Stiva software

Ce este un apel de sistem?

Un apel de sistem este o cerere a unei aplicații, care rulează în user-space în mod neprivilegiat, către sistemul de operare pentru efectuarea unei operații privilegiate, care va rula în kernel space. Este singurul mod prin care software-ul poate genera întreruperi.
Mecanism care asigură trecerea din user space în kernel space la solicitarea user space.
Pentru buna functionare a unui sistem de calcul pe care se pot rula mai multe aplicatii este nevoie de un sistem de operare care sa medieze accesul la resursele sistemului (intre aceste aplicatii/procese). Sistemul de operare (mai precis kernelul) este ca o biblioteca, o componenta software care ofera o interfata nivelului superior, cel al aplicatiilor user. Un element din aceasta interfata este un apel de sistem. Apelul de sistem este invocat, ca o rutina, de catre nivelul aplicatiilor user pentru a accesa resursele sistemului.

De ce sunt necesare apeluri de sistem?

Pentru ca doar sistemul de operare să aibă acces în mod direct la resursele sistemului și pentru a permite acestuia sa controleze alocarea resurselor și modul în care acestea pot fi accesate.

User space-ul nu are privilegiile de a realiza o operație din motive de securitate și apelează la kernel space pentru acest lucru.
Apelurile de sistem sunt necesare pentru acelasi motiv pentru care sistemul de operare este necesar, sistemul de operare este necesar pe sisteme de calcul pe care se doresc a rula aplicatii multiple, apelurile de sistem sunt doar interfata acestui nivel, sunt necesare pentru a folosi sistemul de operare. Unul din rolurile sistemului de operare, pe langa cel de mediere, este acela de a proteja sistemul, tu ca user, ca aplicatie, pentru a folosi sistemul, pentru a accesa resursele sale trebuie sa treci prin sistemul de operare, trebuie sa faci aceste apeluri de sistem (nu poti sa faci un fel de bypass si sa accesezi singur ca nebunul ce resurse vrei). Acest lucru se intampla prin mecanismul de user space/kernel space. Aplicatiile ruleaza in user space, fara privilegii, in acest mod nu poti sa accesezi resursele sistemului, ce poti sa faci este un apel de sistem. Dupa apelare se trece in kernel space, unde se acceseaza resursele. Numai sistemul de operare are astfel acces la resurse, userii/aplicatiile trebuie sa faca apel la el.

Ce avantaj / dezavantaje au apelurile de sistem?

Avantaje - resursele sistemului sunt alocate doar cand este nevoie de ele.

Dezavantaje - overhead la trecerea din modul neprivilegiat în privilegiat.

Atunci cand realizezi un apel de sistem, un lucru pe care il faci este schimbarea de nivel de privilegiu (din user mode intri in kernel mode). Deci, un avantaj ca faci un apel de sistem este securitatea.

Dezavantajul este un mic overhead de performanta (ca sa faci un apel de sistem, se genereaza o intrerupere care cauta intr-o tabela care e apelul de sistem)

Un avantaj este ca la un apel de sistem noi apelam cod scris de altcineva, nu trebuie sa il scriem noi. Fara sistemul de operare, si implicit apelurile de sistem, ne-ar fi mult mai greu sa folosim calculatorul. Alt avantaj este securitatea sistemului, userii nu pot strica sistemul.

Un dezavataj este overhead-ul (trecerea din user space in kernel space) si eventual cod mai mult (mai multe instructiuni) decat ce ne este noua necesar intr-o anumita aplicatie (adica noi am scrie mai putine instructiuni pentru a face acelasi lucru ca apelul de sistem).

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

User mode este modul în care rulează, in general, un proces. Unui proces îi este atribuit un spațiu de adrese virtual care poate fi accesat doar de acel proces.

Kernel mode este modul in care se rulează sistemul de operare (dar nu partea de aplicații din SO, ci mai degrabă kernelul). In mod privilegiat ai acces la un set mai mare de instrucțiuni/registre decât în modul neprivilegiat.

User mode: in user mode codul nu poate accesa direct resursele sistemului. El trebuie sa faca un apel de sistem pentru acest lucru.

Kernel mode: in kernel mode codul are acces nerestrictionat la resursele sistemului. Doar codul din sistemul de operare are acest privilegiu.

Cum se realizează tranziția în mod privilegiat?

Modul este indicat de un bit de mod (1=user, 0=kernel). La bootare, hardware-ul se afla în kernel mode, iar SO se încarcă și aplicațiile pornesc în user mode. Cand apare o intrerupere (trap), se trece în kernel mode. În momentul în care se face apelul de sistem, se trece din user mode in kernel mode și bit mode devine 0. Se executa rutina asociată întreruperii în kernel mode și apoi se reintră în user mode, bit mode devenind 1.

Sistemul de operare expune o interfață (system call interface, system API) prin care aplicațiile solicită acțiuni privilegiate sistemului de operare.

Printr-un apel de sistem.

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

Aplicațiile solicită acțiuni privilegiate sistemului de operare. Sistemul de operare verifică dacă aplicația are permisiunile necesare, realizează acțiunea cerută și întoarce rezultatul. Sistemul de operare are rolul unui furnizor de servicii privilegiate pentru aplicații.

Se schimba un bit care specifica modul in hardware. CPU-ul, hardware-ul asigura existenta modului privilegiat. Daca in user mode incerci sa faci acces la memorie sau alte nebunii, hardware-ul nu te lasa si notifica sistemul de operare.

Ce este o bibliotecă?

O biblioteca este o colectie de functii precompilate. În momentul în care un program are nevoie de o funcție, linker-ul va apela respectiva funcție din bibliotecă.

Care este asocierea apel de bibliotecă / apel de sistem?

Un apel de biblioteca presupune zero, unul, sau mai multe apeluri de sistem.

De exemplu, pentru un scanf, se va face un read in spate daca datele ce trebuie citite nu sunt deja in bufferul mentinut de libc de la un read anterior. Altfel, nu se va face read.

Care este rolul bibliotecii standard C (libc)?

Ofera aplicatiilor wrappere peste apeluri de sistem pentru a putea fi apelate ca funcțiile dintr-o biblioteca. Acest lucru face codul scris cu functiile din libc portabil peste mai multe interfete de sistem/platforme.

Ce acțiuni se pot executa doar în mod privilegiat?

I/O, mapari si alocari de memorie, managementul timerelor si al intreruperilor, comunicarea cu alte procese, configurarea de permisiuni, pornirea/oprirea aplicatiilor

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

Opinie: Stiind ca atunci cand se modifica memoria este nevoie de modul privilegiat, load si store reprezinta un raspuns bun la intrebare.
Doar unele load-uri si unele mov-uri, conform:
Which are the instructions that can run only in kernel mode?

Ce este un sistem de operare monolitic?

Un sistem de operare monolitic este un sistem în care user services și kernel services sunt implementate în același spațiu de adrese (motiv pentru care este eficient). Este mai puțin flexibil, pentru ca atunci cand se face o modificarea asupra unei componente, trebuie schimbat tot. Este un sistem mai mare, intrucat kernelul este mai mare. Dacă un serviciu pica, pica toate. Toate functiile SO-ului se realizeaza prin apeluri de sistem.
Exemple: unix

Este o arhitectura de nucleu în care întreg nucleul se rulează în spațial nucleului și în mod administrator.

Caracteristici:

- distincție între user-space și kernel-space
  - aplicațiile rulează în user-space
  - nucleul și driverele rulează în kernel-space
  - între ele există un strat de apeluri sistem
- suportă sistem complexe cu multe aplicații
- oferă protecție între aplicații
  - dacă o aplicație se blochează sau funcționează incorect atunci nu afectează pe celelalte
- poate rula și pe arhitecturi fără MMU, dar cu protecție limitată

In sistemul de operare monolitic se afla mai multe componente, ceea ce da o performanta mai buna (totul este in kernel mode), un apel de sistem face mai multe (nu trebuie sa facem noi mai multe apeluri de sistem), dar totul este intr-un singur loc, suprafata de atac mai mare

Ce este un sistem de operare de tip microkernel?

Un sistem de operare de tip microkernel are user services si kernel services implementate în locuri diferite din memorie (motiv pt care este mai lent, serviciile fiind nevoite sa comunice între ele). Se poate extinde ușor și este un SO mic, kernelul fiind mic. Dacă un serviciu, fie el kernel sau user, pica, nu afectează alte servicii.

Caracteristici:

- teoretic cea mai performantă arhitectură de SO
- practic implementările existente prezintă prea multe limitări din cauza complexității
- toate funcțiile SO rulează în user-space în afara unui strat foarte subțire de message passing
- sistemul are un overhead mare datorită mulțimii de mesaje ce trebuie să treacă din user-space în kernel-space și invers

Sistemul de operare este mic, suprafata de atac este mica, performanta este si ea mai mica (avem nevoie de mai multe apeluri de sistem, mai multe tranzitii user-kernel)

Care sunt avantajele unui sistem de operare monolitic?

Executie mai rapida, necesita mai putin cod.

Avantaj pentru un nucleu monolitic este viteza de execuție (overhead-ul redus).

Dezavantaje pentru un nucleu monolitic sunt securitatea potențial mai slabă (întrucât kernelul este mai mare și are suprafață de atac mai mare, spunem că are TCB (Trusted Computing Base) mai mare); este mai puțin modular decât un microkernel însemnând o toleranță la defecte mai redusă și o proiectare mai greu extensibilă.

