărui fir de execuție sau proces să aibă propria referință la același fișier, ceea ce permite operarea simultană și independentă asupra acestuia.

6) Ce efect are apelul close()?

Eliberează file descriptor-ul asociat. Scade numărul de link-uri către un open file structure. Dacă numărul ajunge să fie 0, atunci open file structure-ul este eliberat la rândul său

7) Ce apeluri modifică pointer-ul/cursorul de fișier (file pointer)?

Open(), read(), write(), lseek()

8) Ce apeluri modifică dimensiunea fișierului?

`open()`, `write()`, `ftruncate()`

La open poti pune un flag O_TRUNC care iti truncheaza fisierul daca deja exista si tu o sa faci scriere in el, iar daca nu e fisier de scriere iti lasa dimensiunea cat era

Dacă folosești flag-ul **O_TRUNC** împreună cu funcția **open()**, acesta va trunca fișierul la dimensiunea zero la momentul deschiderii. Acest flag specifică faptul că fișierul trebuie golit la deschidere. Prin urmare, orice conținut anterior al fișierului va fi eliminat și dimensiunea acestuia va fi setată la zero.

9) Ce efect are apelul/comanda truncate?

Setează dimensiunea fișierului la size-ul care este dat ca argument

10) Care sunt tipurile de fișiere pe un sistem de fișiere uzual Unix?

regular

directory

symlink

FIFO

block

char

socket.

Iată o explicație pentru fiecare tip de fișier menționat:

1. Fișier obișnuit (regular file): Acesta este tipul de fișier comun întâlnit în sistemul de fișiere. Un fișier obișnuit conține date sau informații într-un format specific, cum ar fi fișierele text, fișierele binare, fișierele executabile etc. Aceste fișiere pot fi citite și modificate de către utilizatori sau aplicații.

2. Director (directory): Acesta reprezintă un container special care conține o listă de nume de fișiere și alte informații despre fișierele și subdirectoarele conținute în interiorul său. Directoarele sunt utilizate pentru a organiza fișierele într-o structură ierarhică, permit navigarea și accesul la fișierele conținute în ele.

3. Symlink (symbolic link): Symlink-ul (link simbolic) este un tip special de fișier care conține o referință către un alt fișier sau director. Aceasta oferă un mecanism de creare a unui alias sau unei legături către un alt fișier sau director din sistemul de fișiere. Un symlink poate fi considerat un "shortcut" sau un "link" către un alt fișier sau director și poate fi utilizat pentru a accesa ușor resursele fără a fi necesar să se specifice calea completă către acestea.

4. FIFO (First-In, First-Out): FIFO, cunoscut și sub denumirea de "pipe numit" (named pipe), este un canal de comunicare între procese într-un sistem de operare. Acesta permite schimbul de date între procese, indiferent dacă procesele rulează în cadrul aceluiași sistem de operare sau pe sisteme de operare diferite. FIFO utilizează principiul primul-intrat, primul-ieșit, în sensul că datele sunt citite în ordinea în care au fost scrise în FIFO.

5. Block device (dispozitiv bloc): Fișierele de tip dispozitiv bloc reprezintă dispozitive hardware care permit accesul la date în blocuri de dimensiuni fixe. Acestea includ discuri rigide, discuri SSD, dispozitive de stocare externă etc. Accesul la fișierele de tip dispozitiv bloc se face în blocuri de date și poate implica operațiuni de citire și scriere.

6. Character device (dispozitiv caracter): Fișierele de tip dispozitiv caracter reprezintă dispozitive hardware care permit accesul la date într-un flux continuu de caractere. Acestea includ, de exemplu, porturile seriale, tastaturile, mouse-urile și alte dispozitive de intrare/ieșire. Accesul la fișierele de tip dispozitiv caracter se face într-un flux continuu și permite transmiterea și recepționarea de caractere.

7. Socket: Socket-urile sunt utilizate pentru comunicarea între procese prin rețea

ea. Ele reprezintă punctele terminale ale unei conexiuni de rețea și permit transmiterea și recepționarea de date între procese care rulează pe mașini diferite într-o rețea. Socket-urile pot fi utilizate pentru comunicare în timp real, transmiterea de date, comunicarea între client și server etc.

Aceste tipuri de fișiere reprezintă diferite entități în sistemul de fișiere și oferă funcționalități și comportamente specifice în ceea ce privește accesul, manipularea și utilizarea datelor și resurselor într-un sistem de operare.

11) Ce este un sistem de fișiere virtual?

ex: procfs

Un sistem de fișiere virtual este o interfață abstractă pe care sistemul de operare o expune

către sistemele de fișiere tradiționale (ext3, VFAT, FreeBSD).

Astfel, diferențele de implementare interne fiecărui astfel de sistem de fișiere nu devin o

problemă, iar kernelul dobândește o modalitate de comunicare facilă cu multiple sisteme de

fișiere simultan.

https://en.wikipedia.org/wiki/Virtual_file_system

Un sistem de fișiere virtual (Virtual File System - VFS) este un nivel de abstractizare sau o interfață între sistemul de operare și diferitele sisteme de fișiere suportate. Scopul principal al unui VFS este de a oferi o reprezentare uniformă și unificată a resurselor de stocare (fișiere și directoare) către aplicații și utilizatori, indiferent de specificațiile și caracteristicile sistemului de fișiere subiacent.

Un VFS permite aplicațiilor și utilizatorilor să acceseze și să gestioneze fișierele într-un mod consistent, indiferent de tipul sau locația acestora (cum ar fi sistemul de fișiere local, sistemul de fișiere de rețea, sistemul de fișiere montat etc.). Acesta abstractizează și ascunde detaliile specifice ale sistemelor de fișiere individuale și oferă o interfață comună pentru operațiuni precum citirea, scrierea, crearea și ștergerea fișierelor.

Prin intermediul unui VFS, aplicațiile pot utiliza operații standard pentru a accesa și manipula fișierele, indiferent de locația sau tipul sistemului de fișiere. VFS asigură transparența și interoperabilitatea între diferitele sisteme de fișiere, permițând, de exemplu, să se utilizeze aceleași apeluri de sistem pentru a citi sau a scrie atât în fișiere locale, cât și în fișiere de rețea.

Un sistem de fișiere virtual include următoarele componente cheie:

1. Interfață standard: Un set de funcții și apeluri de sistem standardizate pe care aplicațiile le pot utiliza pentru a accesa și manipula fișierele.

2. Tabele de operații: Fiecare sistem de fișiere subiacent suportat de VFS are asociată o tabelă de operații care definește funcțiile specifice ale acelui sistem de fișiere. Aceste funcții sunt apelate de VFS pentru a executa operațiile specifice pe fișiere în cadrul sistemului de fișiere subiacent.

3. Traducere și mapare: VFS traduce apelurile și operațiile comune ale sistemului de fișiere în apeluri specifice și operații specifice ale sistemului de fișiere subiacent. Aceasta include traducerea căilor și a permisiunilor, gestionarea bufferelor și a cache-ului, gestionarea blocurilor și alocarea/dealocarea spațiului de stocare etc.

Un exemplu de implementare a unui sistem de fișiere virtual este VFS din Linux, care permite utilizatorilor și aplicațiilor să acceseze și să utilizeze diferite tipuri de sisteme de fișiere, cum ar fi ext4, FAT32, NFS, CIFS etc., printr-o interfață comună. Aceasta oferă un nivel de abstractizare și portabilitate în gestionarea fișierelor și a resurselor de stocare în sistemul de operare.

12) Ce este un dispozitiv virtual?

Un dispozitiv virtual mimează existența unui dispozitiv hardware fizic, acesta fiind, însă, o

simplă intrare în sistemul de fișiere și neavând niciun backup fizic în spate.

Un astfel de dispozitiv virtual este creat cu apelul `mknod`.

https://en.wikipedia.org/wiki/Virtual_device

Un dispozitiv virtual este o reprezentare software a unui dispozitiv fizic sau a unei componente hardware. Acesta emulează comportamentul și funcționalitatea unui dispozitiv real, oferindu-le aplicațiilor și sistemului de operare aceleași interfețe și funcționalități ca și dispozitivul fizic.

Dispozitivele virtuale sunt create pentru a oferi flexibilitate, portabilitate și abstractizare între software și hardware. Ele pot fi utilizate într-o gamă largă de scenarii, inclusiv virtualizare, dezvoltarea și testarea de software, simulări și mai mult.

Iată câteva exemple de dispozitive virtuale:

1. Mașină virtuală: O mașină virtuală (Virtual Machine - VM) este un mediu de virtualizare care emulează un sistem de operare complet și rulează pe un sistem de operare gazdă. Mașinile virtuale permit rularea mai multor sisteme de operare sau aplicații separate în același timp, izolate unele față de celelalte.

2. Disk virtual: Un disk virtual este un fișier sau o zonă de spațiu de stocare emulată care simulează un dispozitiv de stocare fizic, cum ar fi un hard disk sau un dispozitiv flash. Aceste discuri virtuale pot fi utilizate în virtualizare, în dezvoltarea și testarea de software sau pentru a crea imagini de sistem și copii de rezervă.

3. Interfață de rețea virtuală: O interfață de rețea virtuală este o reprezentare software a unei interfețe de rețea fizică, cum ar fi un card de rețea. Aceasta permite comunicația între mașinile virtuale, între mașina virtuală și sistemul gazdă sau între diferitele mașini virtuale dintr-un mediu de virtualizare.

4. Controler virtual de stocare: Un controler virtual de stocare emulează comportamentul unui controler fizic de stocare, cum ar fi un controler RAID sau un controler SCSI. Acesta permite administrarea și gestionarea discurilor virtuale și a resurselor de stocare într-un mediu virtualizat.

5. Dispozitiv virtual de intrare/ieșire: Acestea includ tastaturi virtuale, mouse-uri virtuale și alte dispozitive de intrare/ieșire care emulează funcționalitatea dispozitivelor fizice. Ele sunt utilizate în diferite scenarii, cum ar fi testarea aplicațiilor, simularea evenimentelor de intrare/ieșire și mai mult.

Dispozitivele virtuale oferă o abstracție a funcționalității hardware și permit software-ului să utilizeze și să interacționeze cu dispozitivele, indiferent de detaliile specifice ale hardware-ului fizic subiacent. Acest lucru permite portabilitatea, flexibilitatea și eficiența în dezvoltarea și utilizarea software-ului.

13) Ce tipuri de dispozitive cunoașteți? Clasificați-le din orice punct de vedere cunoașteți

În funcție de câți bytes gestionează:

1. char devices (se ocupă cu un singur caracter la un moment dat - dispozitive seriale):

tastatură, mouse, terminal.

2. block devices (se ocupă cu blocuri de date cu dimensiunea multiplu de 512 - random

acces devices): disk, CD-ROM, flash drives

14) Cu ce diferă un dispozitiv de tip bloc de un dispozitiv de tip caracter? Dați câte un exemplu de fiecare

1. char devices: pentru char devices nu se face buffering; read și write există ca funcții și

sunt blocante; au un driver mai simplu, pentru ca sunt de tip stream și nu au de reținut decât

o singură poziție (cea curentă).