Care sunt avantajele unui sistem de operare de tip microkernel?

Se extinde mai ușor, dacă un serviciu crapă, nu afectează alte servicii.

Mai lent (comunicare intre servicii)

Componentizabil, flexibil, modular

TCB redus (design mai sigur)

Care tip de sistem de operare are mai multe apeluri de sistem?

Cel de tip **microkernel** pentru ca sunt mult mai multe tranzitii user-kernel pentru a permite componentelor software care rulează în user mode să interacționeze

Care este avantajul folosirii mașinilor virtuale din perspectiva securității?
Mașinile virtuale (inclusiv sistemul de operare al acestora) nu este parte din TCB. Sunt și un mediu relativ izolat.

https://docs.google.com/document/d/e/2PACX-1vS10vLsN9jtxbYkrkBE7p3MKm0fq0Yk22_qfasxRlrRgGVjegDnD3dIvvc2cj_d83suM0_H2CEJTn9z/pub

Ce este o bibliotecă statică? Ce este o bibliotecă dinamică?
O biblioteca statica (.a pe Linux, .lib pe Windows) este o colectie de functii link-ata la compile time. Biblioteca statica face parte din executabilul obtinut.
O biblioteca dinamica(.so pe Linux, .dll pe Windows) este o colectie de functii link-ata la load time.

Când preferăm folosirea static linking, respectiv dynamic linking? Cu ce diferă un executabil dinamic de un executabil static?
Static linking = se obține un executabil static, de dimensiune mare (ex: un program care doar printeaza “hello world” are aprox 800 K, pt ca el trebuie sa aducă tot printf-ul si toate funcțiile folosite de printf); un proces obtinut dintr-un executabil static nu poate partaja nicio parte din zona sa cu alte procese obținute din executabile statice.
Dynamic linking = se obține un executabil dinamic, care nu contine toate secțiunile și menține referințe nerezolvate, acestea rezolvandu-se la load time, când se creeaza procesul corespunzător; procesele obținute din executabile dinamice partajează zonele read-only (.text, .rodata)
Un executabil static e mai mare ca dimensiune deoarece are toate bibliotecile incluse.
La faza cu folosirea static linking in loc de dynamic - daca ai un executabil obtinut prin static linking, poti sa il distribui mai usor, in sensul ca e ceva compact, care are tot codul de care are nevoie ca sa ruleze, nu mai are nevoie de nimic la runtime...nu stiu exact cum sa formulez si daca stie altcineva alte motive, pls completati <3

Executabilul static este mai rapid pentru ca nu mai face linking la load time.

Dați exemplu de apel de sistem blocant.
read() - se blocheaza pana poate fi citit cel putin un byte
write() - se blocheaza pana s-a putut scrie intregul buffer

Dați exemplu de apel de sistem neblocant.
send() - nu se blocheaza daca nu se incearca trimiterea mai multor bytes decat incap in buffer si daca nu este configurat anume ca sa se blocheze
Open, close, seek+

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 ca trebuie sa fie intr-un mod privilegiat (apelul de sistem executat cu succes asigura aceste privilegii).
La a doua cred ca se refera la faptul ca trebuie sa fie mapat in memorie pentru a putea realiza comunicatia.

Ce înțelegem prin overhead spațial și overhead temporal?
Spatial - de stocare, de exemplu cand apelezi functii recursiv generezi overhead spatial + bufferare de diverse chestii.
Temporal - in general timp pierdut “degeaba in timp ce astepti ceva/dupa ceva”, exemplu: cat astepti sa se elibereze un lock

Dați exemplu de mecanism / funcție care reduce overhead-ul spațial și unul care reduce overhead-ul temporal.
Mecanismul de zero-copy reduce overhead-ul spatial. Pentru a face read sau write exista 2 buffere, unul in user space si unul in kernel space (overhead spatial, 2 buffere pentru acelasi lucru). Ne dorim sa scapam de unul din ele, de ala din kernel nu putem scapa. Mecanismul de zero-copy inseamna sa facem bypass la bufferul din user space, si sa folosim doar buffer-ul din kernel space. Acest lucru putem sa il facem doar cand nu este nevoie sa prelucram datele (nu avem nevoie de ele efectiv in program, vrem doar sa le transferam dintr-o parte in alta). Functii pentru zero-copy sunt de exemplu splice si sendfile pe linux.

Pentru a reduce overhead-ul temporal ne dorim sa facem mai putine apeluri de sistem (pentru ca stim ca apelurile de sistem aduc overhead, in special tranzitiile user-kernel). Pentru asta in loc sa facem write(1) de 10 ori, facem write(10) o data. Se numeste ca facem buffering, adica avem date de scris, dar nu le scriem, le bufferam, cand bufferul e plin atunci le scriem. Operatiile fread si fwrite din libc fac asta.

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

Producer consumer:

Double buffering = tehnica prin care 2 buffere sunt folosite pentru cresterea performantei. Astfel procesarea unor date se poate face independent de aducerea altora: in timp ce se lucreaza cu datele dintr-un buffer, se pot pregati date in altul.

Exemplu:

single buffering: SO vrea sa scrie date catre disk care e lent si ca sa nu stea dupa el scrie intr-un buffer si le ia in ritmul lui.

Problema: are bufferul plin si nu poate sa-l foloseasca pentru altceva.

Solutie: double buff: mai ia inca unul ca sa ai unde sa scrii pana discul ia ce-i trebuie.

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

Hardware-ul notifica sistemul de operare si se apeleaza o rutina din sistemul de operare. Sistemul de operare trebuie sa trateze eroarea.

Scheduling

Ce este un proces?

Un proces este o instanță a unui program în execuție si e o abstractizare facuta de sistemul de operare pentru o serie de actiuni din ratiuni de multitasking si pentru securitate.

Un **proces** este un program în execuție. Procesele sunt unitatea primitivă prin care sistemul de operare alocă resurse utilizatorilor. Orice proces are un spațiu de adrese și unul sau mai multe fire de execuție. Putem avea mai multe procese ce execută același program, dar oricare două procese sunt complet independente.

Procesul este incapsularea unei actiuni in sistemul de calcul. E o abstractizare a sistemului de operare pentru a face actiuni.

Un proces este abstractizarea a trei lucruri: memorie prin memoria virtuala, procesor prin threaduri si I/O prin tabela de file descriptori.

Ce este un thread?

Un thread (fir de executie) este unitatea de planificare (pe Windows e asa; pe Linux procesu' e si un thread, e vazut tot la nivel de proces). Un proces este format din unul sau mai multe thread-uri.

Fir de execuție (sau thread) este unitatea elementară de planificare într-un sistem. Ca și procesele, firele de execuție reprezintă un mecanism prin care un calculator poate să ruleze mai multe task-uri simultan.

Un thread poate fi vazut ca un tuplu de forma (Instruction Pointer, Stack). El are al sau doar registrele si stiva / memoria sa proprie TLS parca ii zice :(.

Cu ce diferă un thread de un proces?

Un proces are unul sau mai multe threaduri. Procesele nu partajează resurse între ele, pe când firele de execuție partajează în mod implicit majoritatea resurselor unui proces (poate merge zis si pe care). -> practic toata memoria procesului in afara de registrele pe care le are fiecare

Daca un proces intampina o problema, nu afecteaza celealte procese. Daca un thread crapa, tot programul crapa.

Threadurile sunt mai eficiente decat procesele ca si timp de creare si schimbari de context (daca se cer).

Threadurile in general au nevoie de sincronizare explicita.

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

Se mai aloca memorie pentru stiva noului thread (standard 8MB, dar se poate modifica)

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

- segmentele de memorie precum .heap, .data și .bss
- nu-s in memorie. Au comun tot in afara de stiva si TLS si program counterul (au acces si unu' la stiva altuia daca vor ce-i drept daca isi transmit stack pointerul intre ele prin variabile globale, dar destul de greu sa faci ce trebuie doar daca ii vezi stiva ca nu stii ce e fiecare si poti sa ii crapi programul).

Ce conține PCB (Process Control Block)?

Atributele unui proces (PID, spațiu virtual de adrese, timp de lucru pe procesor, fișiere deschise, FDT(file descriptor table), user/group, starea unui proces) sunt păstrate într-o structură numită PCB sau TCB(pt ca orice proces e un thread).

Care sunt stările în care se poate găsi un proces/thread?

Un proces se poate afla în 5 stări posibile: READY, RUNNING si WAITING / BLOCKING. + NEW si TERMINATED

Ce efect are apelul fork()?

Apelul fork() creează un proces copil ca fiind o copie a procesului părinte + ca returnează de 2 ori, și despre copy on write.

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

VAS-ul copilului va fi identic cu VAS-ul parintelui la creare (prin sistemul copy-on-write ele vor referi aceleasi pagini fizice), VAS-ul insa este propriu, orice modificare a copilului nu se va vedea in parinte si invers. FDT-ul este si el duplicat, insa fd-urile din FDT-ul copilului vor pointa catre aceleasi structuri de fisier deschis ca si parintele.