2. block devices: block devices sunt accesate prin cache; read și write sunt asincrone; au

un driver complicat, pentru că acesta trebuie să treacă prin buffer cache pentru a scrie și citi.

https://tldp.org/LDP/khg/HyperNews/get/devices/basics.html

15) De ce nu are sens operația de seek pe un dispozitiv de tip caracter?

Pentru că dispozitivele de tip caracter sunt de tip stream și rețin un singur byte la un moment dat.

16) Ce adresă IP locală și ce port local are un socket întors de apelul accept()?

Apelul `accept()` întoarce un fd pentru un socket cu adresa IP a rețelei locale și portul aplicației care dorește să se conecteze.

17) Ce valoare poate întoarce un apel read() sau un apel write()?

`read()` întoarce numărul de bytes citiți, 0 pentru EOF sau -1 în caz de eroare.

`write()` întoarce numărul de bytes scriși sau -1 în caz de eroare.

18) Ce operații se pot face pe fișiere?

`create()`, `open()`, `read()`, `write()`, `lseek()`, `truncate`, `close`, `delete`

19) Ce operații asupra fișierelor modifică/nu modifică valoarea cursorului unui fișier?

Modifică cursorul: `open()`, `read()`, `write()`, `lseek()`.

Nu modifică cursorul: `ftruncate()`

20) . Ce operații asupra fișierelor modifică/nu modifică dimensiunea fișierului?

Modifică dimensiunea: `open()`, `write()`, `truncate()`.

Nu modifică dimensiunea: `read()`, `lseek()`

21) Unde este reținută valoarea cursorului de fișiere (file pointer) și unde este reținută dimensiunea fișierului?

Cursorul de fișiere este reținut în open file structure, pe când dimensiunea fișierului este reținută în FCB.

22) De ce avem două buffere asociate fiecărui socket, ce rol are fiecare?

Socket-ul, spre deosebire de pipe, permite comunicarea bidirecțională. Pentru a asigura faptul că sender-ul și receiver-ul nu trimit informație dintr-o parte în alta în același timp, pe același canal, există două buffere: unul de send și altul de receive.

23) Ce este o operație asincronă?

O operație asincronă este caracterizată de faptul că nu poate fi determinat momentul în care aceasta apare și de faptul că rulează în mod independent.

24) Ce este o operație neblocantă?

O operație neblocantă este caracterizată de faptul că se întoarce imediat, indiferent dacă a finalizat acțiunea sau nu. Spre exemplu, un `read()` neblocant ar citi direct (dacă ar avea ce) sau s-ar întoarce cu un mesaj de eroare special, care anunță faptul că nu și-a dus la bun sfârșit task-ul, pentru că s-ar fi blocat.

https://www.geeksforgeeks.org/blocking-and-nonblocking-io-in-operating-system/

25) Ce întoarce o operație asincronă?

Operațiile asincrone nu întorc date, ci un cod de eroare, 0 sau 1. Datele pot fi accesate printr-un `eventfd()`.

26) Ce întoarce o operație blocantă?

Operația blocantă întoarce întotdeauna rezultatul acțiunii; tocmai de aceea este blocantă, pentru că așteaptă mereu până la încheierea task-ului.

27) Cu ce diferă un socket de rețea de un socket UNIX?

Socket-ul de rețea este folosit pentru comunicarea proceselor de pe sisteme diferite. Socket-ul UNIX este o formă de comunicare între procese aflate pe același sistem.
https://stackoverflow.com/questions/22897972/unix-vs-bsd-vs-tcp-vs-internet-sockets

28) Care este diferența între un pipe anonim și un pipe cu nume (named pipe)?

Pipe-ul cu nume permite comunicarea între două procese neînrudite, pe când prin pipe-ul anonim se poate transmite informație doar între procese înrudite. Mai mult, pipe-ul cu nume apare ca o intrare în sistemul de fișiere și reprezintă chiar un tip de fișier (FIFO). pipe cu nume → intrare in sist de fisiere pipe cu nume → FIFO

29) Ce este buffer cache-ul? Care este rolul său?

Buffer cache-ul este un buffer de blocuri aflat în kernel space. Acesta reține blocurile de date folosite recent, pentru a scădea numărul de interacțiuni cu disk-ul, o componentă extrem de lentă față de memorie.

https://www.oreilly.com/library/view/understanding-the-linux/0596002130/ch14s02.html

30) De ce operația write pe fișiere este foarte rar blocantă?

Operația de `write()` ar fi blocantă dacă buffer-ul din kernel space ar fi plin. Acesta este extrem de rar plin, pentru că dacă se umple, este fie eliberat (prin scrierea pe disk a informațiilor pe care le reține), fie mărit.

31) . În ce situație operația read() pe fișier se blochează?

Operația de `read()` este blocantă în cazul în care buffer-ul din kernel space este gol. Acțiunea va fi blocată până când cel puțin un octet ajunge în buffer.

32) Care este rolul unui device driver?

Device driver-ul este o bucată de software folosită pentru a controla diferite componente hardware. Are rolul de a permite sistemului de operare să se folosească de periferice.
https://en.wikipedia.org/wiki/Device_driver

33) Ce rol are controller-ul hardware?

Controlează componentele hardware (are rol de administrator). Este la rândul său o componentă hardware care permite transmiterea, flow-ul de date, între mai multe entități fizice.
https://www.techtarget.com/whatis/definition/controller

34) Ce înseamnă zero-copy? Ce mecanism/apel folosește zero-copy?

Zero-copy se referă la transferul de date între mai multe buffer-e din kernel space fără a se trece prin user space. Interacțiunea cu aplicația devine minimă, iar costul temporal adus de domain switch este evitat. Apeluri folosite pentru zero-copy sunt `sendfile()`, `splice()` (UNIX), `TransmitFile` (Windows).
https://open-education-hub.github.io/operating-systems/Lab/I/O/Zero-Copy/content/zero-copy

35) Ce garanții ni se oferă în momentul în care apelul send() se întoarce în user space?