Copilul primeste o copie a file descriptorilor, care pointeaza catre acelasi fisier, dar daca inchizi un fd din copil, nu-l inchide si in parinte. Ca sa se intample asta trebuie sa si transmita pointerul catre structura de fisier deschis

Procesul copil si parinte partajeaza pe toata durata vietii (atata timp cat imaginea copilului nu e modificata) de zonele care nu sunt writable .text, data, bss (rodata) pentru ca n-are sens sa le cloneze daca sunt identice

Ce efect are apelul exec()?
Lanseaza in executie procesul creat cu fork(). Fork() a creat copilul ca o copie a parintelui. Exec() ne va lasa sa “suprascriem” acea copie cu orice dorim noi sa lansam spre executie. Exec() incarca imaginea procesului copil dintr-un executabil in procesul curent. Loaderul incarca imaginea in memorie si intoarce un pointer la imagine la kernel.

Câte procese se pot găsi în starea RUNNING, READY și WAITING?
1. În starea RUNNING se găsesc procesele care execută cod pe procesor. Numărul maxim de procese în acea stare este dat de numărul de procesoare.
2. În starea READY, respectiv WAITING se pot găsi oricâte procese în limita resurselor sistemului. În READY se vor găsi procese gata de rulare, neblocate, care așteaptă acordarea de timp pe procesor; în starea WAITING se vor găsi procese blocate în așteptarea unei acțiuni de deblocare (dispozitiv de I/O, semafoare cozi de mesaje). Nu există o limitare dată pentru procesele din starea READY sau WAITING.

Ce este o schimbare de context? Ce se întâmplă la o schimbare de context?
Se referă la schimbarea unui proces care rulează pe un procesor (este în starea RUNNING) cu un alt proces (aflat în starea READY). Este necesară pentru a asigura folosirea optimă a procesorului (dacă un proces se blochează îi ia altul locul) și pentru asigurarea echității (fairness) a sistemului (procesele se schimbă cu altele pentru a permite câtor mai multe să ruleze în sistem).
Schimbare de context intre P1 si P2 : P1 salveaza starea in PCB ul lui, P2 o restaureaza din al sau.

Ce cauzează schimbări de context?
5 conditii principale care iti genereaza context switch:
1) RUNNING → READY - schimbare voluntara (yield) // what? Yield inseamna sa plece singur de pe procesor. Cum sa faca yield daca trece in running? Yeah, le-am scris invers, my bad
2) RUNNING → BLOCKED (waiting) - schimbare voluntara, o actiune executata explicit de procesor are nevoie de resurse (operatii blocante sau I/O o declanseaza)
3) RUNNING → READY - schimbare nevoluntara (apare un proces candidat mai bun, prioritar)
4) RUNNING → READY - schimbare nevoluntare (eg expira cuanta)
5) RUNNING → TERMINATED - aici depinde ce fel de schimbare e dupa cum s-a terminat. Este scos de pe procesor.

Procesele IO intensive din running vor avea tendinta sa tranziteze in BLOCKED iar cele CPU intensive in READY.

Ce este o schimbare de context voluntară și o schimbare de context nevoluntară?
Ramane valabil ce e la intrebarea anterioara de adaugat ca 3 si 4 sunt nevoluntare si tin de faptul ca “asa a dictat schedulerul sa se intample”.

Ce sunt thread-urile cu implementare user-level și thread-urile cu implementare kernel-level?
User threadurile sunt vazute la fel ca si procese la nivel de kernel. Se poate ca mai multe user level threads sa corespunda unui kernel thread unic. Kernel threads au suport in kernel pentru creare, planificare si join.
In c, cele kernel, sunt alea din pthread, daca vrei user-level threads trebuie sa ti le implementezi singur, exista mai multe mecanisme din kernel care asigura fiecare thread ca ruleaza cum trebuie, one to many, many to many etc

În ce situație este utilă zona TLS (thread local storage)?
Cand dorim sa avem variabile globale, astfel incat sa fie accesate de toate functiile, dar nu dorim ca acele variabile sa fie partajate intre thread-uri.
Cand nu te intereseaza sincronizarea si vrei sa o eviti pe cat posibil, ea fiind o operatie costisitoare dpdv al resurselor

De ce este necesară sincronizarea proceselor/thread-urilor?
Pentru a ne asigura ca nu exista race condition-uri sau situatii in cazul in care programul are caracter nedeterminist.

Ce înseamnă race condition?
Evenimentul care are loc atunci cand 2 sau mai multe threaduri incearca sa acceseze aceeasi resursa comuna si sa o modifice simultan. Exemplul cu a = 0, facem a++ care poate sa rezulte in a=1 sau a=2.

Ce înseamnă deadlock?
Cand 2 sau mai multe threaduri asteapta simultan unul dupa altul. Ai exemplul clasic cand A are lock pe Lock1 si asteapta dupa Lock2 si B are Lock2 si il asteapta pe Lock1.

Ce înseamnă livelock? Cum diferă de un deadlock?
Livelock = forma de deadlock cu spinlock
Livelock se produce atunci cand doua sau mai multe procese care se afla in starea RUNNING ajung intr-un ciclu, deoarece le este refuzat accesul la un lock comun, iar acest lucru neputand duce la finalizarea task-ului.
In cazul unui deadlock, procesele se afla in starea WAITING in incercarea de a acapara un mutex.

Care sunt dezavantajele sincronizării?
Implementarea sincronizarii poate fi **incoerenta**(deadlock, asteptari nedefinite), implementari **ineficiente**(regiune critica foarte mare=>cod serial, lock contention<1 sau mai multe threaduri incearca sa obtina lockul altui thread >, thundering heard< se trezesc toate threadurile, toate incearca sa obtina lockul, doar unul reuseste si restul se blocheaza pana la urmatorul apel de unlock/notify)>), **overhead mare** (de apel sau de asteptare).

Ce înseamnă TOCTTOU (time of check to time of use)?
Daca incerci sa mentii sincronizarea utilizand varaibile (gen sa te iei dupa valoarea unei variabile ca sa faci ceva sincronizat), poate aparea o greseala din cauza TOCTOU. Tu practic poti sa determini valoarea X pt variabila la momentul la care o verifici, dar la momentul la care o folosesti, alt thread o modifica si nu mai e X, e Y. => compare_and_swap (atomica)

Exemplu:

Thread 1:
```
a = 0;
while(1)
   if (a == 2)
      printf("a = %d\n", a);
```

Thread 2:
```
a = 0;
while(++a);
```

Valoarea afisata nu va fi neaparat 2. Intre verificarea din if si afisare, Thread 2 poate face oricate incrementari in plus.

Când se blochează un producător în problema producător-consumator? Dar un consumator?
Producatorul se blocheaza cand bufferul e plin si asteapta eliberarea bufferului pentru a putea produce in continuare.
Consumatorul atunci cand bufferul este gol. Va fi trezit cand a sosit un produs in buffer.

Cum se implementează pe un sistem single-core un spinlock? Cum se implementează pe un sistem multi-core un spinlock?
https://en.wikipedia.org/wiki/Spinlock#Example_implementation - single core (cred)
Pe single core -> cu compare and swap care are suport in hardware.
Pe multicore cred ca pui lock pe magistrala de date ptc magistrala ta e partajata. + pleci de la premisa ca spinlock e atomica single core

De ce este necesară prezența unei instrucțiuni de tipul atomic_compare_and_swap în fiecare ISA?
Atomic CAS este operatia pe care se bazeaza toate celelalte operatii de sincronizare. Prin CAS se implementeaza intai spinlock-ul, prin spinlock mutex-ul, samd. + CAS este suportul din hardware al spinlockului

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

Spinlock-ul folosește busy-waiting și are operații de lock() și unlock() ieftine prin comparație cu mutex-ul. Operațiile de lock() și unlock() pe mutex sunt de obicei costisitoare întrucât pot ajunge să invoce planificatorul. Având operații rapide, spinlock-ul este potrivit pe secțiuni critice de mici dimensiuni în care nu se fac operații blocante; în aceste cazuri faptul că face busy-waiting nu contează așa de mult pentru că va intra rapid în regiunea critică. Dacă am folosi un mutex pentru o regiune critică mică, atunci overhead-ul cauzat de operațiile pe mutex ar fi relativ semnificativ față de timpul scurt petrecut în regiunea critică, rezultând în ineficiența folosirii timpului pe procesor.

Deoarece spinlock-urile folosesc busy waiting, sunt recomandate doar pentru secțiuni critice mici, ce se execută rapid.

Regiunile critice cu operații de I/O sunt, de obicei, lungi. De asemenea, operatiile I/O pot duce la blocarea thread-ului, caz în care nu poate fi folosit spinlock-ul.

Ce efect are folosirea operatorului & din shell în crearea unui proces?
Il trimite in background(il face daemon) si ~~ruleaza maxim pana ce s-a inchis consola.~~
Trimite procesul creat in background.

“dacă rulăm o comandă cu & la sfârșit (pentru rulare în background), shell-ul nu așteaptă încheierea comenzii” - Notite Cap 3 pai daca nu comunica nu mai are de ce sa mai astepte

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, dar care nu a fost așteptat de procesul său părinte. El inca tine informatii despre cum s-a terminat, de aceea nu dispare complet, pentru ca parintele sa aiba sansa sa faca wait si sa afle informatii despre cum s-a terminat. O problema ar fi faptul ca ei continua sa ocupe spatiu in sistem (procesele zombie desi sunt terminate inca ocupa spatiu in tabela de procese a sistemului de operare si in cazul multor astfel de procese si tabela plina, sistemul de operare nu va putea crea procese noi).