Că datele în întregime sau doar o parte (de aceea `send()` trebuie făcut într-un loop pentru a asigura că s-au trimis toate datele) au fost copiate într-un buffer de send.

36) Cu ce diferă afișarea folosind printf() față de folosirea write()?

`printf()` spre deosebire de apelul de sistem `write()`, este (line) buffered, în sensul în care menține un buffer în user space care va fi flush-uit și trimis spre kernel mai rar (la endline, la apel explicit de flush, la închiderea descriptorului sau când bufferul din user space s-a umplut).

37) De ce subsistemul de networking nu folosește buffer cache-ul?

Subsistemul de networking poate să se folosească de buffer cache, dar nu o face din motive de performanță. Buffer cache-ul este destinat gestionării ăi stocării datelor care (1) se află ăn procesare sau (2) sunt accesate frecvent, în memoria locală a unui sistem. Subsistemul de rețea se ocupă cu transmisia de date între sisteme și poate gestiona un volum mare de date. Este necesară o procesare mai rapidă decât cea oferită de cache, deci utilizarea acestuia nu ar avea sens.

În locul buffer cache-ului, subsistemul de networking utilizează mecanisme și optimizări specifice pentru a gestiona și manipula eficient datele de rețea, cum ar fi folosirea bufferelor circulare (circular buffers), tehnici de "zero-copy" și optimizări ale stack-ului de rețea pentru a asigura performanța și eficiența în transferul datelor.

38) Ce rol are apelul / comanda sync?

`sync()` asigură integritatea datelor. Forțează scrierea tuturor datelor modificate în buffere-le din kernel space, în mod sigur, pe disc. În alte cuvinte, sincronizează sistemul de fișiere cu un dispozitiv de stocare.

39) Care este rolul apelului ioctl / DeviceIoControl?

`ioctl()` este un apel de sistem ce se ocupă cu comunicarea dintre aplicații și device driver-e. El poate fi folosit pentru mai multe tipuri de fișiere, precum char/block devices, sockets, în general tot ceea ce are un file descriptor.

https://github.com/open-education-hub/operating-systems/blame/master/content/chapters/io/lecture/slides/devices.md#L107

40) De ce în general doar utilizatorul root are permisiuni de scriere (uneori doar root are permisiuni de citire) pe intrările din /dev?

Pentru că interacționarea cu device-urile hardware se face privilegiat. Astfel, user-ul nu poate aduce riscuri de securitate, iar comunicarea este lăsată în mâna kernel-ului, aceasta rămânând funcțională și sigură.

41) De ce este apelul fwrite() mai rapid decât write atunci când facem multe scrieri?

`fwrite()` fiind buffered, devine mai rapid la mai multe scrieri: acesta le va combina pe toate într-un singur mare apel de sistem, în loc să facă pentru fiecare acțiune de scriere un syscall separat. https://itecnote.com/tecnote/linux-why-the-fwrite-libc-function-is-faster-than-the-syscall-writefunction/

42) Ce se întâmplă dacă facem open de mai multe ori consecutiv pe același fișier?

La fiecare apel de `open()`, se va adăuga un nou file descriptor în FDT și se va crea un alt open file structure pentru fd-ul creat.

https://stackoverflow.com/a/51750060

43) Cum se modifica tabelul de file descriptori după open() si dup()? Este echivalent cu doua apeluri open()?

`open()` creează un nou file descriptor și un nou open file structure în FDT, pe când `dup()` creează un nou file descriptor și point-ează către același open file structure de la vechiul fd, pe care îl primește ca argument, incrementând numărul de referințe (link-uri) din structură. `dup()` nu este neapărat echivalent cu două apeluri `open()` pentru că NU creează un open file structure; el doar copiază referința dintr-o parte a vectorului de fd-uri în alta.

44) Ce fișiere sunt deschise, în general, la crearea unui proces nou?

`stdin`, `stdout`, `stderr` cu file descriptorii 0, 1, 2

45) De ce este utilă prezența unor dispozitive pur virtuale în ierarhia /dev (ex. /dev/vboxnetctl, /dev/urandom)?

Pentru a putea emula un tip de dispozitiv hardware fără a avea la îndemână piesa respectivă; pentru a putea avea funcționalități precum cele date de urandom și random, pentru care nu există un echivalent fizic.

46) De ce, în general, apelul write() pe un fișier nu blochează threadul curent?

Pentru că realizează un simplu `memcpy()` între buffere-ele din user space și cele din kernel space.

47) În ce situație un apel read() pe un fișier blochează threadul curent?

Atunci când buffer-ul din kernel space e gol.

48) De ce spunem că operațiile read() / write() seamănă cu operația memcpy()?

Operațiile de `read()` și `write()` seamănă cu `memcpy()` deoarece doar trimit informația dintr-o parte în alta (buffer-e US - KS) prin copiere.

App Interaction Q&A

1) Ce forme de comunicare inter-proces cunoști?

* comunicare: pipe-uri, fișiere, sockets, shared memory

* sincronizare: prin semafoare cu nume

* semnale

pipe-uri ->

Pipe-urile sunt mecanisme de comunicare între procese utilizate în sistemele de operare pentru a permite transmiterea de date între procese. Un pipe poate fi considerat ca un canal sau un tub virtual prin care datele pot fi transmise de la un proces-sursă la un proces-destinație.

Există două tipuri de pipe-uri: pipe-uri cu nume (named pipes sau FIFOs) și pipe-uri anonime (anonymous pipes). Voi descrie în continuare pipe-urile anonime, care sunt cele mai frecvent utilizate.

Un pipe anonim se bazează pe crearea unui canal de comunicare între două procese care au o relație de părinte-copil sau care sunt create de același proces. Pipe-ul este unidirecțional și permite transmiterea datelor într-o singură direcție, de la procesul-sursă la procesul-destinație.

Procesul care creează pipe-ul (părinte sau procesul inițial) utilizează funcția de sistem `pipe()` pentru a crea canalul de comunicare. Această funcție returnează două descriptori de fișiere: unul pentru citire (read end) și unul pentru scriere (write end) a datelor în pipe.

Procesul părinte utilizează descriptorul de scriere pentru a trimite date către pipe, în timp ce procesul copil utilizează descriptorul de citire pentru a citi datele din pipe. Astfel, procesul-sursă (părinte) scrie în pipe și procesul-destinație (copil) citește din pipe.

Comunicarea între procese prin pipe-uri se realizează prin operațiile de citire și scriere folosind descriptorii de fișiere asociate. De exemplu, în limbajul C, se pot utiliza funcțiile `write()` pentru a scrie în pipe și `read()` pentru a citi din pipe.

Pipe-urile anonime sunt utilizate în general pentru comunicarea simplă între procese, în cazul în care datele sunt transmise într-un singur sens și nu există necesitatea de a transmite mesaje structurate sau de a gestiona sincronizarea între procese.

Este important de menționat că pipe-urile anonime sunt create doar între procese aflate în relația de părinte-copil sau în cadrul aceluiași proces, și nu pot fi utilizate pentru comunicarea între procese independente. Pentru comunicarea între procese independente sau interprocesuale, se pot utiliza alte mecanisme de comunicare, cum ar fi pipe-uri cu nume, cozi de mesaje sau canale de comunicație rețea.

Pipe-urile cu nume (named pipes), cunoscute și sub denumirea de FIFOs (First In, First Out), sunt mecanisme de comunicare bidirecțională între procese, care permit transmiterea de date între procese care rulează independent. Spre deosebire de pipe-urile anonime, care sunt limitate la comunicarea între procese în relația părinte-copil sau în același proces, pipe-urile cu nume pot fi utilizate pentru comunicarea între procese independente și chiar între procese care rulează pe sisteme de operare diferite.

Iată câteva caracteristici ale pipe-urilor cu nume:

1. Numele și crearea: Pipe-urile cu nume sunt identificate prin nume de fișier în sistemul de fișiere și pot fi create utilizând funcții specifice oferite de sistemul de operare, cum ar fi `mkfifo()` în sistemele bazate pe UNIX/Linux. Numele pipe-urilor cu nume pot fi văzute în sistemul de fișiere și pot fi utilizate pentru accesarea pipe-ului de către diferite procese.

2. Comunicare bidirecțională: Pipe-urile cu nume permit transmiterea de date bidirecțională între procese. Un proces poate scrie date în pipe și un alt proces poate citi acele date din pipe. Astfel, pipe-urile cu nume asigură o comunicație flexibilă și în ambele direcții între procese.

3. Comunicare independentă: Pipe-urile cu nume permit comunicarea între procese independente, adică procese care rulează în cadrul aceluiași sistem de operare sau chiar pe sisteme de operare diferite. Acestea pot fi utilizate pentru a realiza comunicație între procese din diferite aplicații sau chiar între aplicații distribuite.

4. Asincrone și sincrone: Pipe-urile cu nume pot fi utilizate în mod asincron sau sincron. În modul asincron, procesele pot scrie și citi din pipe în mod independent și neinteractiv, fără așteptare reciprocă. În modul sincron, procesele pot aștepta unul pe celălalt, astfel încât comunicarea să se desfășoare într-un mod sincronizat.

Pipe-urile cu nume sunt utilizate pentru a permite comunicația între procese și facilitarea schimbului de date între ele. Ele oferă o interfață simplă și eficientă pentru transmiterea de date și pot fi utilizate într-o gamă largă de aplicații, cum ar fi comunicarea între procese în rețele de calculatoare, interacțiunea între diferite componente ale unui sistem distribuit sau implementarea de funcționalități de tip client-server.

Fisiere -> Fisierul este accesat de ambele procese si un proces poate scrie in fisier informatie iar celalalt sa o preia si sa o foloseasca

Sockets -> Socket-urile au doua canale de comunicare, sender si receiver. Procesele pot astfel comunica prin aceasta modalitate. Un alt exemplu este biblioteca MPI

Shared Memory -> Procesele comunica printr-un spatiu de memorie alocat in memoria fizica, fiecare avand memorie virtuala care sa faca referire la aceeasi memorie fizica

2) Ce este un semnal? Când se trimite un semnal către un proces?

Semnalul este un mesaj trimis asincron proceselor pentru a declanșa un anumit

comportament.

La primirea semnalului este rulat un handler specific care tratează eroarea, întrerupe,

suspendă sau omoară procesul

Un semnal reprezintă o notificare asincronă trimisă de către un proces sau de către sistemul de operare către un alt proces pentru a comunica anumite evenimente sau stări. Semnalele sunt utilizate în sistemele de operare pentru a permite proceselor să interacționeze și să gestioneze evenimente neașteptate sau excepționale.

Un semnal este, în esență, o întrerupere sau o indicație că s-a întâmplat un eveniment specific. Aceste evenimente pot fi diverse, cum ar fi apăsarea unei taste speciale pe tastatură, un proces care a terminat de executat, o eroare de acces la memorie, sau alte situații neașteptate.