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

Un proces al cărui părinte s-a terminat poartă numele de **proces orfan**. Acest proces este adoptat automat de către procesul init, dar poartă denumirea de orfan în continuare deoarece procesul care l-a creat inițial nu mai există. Nu cred ca el mai e numit orfan dupa adoptarea sa de catre procesul init. De aia nici nu e orfan prea mult timp, ca e adoptat imediat de init.

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

Da. Parintele isi asteapta copilul insa se intampla unul din cele 2 scenarii: moare parintele SAU copilul moare inainte ca parintele sa il astepte // aici cred ca e formulat prost raspunsul...nu se poate cu SAU. ca sa fie si zombie si orfan trebuie asa: 1. Sa fie zombie, adica sa sa termine procesul si sa nu mai fie asteptat de parinte 2. Sa fie apoi orfan, adica sa se termine si parintele. Nu vad de ce ar fi acel SAU acolo. Dap, trebuie sa fie zombie si dupa procesul parinte sa moara. Daca se intampla asta este adoptat de procesul init si eliminat din sistem (o sa dispara). // Init va primi datele pe care voia sa le returneze copilul

Ce se întâmplă dacă un thread realizează un acces nevalid la o zonă de memorie?

Segfault si procesul crapa (adica e omorat)
In cazul in care se face demand paging, la accesul nevalid al memeoriei, se genereaza un page fault, iar daca nu are permisiunile necesare pentru a accesa acea zona din memorie, procesul este terminat cu Segmentation Fault. In caz contrar, sistemul de operare determina ca pagina era rezervata si astfel aloca pagina necesara procesului si se mapeaza in memoria virtuala.
Primeste SIGSEGV si se apeleaza rutina de tratare a exceptiei.

Poate un thread să acceseze stiva altui thread? Cum?

Trebuie ca threadurile sa fie ale aceluiasi proces.
Da. Nu exista niciun mecanism care sa nu ii permita unui thread sa acceseze orice zona din memorie, inclusiv stiva altui thread. Ii trebuie cumva leak-uite adrese din alta stiva pentru a face asta. Da, nu stiu siguuur, ideea e ca asa a zis Radovici la curs si nu am facut inca research:))
Da, cu pointeri.
Folosesti variabile globale in care pui pointerul de la stiva.

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?

Pt ca doua thread-uri care apartin aceluiasi proces share-uiesc memoria. Nu se da flush la TLB.

Pentru ca atunci cand schimbi 2 threaduri din 2 procese diferite schimbi implicit si procesele intre ele => flush la TLB ca si prima consecinta. ++ e faptul ca o schimbare de context intre 2 threaduri din acelasi proces are overhead considerabil mai mic decat schimbarea pt 2 procese.

Ce forme de comunicare inter-proces cunoști?
Semnale - merg pe principiul de notificari, mereu ai un handler de semnal care ruleaza cand se primeste un semnal(il trateaza). + sunt ca intreruperile de la PM, gen ai un context, vine notificare, salvezi contextul, rulezi handler, restaurezi contextul si reiei executia de unde ai lasat-o
Pipe - "comunici pe teava", e unidirectionala, sunt cele **anonime**(intre procese inrudite de ex copilul comunica (,) cu parintele) si **cu nume** (nu mai ai restrictii legat de ce procese pot comunica unele cu altele). Mai e si **|** din terminal.
Memorie partajata - mai multe procese au acces la ea, ai nevoie de sincronizare
Socketi - de facto cei Berkely // sau UNIX
Comunicare prin mesaje - exemplu: MPI

Ce este un semnal? Când se trimite un semnal către un proces?
Notificarea asincrona pe care i-o trimiti unui proces sau unui thread specific al sau

Cine trimite un semnal unui proces?
Sistemul de operare, un alt proces sau el insusi.

Cum este implementat operatorul | din shell?

```
int fd[2];
pipe(fd);
int pid = fork();
if (!pid) {
   dup2(fd[0], 0);
   close(fd[1]);
   exec();
} else {
   dup2(fd[1], 1);
   close(fd[0]);
   exec();
}  ---ma indoiesc ca se asteapta sa ii povesteti cod
```

Tl;dr folosesti dup2 in implementare pt redirectare; ai un pipe anonim(capat citire si capat scriere) Outputul primului proces este inputul celui de al doilea. Primul proces scrie prin capatul de scriere al pipeului si cel de-al doilea citeste prin cel de citire.

Care sunt avantajele și dezavantajele folosirii memoriei partajate pentru comunicarea inter-proces?
Avantaj: Mai putina zona de memorie folosita; mai rapida comunicarea prin memorie decat prin mesaje/socketi
Dezavantaj: Sincronizarea, in lipsa ei se pot genera date incoerente, nu poti trimite pointeri.

Care sunt avantajele și dezavantajele pipe-urilor pentru comunicarea inter-proces?
Nu este nevoie de sincronizare :: Comunicarea se face printr-un canal unidirectional

Avantaje: comunicare intre procese inrudite, daca sunt cu nume poti si intre procese total diferite. | e un mod super easy de redirectare in linia de comanda
Dezavantaje: comunicare unidirectionala; destul de basic(poti doar sa scrii si sa citesti)

- Nu poti mapa fisiere prin pipe in memorie
- Au capacitate maxima per pipe, mai mica, si daca dai write si pipe-ul e full o sa ramana procesul in blocked pana cand e loc sa scrii acolo

Ce se întâmplă când toate procesele sistemului sunt blocate?
Pe fiecare procesor se planifica un proces special numit IDLE (care face busy waiting) daca nu exista alt proces care poate fi planificat. (Instructiunea HLT din x86)

Ce înseamnă waiting time (timp de așteptare) în planificarea proceselor?
Noțiunea de waiting time se referă la timpul de așteptare al unui proces în coada READY a planificatorului. Pentru un sistem interactiv/responsiv este de dorit ca timpul de așteptare să fie cât mai scurt.

Putem avea un sistem multi-core cu un singur proces aflat în starea RUNNING și mai multe procese în READY?
Nu, totate core-urile vor fi ocupate. Singurul motiv pentru care un proces ar fi in starea READY e ca nu are un procesor available, adica toate core-urile sunt deja ocupate.
Aici cred ca raspunsul e da, pentru ca spune ca un singur proces este in starea RUNNING, asta inseamna ca acel proces poate avea mai multe thread-uri implementate la nivel de kernel si pot fi prioritare fata de thread-urile celorlalte procese aflate in starea READY. Astfel, mai multe thread-uri ale aceluiasi proces vor rula pe core-urile procesorului, iar celelalte procese vor ramane in starea READY pana la preemptie.
Adevarul e ca thread-urile sunt planificate de scheduler, nu procesele. Thread-urile trec prin starile respective. Cred ca intrebarea abstractizeaza thread-ul cu proces single threaded. Procesul nu poate avea stare in context multi-threading pentru ca ce stare are un proces pentru care un thread ruleaza pe procesor si alt thread nu.

TLDR: daca privesti ideea ca procesul are mai multe thread-uri, atunci TEORETIC un singur proces e pe CPU, dar nu cred ca e asta scopul intrebarii

Ce este un proces I/O intensive?
Un proces care lucrează mult cu I/O-ul și in consecinta are multe apeluri blocante (trebuie de multe ori sa astepte pentru a primi și a scrie date). De cele mai multe ori un astfel de proces cand va ieși din starea RUNNING se va muta în BLOCKED.

Ce este un proces CPU intensive?
Un proces care face calcule intense, gen machine learning si care nu se blocheaza foarte des, pentru a iesi din starea RUNNING de obicei e nevoie sa fie scos nevoluntar. De cele mai multe ori un astfel de proces cand va iesi din starea RUNNING se va muta in READY.

Cum tratează planificatorul procesele I/O intensive și procesele CPU intensive?
In general este o idee buna pentru planificator sa acorde o cuanta mai mare proceselor I/O-intensive. De ce? Cuanta de timp nu se reseteaza atunci cand un proces intra in starea READY. Planificatorul isi doreste throughput si fairness. Doreste un overhead cat mai mic din partea lui, dar si un waiting time mic pentru procesele din READY. Nu isi permite sa dea o cuanta mare celor CPU intensive pentru ca asta ar mari waiting time-ul, insa isi permite sa dea la cele I/O

intensive pentru ca acestea vor iesi voluntar (se mentine watiting time mic). Nu am de ce sa ii dau o cunata mica si sa il scot nevoluntar de multe ori, cand el oricum va iesi voluntar. Daca tot e atat de treaba si iese singur voluntar (imi tine waiting time-ul mic) de ce sa il mai scot si eu nevoluntar si sa introduc overhead inutil/
(neresetarea cuantei e relevanta pentru ca daca s-ar reseta nu prea ar mai conta dimensiunea cuantei la procesele I/O intensive, ea resetandu-se de fiecare data la orice iesire voluntara)

In functie de ce algoritm de planificare folosesti, depinde si de ce cerinte indeplineste acel sistem, el nu poate fi productiv si totodata sa iti dea o experienta smooth.

Două thread-uri ale unui proces execută aceeași funcție. Care sunt diferențele între cele două thread-uri?
Au code pointeri diferiti si stive diferite, astfel ca variabilele locale din functie sunt specifice thread-ului.

Ce este un apel thread-safe? Ce este un apel reentrant?
O functie thread-safe garanteaza ca va fi un singur thread care executa functia intr-un moment de timp, pe cand o functie reentranta garanteaza ca va produce rezultatul asteptat oricare ar fi numarul de thread-uri care apeleaza acea functie. Astfel, orice functie reentranta este thread-safe, dar nu si invers.
Intr-o functie reentranta pot sa intre mai multe thread-uri, dar ele nu se calca pe picioare pentru ca folosesc fiecare un buffer propriu si nu buffer-ul global.
Un apel thread safe = un apel care poate fi apelat de mai multe threaduri in acelasi timp.
Functia reentranta e o functie care se executa si termina corect indiferent de cine e apelata
A reentrant subroutine can achieve thread-safety,[1] but being reentrant alone might not be sufficient to be thread-safe in all situations. Conversely, thread-safe code does not necessarily have to be reentrant https://en.wikipedia.org/wiki/Reentrancy_(computing)
Un exemplu de apel reentrant: bariera reentranta.

Reentrant = codul este incapsulat, nu depinde de variabile globale sau statice si nu cheama alte functii nereentrate, adica daca f(x) este reentrat si apelez f(1) mereu va avea acelasi rezultat, indiferent de cate ori “intru” in functie

Cum tratăm situația în care apelăm o funcție non-reentrantă într-un handler de semnal?
Sunt dezactivate intreruperile inaintea tratarii intreruperii, iar apoi sunt reactivate dupa tratarea acesteia. Scopul acestei solutii este de a evita un deadlock (in cazul in care acea functie non-reentranta este thread-safe, eg. malloc).

Memorie

Cum asigură sistemul de operare separația între procese?
Prin mecanismul de memorie virtuala. Fiecare proces are impresia controlului intregii memorii. Ii este imposibil astfel sa acceseze memoria altui proces.

Ce înseamnă mecanismul de memorie virtuală?
Mecanismul de memorie virtuală este folosit de către nucleul sistemului de operare pentru a implementa o politică eficientă de gestiune a memoriei.

Mecanismul de memorie virtuala este un mod al sistemului de operare de a imparti memoria fizica intre procese si in acelasi timp sa realizeze o izolare intre acestea (sa nu se bage un proces peste memoria altui proces).
Fiecare proces, in loc sa lucreze cu memorie fizica, lucreaza cu memorie virtuala (care nu exista). Astfel fiecare proces are impresia ca are la dispozitie toata memoria (ii este astfel imposibil sa acceseze memorie care nu e a sa, din moment ce toata e a sa). Pentru a ajunge totusi la memorie fizica, sistemul de operare aloca/mapeaza fiecarei pagini virtuale a unui proces (page) o pagina din memoria fizica (frame). In momentul cand un proces refera o zona de memorie, el o refera din memoria virtuala, dintr-un page. Page-ul apoi este translatat de MMU (memory management unit) in frame-ul corespunzator si procesul isi primeste datele dorite.

Ce reprezintă spațiul virtual de adrese al unui proces?
Intr-un sistem de operare care foloseste conceptul/mecanismul de memorie virtuala, fiecare proces lucreaza cu memorie virtuala, lucreaza cu adrese virtuale. Acest spatiu de adrese virtuale se numeste spatiul virtual de adrese al procesului.
El are corespondent in memoria fizica (nu mereu, daca ai on demand), vezi tabela de pagini.

Ce este paginarea memoriei?
Pentru a imparti memoria eficient, tinand cont ca diferite procese au nevoie de dimensiune de memorie diferita, procesele pot cere mai multa memorie la runtime si pentru ca permisiunile la diferite zone din aceasta memorie trebuie sa fie diferite se doreste alocarea de bucati, in loc de doar o zona mare. Pagina este o bucata de dimensiune fixa (de obicei 4kb). Paginarea memoriei inseamna impartirea memoriei in pagini. Paginarea e cea care genereaza fragmentarea interna.

Ce este fragmentarea internă a memoriei?
Fragmentarea interna se intampla la paginare si inseamna ca am gauri in ceva deja alocat. Tinand cont ca la paginare se aloca bucati de dimensiune fixa, orice alocare de memorie care nu este multiplu de acea dimensiune va crea fragmentare interna. De ex: pagina de dimensiune 4Kb, vreau sa aloc 5Kb, am nevoie de 2 pagini, asadar voi aloca 8Kb. Gaura de 3Kb este fragmentare interna.
Exemplu des intalnit: cand folosesti struct-uri si compilatorul nu ti le ordoneaza , trebuie sa le pui de la mare la mic ca sa n-ai probleme cu padding ul si sa ai foarte multe goluri intre elehttp://katecpp.github.io/struct-members-order/

Ce este fragmentarea externă a memoriei?
Fragmentarea externa se intampla la segmentare si inseamna ca am gauri in ceva ce nu e alocat. La segmentare ni se dau bucati de dimensiune variabila, in functiie de cat este nevoie. Cand una din aceste bucati este eliberata va aparea o gaura, iar daca nu este nevoie de exact aceeasi dimensiune inca o data, gaura va ramane acolo (sau va fi acoperita dar nu in totalitate, sau tot ramane gaura). Aceea gaura este fragmenare externa.

Ce rol are tabela de pagini?
Tabela de pagini are rolul de a stoca corespondenta pagini virtuale - pagini fizice. Este individuala fiecarui proces. Este folosita de MMU pentru a translata din pagini virtuale in pagini fizice.

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

MMU-ul este unitatea hardware care translateaza paginile virtuale in pagini fizice. MMU-ul notifica sistemul de operare de accese invalide (segfault-uri).
MMU e cel care genereaza o intrerupere care va fi capturata de catre SO si tratata.

Ce rol are TLB?

TLB (Translation Lookaside Buffer) este o memorie cache la nivelul sistemului care cache-uiește intrările din tabelele de pagini ale proceselor. Întrucât fiecare acces la memorie necesită de fapt două accese (unul la tabela de pagini, alta la datele efective), TLB-ul micșorează timpul de acces simplificând primul acces (la tabela de pagini). TLB îndeplinește astfel rolul eficientizării accesului la memorie. Hit rate bun -> timp de translatare mai mic. Are intre 128-256 intrari

Ce conține o intrare în tabela de pagini?
Numarul cadrului, permisiunile si daca pagina este valida
Indexul paginii virtuale, permisiunile, daca pagina e valida, bit dirty, (bit de nx eventual) si indexul paginii fizice
Bitul dirty se activeaza cand scrii in pagina daca nu e deja activat.
El se dezactiveaza, fie periodic, fie atunci cand e swapata.
El este folosit la swap-out, paginile cu bit dirty 0 (adica nemodificate de ceva timp / nefolosite) sunt primele candidate alese pentru swap-out.
Bitii de available programabili

Ce înseamnă tabelă de pagini multi-nivel (ierarhică)? De ce este utilă?
Utila pentru ca e mai rapid, dar ocupa mai multa memorie //nu cred, e fix invers, e mai incet dar ocupa mai putina memorie +35
Se pune problema cat ocupa o tabela de pagini (cata memorie fizica ram ocupa). Pentru adrese de 32 de biti si pagini de 4Kb (2^12), tabela de pagini are 2^32 / 2^12 = 2^20 de intrari. Daca o astfel de intrare ocupa 4 octeti, spatiul ocupat de tabela ar fi de 4Mb (sunt megabytes nu megabiti). Pentru adrese de 64 de biti avem 128Tb/4Kb = 32G intrari * 4 = 128Gb memorie ram ocupata de fiecare proces in parte pentru tabela sa de pagini. Evident acest lucru nu este fezabil, in special pe sisteme de 64 de biti. Pentru a micsora dimensiunea tabelei de pagini a aparut tabela de pagini multi-nivel. Tabela de pagini multi-nivel imparte indexul paginii in mai multi indecsi. Astfel primul index va referi un tabel, din acel tabel scoatem base register-ul urmatorului tabel de unde indexam cu al doilea index si tot asa. Avantajul provine din faptul ca bucati mari din ierarhie nu vor trebui alocate fizic. Daca de exemplu in primul tabel avem doar 20% din intrari valide, deja am scapat de 80% din ierarhie. Dupa, urmatorul tabel poate are si el doar 40% din intrari valide, am mai scapat de 60% din 20% si tot asa.

$Tabela de pagini ierarhica are 2^10 intrari. Fiecare intrare refera o structura de tip tabela de pagini. Aceasta contine 2^10 adrese de pagini fizice. Daca o zona lipseste, intrarea in page directory nu refera o tabela de pagini. Prin urmare, spatiul este redus, dar apare mai mult overhead la translatare.

Este mai rapida si ocupa si mai putina memorie (a doua este valabila 100%), daca folosesti o parte din tabela foarte des si pe restul rar, ii va lua mult mai putin sa le gaseasca pentru ca nu trebuie sa parcurga toata tabela.
Când are loc un TLB miss?
Cand pagina pe care vreau sa o accesez nu este in TLB

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

Pentru ca procesele au spatiu separat de memorie si nu au aceleasi pagini de memorie. Exista un singur TLB, trebuie folosit pentru procesul care ruleaza. La context switch, cand se schimba procesul, TLB-ul trebuie populat cu intrarile din tabela de pagini ale noului proces.