Un semnal poate fi trimis unui proces în următoarele situații:

1. De la sistemul de operare către proces: Sistemul de operare poate trimite semnale către un proces pentru a-l notifica despre evenimente precum:

- Terminarea unui proces copil.

- Expirarea unui cronometru sau a unei alarme programate.

- O eroare de acces la memorie.

- Recepționarea unui semnal extern de la hardware (de exemplu, întreruperi de la dispozitivele de intrare/ieșire).

2. De la un proces către alt proces: Un proces poate trimite semnale către un alt proces pentru a comunica diverse stări sau acțiuni. Aceasta poate fi realizată utilizând funcții specifice oferite de sistemul de operare, cum ar fi `kill()` în sistemele bazate pe UNIX/Linux. De exemplu, un proces poate trimite semnale precum:

- Semnal de întrerupere (SIGINT) pentru a solicita oprirea unui proces în mod voluntar.

- Semnal de terminare (SIGTERM) pentru a solicita întreruperea unui proces și eliberarea resurselor aferente.

Procesele pot gestiona semnalele primite prin intermediul funcțiilor de tratare a semnalelor, cum ar fi `signal()` sau `sigaction()`, care permit procesului să specifice acțiuni specifice care trebuie luate atunci când un anumit semnal este primit.

Este important de menționat că tratamentul semnalelor poate varia în funcție de sistemul de operare și de configurarea specifică a procesului. Unele semnale pot fi ignorate, pot fi capturate și tratate de către proces, sau pot duce la întreruperea sau terminarea forțată a unui proces în cazul unor erori grave sau semnale de terminare.

3) Cine trimite un semnal unui proces?

Sistemul de operare, alt proces sau chiar el însuși.

4) Cum este implementat operatorul | din shell?

`|` este implementat similar cu `pipe()` din C: este folosit un buffer comun în care primul

proces trimite informație, iar cel de-al doilea o primește.

Cele două capete ale buffer-ului corespund celor două procese care comunică.

5) Care sunt avantajele și dezavantajele folosirii memoriei partajate pentru comunicarea inter-proces?

Avantaje: mai rapidă decât orice alt tip de comunicare, ieftină ca memorie, este evitată

folosirea și copierea între multiple buffer-e.

Dezavantaje: sincronizarea trebuie făcută "de mână" de către programator

https://www.cs.yale.edu/homes/aspnes/pinewiki/InterProcessCommunication.html

6) Care sunt avantajele și dezavantajele pipe-urilor pentru comunicarea inter-proces?

Avantaje: comunicare facilă și relativ rapidă între procese înrudite (pentru pipe-uri anonime) sau neînrudite (pipe-uri cu nume). Dezavantaje: comunicarea nu e bidirecțională (există doar sender și receiver) și dimensiunea limitată a buffer-ului

7) Care este limitarea folosirii pipe-urilor anonime?

Pot întreține comunicarea doar între procese înrudite. De asemenea, sunt dependente de procesele asociate. Un pipe anonim este șters după terminarea procesului, spre deosebire de FIFO-uri care au un back-up în sistemul de fișiere virtual.

8) Care este diferența principală între FIFO și socket UNIX?

FIFO-urile au un singur canal de comunicare (unidirecțional), pe când socket-urile UNIX prezintă două buffer-e (comunicare bidirecțională), unul pentru send și unul pentru receive.

9) Ce primitive de interacțiune pot fi folosite între procese de pe sisteme diferite?

Doar socket-uri de rețea.

10) Ce primitive de comunicare pot fi folosite doar între threaduri?

Doar shared memory

→ mutex, semaphore, variabile conditionale, bariera, coada sincronizata

11) Ce primitive de comunicare pot fi folosite doar între procese înrudite?

Doar pipe-uri anonime.

12) Cum pot două procese să folosească semafoare anonime?

Două procese înrudite pot folosi același semafor anonim dacă la crearea acestuia este pus 1 pentru argumentul de `pshared`.

13) Cum pot două procese neînrudite să folosească semafoare anonime?

Două procese neînrudite pot folosi același semafor anonim dacă acesta este plasat într-o

zonă de memorie pe care procesele o share-uiesc.

https://stackoverflow.com/a/32207582

14) Ce rol are apelul mmap() în cazul partajării memoriei între două procese?

`mmap()` (prin folosirea flag-ului `MAP_ANONYMOUS|MAP_SHARED`) creează zone de memorie anonime ce pot fi folosite de procese înrudite. Mai mult, map-ează fișiere ce folosesc atât mecanismele de demand paging cât și de lazy loading pentru o folosire cât mai eficientă a memoriei.

Demand paging (încărcare la cerere) și lazy loading (încărcare leneșă) sunt două concepte legate de gestionarea memoriei în sistemele de operare și au o legătură cu funcția `mmap()`.

Demand paging se referă la o tehnică utilizată în sistemele de operare pentru a gestiona eficient memoria virtuală. În cadrul demand paging, paginile din spațiul de adrese virtuale al unui proces nu sunt încărcate în memoria fizică în momentul inițializării procesului, ci sunt încărcate doar atunci când este necesară, adică atunci când se face o referire la o pagină care nu este încă prezentă în memorie.

Atunci când o referință la o pagină virtuală este făcută, dar pagina respectivă nu este încărcată în memorie fizică, se produce o excepție numită "page fault". Sistemul de operare intervine și încarcă în memorie fizică pagina solicitată din spațiul de adrese virtuale al procesului. Astfel, doar porțiuni ale memoriei virtuale care sunt efectiv utilizate sunt încărcate în memorie fizică, economisind spațiu și resurse.