De ce nu este nevoie de TLB flush la schimbarea de thread-uri între două procese?
Pentru ca refera aceeasi memorie
Se face flush la TLB. Este nevoie pentru ca diferite procese au spatiu de adrese diferit. <- de acord; cred ca vrea sa spuna ca ai 2 threaduri care tin de 2 proc diferite si te rezumi la a schimba procesele intre ele

Ce înseamnă mecanismul de copy-on-write?
Cand se aloca paginile pentru procese ca readonly si in momentul in care un proces vrea sa scrie, se face o copie a paginii si se fac modificarile in noua pagina
De exemplu la fork pe linux spatiul de adrese este duplicat din parinte in copil prin mecanismul copy-on-write. Este o metoda de a salva memorie fizica pana cand se fac scrieri (nu am de ce sa duplic o pagina fizica a parintelui si sa o dau copilului daca acestia fac doar read). Procesului nou creat i se va crea tabela sa proprie de pagini. Intrarile din aceasta tabela vor referi aceleasi pagini fizice pe care le refera si intrarile din tabela de pagini ale parintelui. Toate intrarile rw din tabela de pagini si din parinte si din copil, sunt marcate ca read-only, iar la nivelul sistemului de operare sunt marcate cow (copy-on-write). Cand unul din procese face write intr-o pagina, MMU-ul va genera page-fault (nu exista permisiuni de scriere). In handler-ul sistemului de operare pentru page-fault acesta va vedea ca pagina este marcata cow si va duplica pagina fizica in cauza, va scrie in ea ce a scris procesul si va updata in tabela de pagini a procesului respectiv pentru pagina ceruta perimisiuni (din r in rw) si indexul paginii fizice (din aia veche la asta nou alocata).

Ce înseamnă demand paging?
Atunci când sistemul de operare folosește demand paging alocarea de memorie fizică este amânată până în momentul în care nevoie (adică la primul acces). Sistemul de operare doar rezervă memorie virtuală și nu alocă memorie fizică în spate, economisind memorie fizică. La primul acces se alocă și memorie fizică, la cerere (adică on demand) și se face maparea acesteia la spațiul virtual (paging).

Atunci când se rezervă prima oară o pagină virtuală (la load time sau la run time):
- se marchează pagina ca **nevalidă** în **tabela de pagini**;
- se marchează ca "**validă-nemapată**" în informațiile **SO**.

La primul acces, MMU generează excepție (intrarea în tabela de pagini este nevalidă). Excepție e capturată de SO, SO alocă o pagină fizică, o completează în tabela de pagini și marchează intrarea validă (și în informațiile sale interne).

Excepția este numită **page fault** (excepție de acces la pagină).

În ce situație apare page fault fără a cauza segmentation fault?
În cazul demand paging, se alocă o pagină virtuală (page) fără suport fizic (frame). La apariția unui page fault se va aloca o pagină fizică, fără a rezulta o excepție.
În cazul copy-on-write, două pagini virtuale (page) (din două procese diferite) sunt marcate read-only și referă aceeași pagină fizică (frame). În momentul în care unul dintre cele două

procese efectuează o operație de scriere, se obține page fault, se duplică pagina fizică și se continuă execuția.

În cazul unei pagini virtuale (page) valide, al cărei conținut se găsește în swap, un acces generează page fault. Conținutul este swapped in într-o pagină fizică (frame) si procesul își continuă execuția.

Ce rol are spațiul de swap?

Spațiul de swap este spațiul localizat pe disc folosit ca depozitar temporar al informațiilor din memorie RAM. În momentul în care spațiul fizic (memoria RAM) devine insuficient, se evacuează (swap out) anumite pagini fizice. În momentul în care aceste pagini sunt necesare sunt readuse în memoria RAM (swap in).

Când are loc swap in și swap out?

Swap in = avem nevoie de pagina respectiva de memorie si este adusa in ram
Swap out = nu avem suficienta memorie si evacuam pagina din ram pe swap

Care este rolul unui page fault. În ce condiții apare?

Fault minor atunci cand este pagina in memorie, dar nu este mapata.
Fault major atunci cand pagina este in swap.
Cazuri: pagina e nemapata(swap, on demand paging)
Pagina nu e alocata -> segfault
Permisiuni (copy-on-write, readonly)
Cand intrarea din tabela de pagini asociata fie e nevalida, fie nu are permisiuni. Rolul sau este de a notifica sistemul de operare. La page-fault se apeleaza un handler (o rutina) din sistemul de operare.

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

.text .rodata .data .bss, heap, zona unde se mapeaza bibliotecile dinamice, stiva, zona kernel

- **text** - cod, executabil
- **rodata** - constante
- **data** - variabile statice inițializate
- **bss** - variabile statice neinițializate (setate la 0)
- **heap** - zona pentru alocari dinamice
- **biblioteci** (cu subzone: text, rodata, data, bss)
- **stivă** - variabile locale

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

.data .bss (cred ca si heap-ul si stiva sunt writable, fiind rw) da, sunt si heap si stiva

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

Pentru ca vor partaja zone de cod din cadrul bibliotecii dinamice cu alte procese. Ele sunt incarcate o singura data in RAM si apoi mapate in spatiul virtual al fiecarui proces care le foloseste. -fPIC => position independent code

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

Vor partaja bibliotecile dinamice si zona unde este mapat sistemul de operare (kernel space)
Partajeaza si zona de cod si rodata

Se alocă un buffer a[100]. De ce a[105] NU va rezulta, în general, în Segmentation fault? În ce situație a[300] rezultă în Segmentation fault?
a[105] nu va rezulta in SegFault deoarece se afla (in general) in aceeasi pagina cu a.
Accesul la a[300] va rezulta in SegFault in momentul in care adresa (a + 300 * sizeof(a)) se afla intr-o pagina nerezervata/fara permisiunile necesare.

Ce înseamnă maparea unui fișier în memorie? De ce este avantajos să mapăm fișiere față de folosirea read/write?
In loc sa deschizi o structura de fisier si sa scrii in ea, scrii direct la adresa/blocul unde stii ca se afla fisierul. Este considerabil mai rapid pentru ca e practic acces de tipul write la memorie.
Este mai rapida pentru ca ai Memoria ram care asigura ca datele sunt citite foarte rpd + daca vrei sa te muti dintr-o parte in alta, sa stergi, sa citesti etc doar te joci cu pointerii Singurul dezavantaj e ca trebuie sa stii dimensiunea exacta.
De obicei mapam in memorie executabile cu dimensiune cunoscuta. Se reduce overhead-ul datorat apelurilor de sistem inexistente.
Pentru a simplifica lucrul cu fișierele, acestea pot fi mapate în spațiul de adresă al unui proces. Adică scrierea într-o pagină virtuală conduce la scrierea în blocul corespunzător de pe disc al procesului.

Are loc uzual pentru fișiere **executabile** și **biblioteci partajate**: sunt mapate în spațiul virtual al proceselor.

Operațiile cu fișierele sunt acum operații cu memoria, nu mai sunt apeluri de sistem read/write.

Avantaje:
- overhead scăzut temporal - nu se fac apeluri de sistem
- și spațial - nu se alocă buffere în user-space pentru apelurile read/write

Dezavantaje: fișierele trebuie să aibă dimensiunea știută pentru mapare, nu se poate crește dimensiunea (cum se întâmplă atunci când folosim `write()` pentru a scrie dincolo de dimensiunea fișierului).

Câte page fault-uri se pot obține în cazul operației *a = b?
Minim 0, pagina fizica necesara pt a era deja alocata si mapata, maxim 1 => pagina fizica pt a nu era mapata.
[aici nu sunt chiar sigura daca poate sa verifice cineva si sa completeze/corecteze]
Ma gandesc si la permisiuni: daca la adresa a nu se poate scrie (nu sunt permisiuni de rw ca sa se modifice) sau pt b nu sunt permisiuni de citire (adica nicio permisiune, mai rar)
1 pt pointer, 1 pt a si 1 pt b=>3

Ce informații sunt reținute în stivă? Ce variabile C?
Variabilele locale functiei, zona locals in momentul apelarii unei functii

În ce zonă sunt reținute variabilele globale inițializate și cele neinițializate?
Initializate .data, neinitializate .bss

Ce înseamnă operația de striping a unui executabil?
Stergere de simboluri de debugging

Ce se întâmplă la faza de loading (încărcarea unui executabil în memorie și crearea unui proces)?
Loader-ul încarcă secțiunile de cod și date din executabil și pregătește stiva pentru proces (registrul de stivă) și referă registrul de instrucțiuni la entry point-ul procesului, indicat în header-ul executabilului.
După ce pregătește stiva, loader-ul încarcă la începutul ei variabilele de mediu și argumentele programului.
Executabilele și bibliotecile sunt mapate în spațiul virtual de adrese ale procesului.

Ce este entry point-ul într-un executabil?
Adresa primei instructiuni. **Entry point-ul executabilului** (adresa primei instrucțiuni ce va fi executate) se găsește în header-ul executabilului. În general referă un simbol cu numele `_start / start` care face niște acțiuni pregătitoare și apoi apelează funcția `main`.
Cand loader-ul il va incarca in memorie, va seta instruction pointer-ul la valoarea din entry point, bibliotecile dinamice au ,de asemenea, un entrypoint.

Ce utilitare cunoașteți pentru analiză dinamică și ce utilitare cunoașteți pentru analiză statică?
dinamice - gdb, strace, ltrace, ptrace
statice - readelf, objdump, strings

Ce înseamnă analiză statică și ce înseamnă analiză dinamică?
Analiza statica presupune analizarea executabilului si a codului, iar analiza dinamica presupune rularea executabilului si analizarea comportamentului, memorie, registre, apeluri de sistem la runtime.
Analiza statică este analiza pe program fără ca acestea să ruleze. Poate fi analiză statică pe cod sursă sau analiză pe binar / executabil.
Analiza dinamică are loc în momentul rulării programului în proces și are ca țintă principală procesul și resursele folosite de acesta: memorie, registre, fișiere, apeluri de sistem, fluxul de execuție al programului.

Acțiuni specifice analizei statice pe executabile sunt:
- dezasamblarea codului
- identificarea simbolurilor (variabile, funcții): adrese, valori, dimensiuni
- identificarea șirurilor
- construirea grafului de apel al programului (*call graph*)
- identificarea bibliotecilor externe necesare
- identificarea simbolurilor exportate și a simbolurilor importate din surse externe

Acțiunile specifice analizei statice se pot realiza și în analiză dinamică. Pe lângă aceasta, se poate urmări fluxul de execuție al unui program, însemnând că putem:
- realiza **breakpoint-uri** și **stepping**
- inspecta **memoria** și **registrele** programului
- inspecta spațiul de memorie folosit, resursele folosite
- urmări fluxul de execuție al programului: **tracing**
- realiza instrumentarea codului care să permită urmărire sau analiză: model aplicat de `Valgrind`, AddressSanitizer, profilere

Utilitarul esențial de analiză dinamică este depanatorul / **debuggerul** .

Nu prea e debugger-ul in realitate, 99% din cod-ul din productie e ofuscat (abia astepti sa vezi cod asm ofuscat nu ? :))) si rasucit pe toate partile, in general vrei sa vezi ce apeluri de sistem face ca sa-ti dai seama ce se intampla cu adevarat, pt asta, pe windows poti folosi suita Sysinternals, care contine procmon.

Ce înseamnă Stack Guard / Stack Smashing Protection (SSP)?
Este o metoda de a preveni atacurile de forma buffer overflow. Se plaseaza o valoare, numita stack canary (diferita la fiecare rulare) intre buffer si adresa de retur a functiei. Astfel, inainte de terminarea functiei se verifica (__stack_chk_fail) daca acea valoare a ramas aceeasi. Daca nu a ramas aceeasi, se termina fortat procesul (stack smashing detected).

Stack Smashing Protection (SSP) sau **stack canary**: se plasează o valoare între buffer și adresa de retur.

Suprascrierea adresei de retur prin buffer overflow va însemna suprascrierea valorii canar, lucru ce va fi detectat la părăsirea funcției. Canarul este plasat într-o zonă dedicată. Se pot suprascrie în continuare variabile locale.

Dezavantaj: mic cost de performanță => se poate aplica SSP selectiv pe funcțiile ce conțin pointeri.

bypass: se suprascrie canarul cu el însuși. Se plasează uzual 0x00 și 0x0a în canar pentru a *opri* funcții de lucru cu șiruri din suprascriere.

bypass: se suprascrie handle-ul de tratare a suprascrierii canarului

E greu sa faci bypass-urile mentionate de voi, cea mai simpla metoda, dar putin fezabila e sa-l bagi in debugger, pentru ca ori valoarea comparata,ori chiar a SC va fi pusa in registrii, iar in debugger poti modifica registrii cum vrei. In general se face bruteforce pe adresa respectiva, sau daca ai noroc sa suprascrii rutina de la final . Stacksmashing ul e adaugat de compilator, sau de cele mai multe ori, e implementat de programatori, acum tu trebuie sa-ti dai seama de flow si sa vezi cam cum l-ai putea pacali. Ce metode se gasesc pe internet dau 99% fail, in securitate nu exista 2 situatii identice, deci nimeni nu are cum sa prezica o astfel de operatie.

Ce efect are ASLR (Address Space Layout Randomization)?
Maparea functiilor(nu a sectiunilor?) in memorie se face la adrese random la fiecare rulare, astfel incat sa faca dificila descoperirea adreselor (in cazul atacurilor de tip code reuse). Executabilul trebuie sa fie PIE.

Ce efect are PIE (Position Independent Executable)?
Un binar PIE si toate dependentele sunt incarcate la adrese random din spatiul virtual de memorie la fiecare executie a aplicatiei, ceea ce face dificila(dar nu imposibila) tentativa unui atac de tip Return Oriented Programming(ROP).

Ce înseamnă deturnarea fluxului de execuție a unui program (control flow hijack)? De ce este acest lucru relevant pentru un atacator?
Inseamna ca alterez fluxul normal al executiei, fie prin suprascrierea unor code pointeri (in loc sa se execute codul care trebuie, se executa altceva), fie suprascrierea datelor in vederea trecerii anumitor conditii (de exemplu un if).
Adaugarea de arce/noduri in control flow graph (CFG)

Ce înseamnă memory leak / memory disclosure? De ce este acest lucru relevant pentru un atacator?
Inseamna ca cineva care nu ar trebui sa aiba acces la niste date/adrese are acces la ele. Este util pentru atacator pentru a afla date personale sau alte portiuni de memorie de interes prin care poate ajunge la alte date/alte adrese sau pe care le poate suprascrie in vederea denaturarii fluxului.

Ce înseamnă că o secvență de cod este PIC (Position Independent Code)?
Inseamna ca bucata de cod, la fiecare rulare va avea alte adrese. La fiecare rulare se va incarca altundeva codul, la alta adresa.
La compilare se genereaza sectiunile, dar nu stim exact unde se vor incarca si ca sa le gasim avem nevoie de 2 tabele, GOT si PLT.

Adica codul nu mai face referire la adresa absoluta, totul este bazat pe adrese relative.

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?
In principal pentru ca fiecare apel de sistem ar insemna schimbarea contextului, ceea ce ar face flush la TLB si ar fi un overhead temporal foarte mare. Distinctie user privilegiat si nonprivilegiat. Pentru ca zona kernelului este comuna intre procese. Cand se face context switch nu va fi nevoie sa se schimbe intrarile din page table asociate cu zona pentru kernel.

Ce este un code pointer? De ce este interesant din perspectiva securității memoriei?
Un code pointer este un pointer care pointeaza o bucata de cod (de ex return address-ul sau function pointers). Pentru atacatori este o modalitate de a altera fluxul programului (pot face code reuse sau code injection). Din punct de vedere defensiv trebuie sa ii protejam cat mai bine.

Ce este un shellcode?
Este o secvente de cod INJECTAT care contine linia exec(”/bin/sh”), asadar deschide un nou shell prin care un atacator poate sa faca tot felul de nebunii: deschida socketi, schimbe permisiuni etc.

O secvență de cod mașină injectată pentru a fi executată: code injection. Uzual este combinată cu exploatarea unui buffer overflow și suprascrierea unui code pointer pentru a ajunge la acea zonă.

code injection = numele atacului; **shellcode** = codul folosit în atac.

Pentru a fi executată zona trebuie să fie:
- read-write (să poată fi scris shellcode-ul)
- executabilă (să poată fi executat)

Convențional un shellcode deschide un shell (`exec("/bin/sh")`), dar poate fi folosit la orice: deschis un socket, schimbat permisiuni, citit un fișier.

Un **shellcode** conține **cod mașină** și folosește **apeluri de sistem**, nu apeluri de funcții de bibliotecă; nu ar ști unde este plasat în memorie și unde se găsesc adresele funcțiilor.

Ce înseamnă code reuse din perspectiva securității memoriei?

Code reuse este atunci cand un atacator redirectioneaza fluxul programului catre o alta zona executabila din spatiul de adrese al procesului fata de cea inițială.
Se adauga arce in control flow graph (CFG), se foloseste cod existent in program in scop malițios

Ce înseamnă shell injection din perspectiva securității memoriei?
Cand un atacator ruleaza un shell in procesul tau, dar e foarte complicat. El trebuie sa sara la o zona executabila unde sa apeleze mprotect ca dupa sa isi scrie cod propriu si sa il poate executa.

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

```
int *a = NULL;
int b = *a;
```

A doua instructiune da sigsegv daca nu am alocat/rezervat memoria de la adresa a. NULL e la adresa 0 care nu e alocata/rezervata.

Cu ce diferă o funcție de o variabilă într-un executabil și/sau în cadrul spațiului de adrese al unui proces?
Functia conteaza in contextul fluxului de executie al programului, daca ii suprascriem adresa de retur vom putea altera fluxul programului.
O variabila, mai ales una deja intializata, chiar daca ii putem schimba si ei valoarea va fi mai greu sa alteram fluxul cu ajutorul ei. Sa zicem ca ar fi un if care depinde de valoarea acelei variabile dar asta e mult mai greu de depistat.
Zona unde e functia are permisiuni rx, variabila are permisiuni rw

Două procese partajează o zonă de memorie. Cum se manifestă acest lucru în tabelele de de pagini ale celor două procese?
Cadrul apare in zona de memorie al fiecarui proces
Ambele procese vor avea mapata acceasi memorie fizica la memoria virtuala. Intrari din ambele tabele de pagini ale celor 2 procese vor pointa catre aceleasi pagini fizice.

Putem avea mai multă memorie fizică decât dimensiunea maximă a spațiului virtual de adrese al unui proces? Dar invers?
Nu putem avea mai multa fizica decat max size-ul spatiului virtual de adrese, insa invers se poate.
Exemplu cand avem mai multa memorie virtuala decat fizica:
1) biblioteci partajate intre mai multe procese sau ceva de genul, s-a spus la simulare
2) demand paging, este alocata doar memoria virtuala pana la primul acces, cand se aloca o pagina fizica
3) copy on write, dupa un fork, un copil va partaja cu parintele sau zonele pe care le avusese parintele rw, iar acum sunt read-only. Cand unul dintre ei acceseaza cu scopul de a scrie in pagina respectiva, se face un page fault, se aloca o alta pagina pentru acel proces si apoi se reincearca scrierea la adresa din pagina respectiva (nu stiu daca 3 se incadreaza ca raspuns pentru intrebarea data).