Lazy loading se referă la o strategie specifică de demand paging, care implică încărcarea leneșă a resurselor. În cazul lazy loading, resursele (cum ar fi fișierele sau bibliotecile) asociate unei pagini virtuale nu sunt încărcate în memoria fizică în momentul inițial al încărcării paginii. În schimb, aceste resurse sunt încărcate în memorie doar atunci când sunt efectiv necesare, adică atunci când se face o referință la acestea. Aceasta permite economisirea resurselor și o inițializare mai rapidă a procesului.

Funcția `mmap()` este o funcție specifică în sistemele de operare care permite maparea unui fișier în memoria virtuală a unui proces. Aceasta poate fi utilizată în implementarea demand paging și lazy loading. Prin utilizarea `mmap()`, un fișier poate fi asociat cu o regiune de memorie virtuală a procesului și poate fi încărcat în memorie fizică doar atunci când se face o referință la acea regiune.

De exemplu, atunci când un fișier este mapat folosind `mmap()`, paginile corespunzătoare fișierului nu sunt încărcate în memorie fizică în mod imediat. În schimb, paginile sunt încărcate la cerere, adică atunci când se face o referință la acele pagini din memoria virtuală a procesului. Acest lucru este posibil datorită demand paging și lazy loading implementate în sistemul de operare.

Astfel, `mmap()` poate fi utilizat în implementarea eficientă a încărcării la cerere și încărcării lene

șe într-un proces, economisind resurse și permițând accesul la resursele necesare doar atunci când sunt solicitate.

15) Ce mecanisme putem folosi pentru a asigura accesul exclusiv la o zonă de memorie partajată între două procese?

Sincronizarea proceselor este mecanismul prin care asigurăm acces exclusiv: vom folosi mutex-uri (reținute în zone partajate de memorie) sau semafoare (mult mai ușor cu semafoare, pentru că putem folosi semafoare cu nume).

16) De ce este nevoie de sincronizare la folosirea memoriei partajate, dar nu la folosirea unui socket pentru comunicarea între două procese?

Memoria partajată poate fi accesată simultan de mai multe procese, ceea ce poate duce la conflicte și probleme de integritate a datelor. (race conditions) De aici și nevoia de sincronizare. Comunicarea între două procese prin intermediul socket-ului este efectuată în mod natural sincronizat, pentru că transmisia de date are loc secvențial, ordonat.

17) Ce înseamnă aplicație omogenă? Dați un exemplu.

O aplicație omogenă folosește thread-uri/procese pentru a realiza același tip de acțiune concomitent ("Multiple threads/processes doing the same work"). De obicei, astfel de aplicații dau dovadă de interacțiune minimală între thread-uri/procese, acestea doar dau join/wait la finalul execuției job-ului. ex: multi-threaded web servers, `libx264` din `ffmpeg` (thread-uri), paginile de browser din firefox (thread-uri), paginile de browser din chrome (procese). https://github.com/open-education-hub/operatingsystems/blob/master/content/chapters/appinteract/lecture/slides/classification.md#single-process-multi-threaded-homogeneous

18) Ce înseamnă aplicație eterogenă? Dați un exemplu

O aplicație eterogenă folosește thread-uri/procese pentru a realiza tipuri diferite de acțiuni concomitent ("Multiple threads/processes, each doing different work"). De obicei, astfel de aplicații dau dovadă de interacțiune bogată între thread-uri/procese, sunt folosite mecanisme de IPC(inter process comunication).

19) Care este avantajul și dezavantajul distribuirii unei aplicații pe mai multe sisteme?

Avantaje: se evită încărcarea sistemului propriu, putere computațională mai mare (sisteme distribuite). Dezavantaje: comunicare mai greoaie între componente (se pot folosi doar sockets de rețea, mai scumpi ca timp)

20) Care este avantajul implementării unei aplicații în format multi-process față de format multi-threaded?

Securitate: procesele nu pot accesa facil memoria unui altuia.

Robustețe: dacă un proces crapă, nu crapă tot programul (așa cum s-ar întâmpla la

thread-uri).

Scalabilitate: poți avea mai multe procese ale aceleiași aplicații rulând concomitent pe mai

multe sisteme.

21) Cum pot două procese partaja memorie anonimă? Adică folosind flagul MAP_ANONYMOUS la mmap()

* map-are și zeroizarea unei zone distincte de memorie

* copiii moștenesc zonele map-ate ale părinților

* nu există copy on write când unul dintre procese încearcă să scrie

* zonele map-ate anonim sunt folosite ca mecanism de IPC(inter process comunication),

ceea ce numim shared memory

Mai putem map-a anonim memorie și prin deschiderea lui `/dev/zero` pentru citire, după

cum urmează:

```C

fd = open("/dev/zero", O_RDWR);

addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

```

https://stackoverflow.com/a/39945292

22) Ce mecanism de interacțiune între aplicații au un buffer și ce mecanisme de interacțiune nu au un buffer?

* cu buffer: pipes, sockets

* fără buffer: fișiere, shared memory, mutex-uri, semafoare, semnale

Alte Intrebari

cum ppot face acces exclusiv intre 2 procese pe o zona de memorie partajata

cu ce primitva

ori semafor cand am o conexiune intre procese ori folosesc file locking, loc pe fisiere.

cand un apel write pe un socket se blocheaza?

cand bufferul de pe socket este plin

in ce situatie read pe un fisier se blocheaza si nu se blocheaza

se blocheaza cand

procesorul ofera ISA

sistemul ofera system call API

cate tabele de pagini sunt intr un sistem de operare?

una pt fiecare proces

pagini valide sau nevalide in tabela de pagini

pagina nevalida – a fost rezervata dar inca nu are un frame, nu e backed up pe memoria ram

aslr — feautre de securitate in care loaderul pune la adrese diferite segmentele executabilului

vrem pt a spori securitate, un atacator nu stie exact unde se afla zona de text sau zona de data ca sa poata suprascrie zona respectiva

alte mecanisme de securitate

stack canary care pune un canar pe stiva, iar la buffer overflow da abor

electric fens — inainte si dupa pagina ta, o pagina cu permisiuni 0 –

un shared code e un cod pe care pot sa l scriu intr o zona de date si pot sa pun ip ul acolo ca sa il rulez

zone ale unui executabil:

statice: data text rodata bss

dinammice: heap stack

permisiunile unei zone de memorie sunt tinute in tlb si in VAS

copy a.txt b.txt

copy a.txt b.txt

al doilea e mai rapid ca exista chestia asta de cacheing

la prima operatie se pune direct in cache, iar la a doua il ia direct din cache

cu ce e ocupat cel mai mult memoria fizica?

cu cache, memoria sistemului de operare care include cache, memoria proceselor, sistem de fisiere

ce e un sistem de fisier virtual?

ex procfs, tmpfs

tls = spatiu alocat pe stiva threadului specific fiecarui thread

tcb thread control block vs pcb process control block

in pcb gasim tabela de pagini si fdt, iar in tcb nu

pe un sistem multithreading daca nu avem sincronizare am avea race conditions

– facem o suma si nu ne da cat treb

– putem obtine si segm fault (avem un pointer, un thread scrie o adresa invalida si noi dereferniem)

dc shell code urile nu s asa de usor de folosit in sist moderne

ca atunci cand ai drept de write nu ai si drept de execute, si apoi trb sa schimb dreoturile ca sa poti rula

ce limiteaza nr de procese pe care l pot crea intr un sistem?

memoria

fork va da eroare not enough memory

ce limiteaza nr de threaduri pe care l pot crea intr un sistem?

memoria

spatiul virtual

cand fac bypass la overheaad apel de sistem?

cand vreau sa iau de pe un socket in alt socket, send file, zero copy

zero copy face copiere din kernel in kernel

intrerupere uzuala legata de procese : intreruperea de ceas, cand iti expira cuanta pe proces

cum se stabileste cuanta?

da, se poate modifca cuanta in timpul executiei unui proces, in caz ca procesul face anumite actiuni si SO ul poate detecta acest lucru.

ce e mai rapid sa faci?

fork sau exec?

fork face un proces nou

exec spune sa lanseze un cod, inlocuind tot continutul tabelelor de pagini, inlocuieste imaginea procesului

in mod normal exec, ar trebui sa ia executabilul, sa l puna in memorie si sa inlocuiasca in memorie in spatiul fizic toate informatiile vechiului proces

il va pune on demand paging acolo il va pune cand va fi nevoie

in ce moment strcpy va genera tranzitie in kernel mode

daca memoria e on demand, accesarea ei genereaza page fault si se va rula page fault handlerul de kernel space

care utilitatea lui dup?

duplici si devine un ppiinter ca aceeasi cehstie,

mmu - sursa primara de generare de page fault uri

mmu - e folosit tot timpu

page fault handler e la nivelul sist de operare

minor page fault vs major page fault

major page fault – va dura mai mult si trb sa interactioneze cu alta componenta, cu harddiskul, cand avem si fenomenul de swap

spinlock nu e folosit in cazuri in care threadurile se blocheaza!!!

cuanta cum sa fie mai mica sau mai mare?

depinde de ce ne dorim

cuanta mai mare - produc mai mare

ccuanta mai mica - sa primeasca fiecare cate o parte de executie din procesor

cand apelul write se intoarce se garanteza ca datele au fost scrise in cache

write se face in buffer cache

bariera asteapta ca toate threadurile sa ajunga la un punct, pe cand semaforul poate porni si de la valori negative

functie entranta = functie care poate fi apelata din orice context de mai multe threaduri

strcpy

lock reentrant = poti sa faci lock de mai multe ori pe el

facem flush ca avem alt proces

la read daca datele nu sunt in cache ar trebui sa le ia din memorie si ar putea deveni blocant

gdb = analizator static, objdump = analizator dinamic

putem ajunge sa avem deadlock fara sa avem primitive de sincronizare??

NU

trb sa facem lock si celalt tot lock

care e avantajul folosirii executabilului static vs dinamic

static

++ mai putin spatiu

cand ne uitam la camp vedem blocuri ocupate si size

putem avea 0 blocuri si dim 3 milioane ? Da, putem avea truncate.

VM vs Container

VM partajeaza biblioteci comune cu SO ul gazda.

CTRL + C —> sig int → face o intrerupere

zona statica nu are de la inceput size ul cunoscut , iar cea dinamica da

ele se initializeaza la load time cand se incarca executbailul atunci se aloca heapul si stiva si

se pun acolo

apel de sistem -> overhead pt tranz din usermode in kernel mode

Open — seek start

Append —- seek end

Parse file este un fisier fragmentat cu niste gauri prin el

System — face fork si exec

Face mai multe apeluri de sistem

Fork dureaza mai mult ca face un pcb nou

O tabela de pagini

Exec doar schimba ceva

Fork copy on write

Exec demand paging

Read write fol buffered cache

E in kernel

Daca tabela de file descriptori se umple putem face mai multe procese ca fiecare proces are tabela lui sau daca esti privilegiat faci apel catre kernel

Shared memory intre 2 procese de pe un sistem

Shmopen sau open

Din user mode in kernel mode cand expira cuanta sau cand e intrerupere

Epoll e din io multiplexing

Shared memory e cel mai rapid mecanism de comunicare

Ca nu fac sys call uri

o zona de memorie are lungime diferita de alta zona, adresa diferita,

fifo e cea mai putin utila, ofera un prilej de comunicare bidirectionala