Fișiere, I/O

Ce conține un FCB (File Control Block)?

FCB-ul este o structura interna a sistemului de fisiere in care este pastrat state-ul unui open file. FCB-ul e manage-uit de kernel si rezida in memoria programului care utilizeaza fisierul, nu in memoria SO-ului.
FCB-ul este inodul, el nu are informatii despre fisierele deschise, el are metadatele fisierului. (tipul fisierului, indexul sau (id-ul), permisiuni, nr de link-uri, dimensiunea sa si pointeri catre datele sale, numele nu)
La primul open este adus inode-ul de pe disk in memorie, iar strucutra de open file (aia la care pointeaza file descriptorul) va avea un pointer catre acest inode.

Ce reprezintă un descriptor de fișier?

Este un număr (întreg) ce referă o intrare în tabela de descriptori de fișier. Este folosit în operații de lucru cu fișiere, pentru a identifica un fișier deschis.

Ce reprezintă tabele de descriptori de fișiere?

Tabela de descriptori de fișier a unui proces conține pointeri; ca structură de date este un vector de pointeri. Acești pointeri referă structuri de fișier deschis de proces. Când un proces deschide un fișier, se alocă o structură de fișier deschis, iar adresa acestei structuri este stocată într-un loc liber (indicat de descriptorul de fișier) din tabela de descriptori de fișier.

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

Fiecare proces are o tabelă de descriptori de fișier, deci vor exista, la nivelul sistemului de operare, atâtea tabele de descriptori de fișier câte procese există în acel moment în sistem.

Ce efect are apelul dup()?

Apelul dup() este folosit practic pentru redirectarea ieșirii, intrării sau erorii standard în fișier. Altă situație practică este pentru operatorul | (pipe) de comunicare între procese. //asta seamana mai mult a dup2(), unde specifici ce fd redirectezi si unde. La dup() specifici doar unde vrei sa redirectezi, si se alege primul fd liber din FDT care se redirecteaza acolo
dup() duplica un file descriptor pe prima pozitie available din FDT. Noul descriptor va pointa catre aceeasi structura de fisier deschis ca si descriptorul pe care l-a duplicat.

Ce efect are apelul close()?

Kernel-ul elibereaza file descriptorul asociat.Also resursele asociate structurii open file sunt freed.
Structura de fisier deschis are un camp care specifica cati file descriptori pointeaza catre ea. La close se sterge intararea din FDT, se decrementeaza campul respectiv din structura de open file si daca campul devine 0 (nu mai pointeza nimeni catre ea) ea este eliberata.

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

1. lseek/fseek, apeluri al căror rol este de modificare a cursorului de fișier;
2. read/fread/fgets – la fiecare citire cursorul de fișier este incrementat cu numărul de octeți citiți;
3. write/fwrite/fputs/fprints – la fiecare scriere cursorul de fișier este incrementat cu numărul de octeți scriși;
4. ftruncate – trunchiază fișierul (cursorul este plasat pe 0);
5. apelurile echivalente Windows.

Ce se intampla cu cursorul de fisier daca doua procese deschid acelasi fisier?

Daca procesele folosesc acelasi file descriptor si pointeaza la acelasi open file, cursorul de fisier este acelasi, altfel daca au fd diferiti si pointeaza la cate un open file structure fiecare,

au cate un cursor fiecare si in al doilea caz se face overwrite la datele scrise de catre primul proces, in cazul in care si al doilea apeleaza write(). Ideal ar fi sa evitam situatiile in care intr-un fisier scrie mai mult de un proces, fiindca ordinea in care procesele intra in RUNNING este imprevizibila, fiind gestionata de task scheduler.

Two different file descriptors that refer to the same open file description share a file offset value. Therefore, if the file offset is changed via one file descriptor (as a consequence of calls to read(), write(), or lseek()), this change is visible through the other file descriptor. This applies both when the two file descriptors belong to the same process and when they belong to different processes.

Ce apeluri modifică dimensiunea fișierului?

Apeluri care pot modifica dimensiunea unui fișier sunt write (poate scrie dincolo de limita unui fișier), ftruncate (modifică chiar câmpul dimensiune) sau open cu argumentulO_TRUNC care reduce dimensiunea fișierului la 0

Ce efect are apelul/comanda truncate?

Trunchiaza fisierul la o dimensiune specificata. Fisierul poate fi trunchiat la 0 si prin apelul open() cu O_TRUNC.

Ce este un hard link?

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

În general, un FCB **nu** conține numele unui fișier. Numele unui fișier este reținut într-o structură separată numită **dentry** (directory entry).

Un dentry conține:

- referință către inode (inode number)
- numele fișierului.

Un **dentry** se mai numește un **link** (sau un **hard link**). Dentry-urile sunt reținute în blocurile de date ale directoarelor.

Pot exista mai multe dentry-uri care referă același inode, adică mai multe hard link-uri, util pentru a plasa un fișier în mai multe locuri.

Un **FCB** conține și numărul de link-uri. Un fișier este șters când nu mai are link-uri.

Comanda `rm`, apelul `remove()` **nu** șterg un fișier ci șterg un link (un dentry). Apelul de sistem aferent este `unlink()/unlinkat()`.

Ce este un link simbolic/symlink?

Un symlink este un tip de fisier, ii este asociat un inode ca si la celelate tipuri de fisier. In datele sale are un string (cale catre fisierul pe care il refera)

Symbolic link-urile (*symlink*) sunt FCB-uri al căror conținut este o cale, un șir. Șirul este interpretat și rezultă un dentry și FCB-ul corespunzător.

Un symlink poate fi *dangling* dacă șirul nu este o cale validă (nu se rezolvă la dentry și FCB).

Comanda readlink rezolvă șirul de tip cale a unui FCB de tipul symlink.

Un **symlink** este un **inode**(**FCB**), un **hard link** este un **dentry**.

Putem crea symlink-uri la intrări din alte partiții (sisteme de fișiere montate), dar nu hard link-uri. Două sisteme de fișiere diferite au fișiere diferite cu același inode, nu ar funcționa un hard link între sisteme de fișiere, nu ar face diferența.

Care este diferența dintre un link simbolic și un hard link?

Un symbolic link are un inode al său, pe când un hard link este un dentry (un nume și un index de inode). Un symbolic link poate referi directoare în timp ce un hard link nu; un symbolic link poate referi un fișier de pe altă partiție/alt sistem de fișiere, în timp ce un hard link nu.

De ce numele unui fișier nu se găsește în inode?

Nu-l gasim deoarece numele sunt stocate in director si fiecare e folosit drept index in tabela de inodes. Mecanismul asta face viata mai usoara atunci cand vrem sa redenumim/mutam un fisier fiindca in felul asta se schimba doar maparea inodurilor in cadrul directoarelor.

Ce se întâmplă în cazul formatării unei partiții?

Se creeaza un nou file system layout in aceea partitie.

DIN CURS

Partitia - impartita in 2, o parte stocheaza date si o parte metadate. Partea de date e cam de 3 ori mai mare. Cand se face formatarea diskului practic se reinitializeaza metadatele

un sistem de fișiere se află pe o partiție, în urma operației de formatare
formatarea creează superblocul și celelalte zone, creează inode-ul rădăcină
sistemul de fișiere cuprinde un superbloc: acesta conține informații despre celelalte secțiuni (metadate despre metadate)
în mod uzual există o zonă care reține informații despre inode-urile valide/activate/ocupate (inode map) și o zonă care reține informații despre blocurile valide/activate/ocupate (data map); aceste zone sunt uzual bitmap-uri: bit 0 înseamnă inode sau bloc liber, bit 1 înseamnă inode sau bloc ocupat
o zonă dedicată reține inode-urile și altă zonă reține blocurile

Ce se întâmplă cu sistemul de fișiere în cazul folosirii cu succes a comenzii rm?

Folosirea comenzii rm cu success presupune stergerea referintei respectivului obiect din sistemul de fisiere ca o consecinta a unui apel de sistem -> unlink.
Se face unlink, se sterge dentry-ul din datele directorului, se decrementeaza nr de hard linkuri pe care le are inode-ul. Daca nr de hard link-uri devine 0 inode-ul este sters.
Rm sterge un hard link

Care este un avantaj al folosirii hard link-urilor și un avantaj al folosirii link-urilor simbolice?

Hard link: putem folosi link-ul si dupa ce fisierul original a fost sters sau mutat.
Symlink: putem sa referim fisiere de pe alte partitii.

Ce efect are comanda mv /path/to/a.dat /new/path/to/b.dat în sistemul de fișiere?
Se creeaza un nou hard link in noua cale si dupa se face rm (unlink) la primul fisier.

Care sunt tipurile de fișiere pe un sistem de fișiere uzual Unix?
Regular, directory, symbolic link, FIFO special, block special, character special si socket.

Care tipuri de fișiere nu au blocuri de date?
Numai REG, DIR si LNK au date. FIFO, BLK, CHR, SOCKET

Ce conțin blocurile de date ale unui director?
Conțin un vector de dentry-uri. Un dentry este o structură ce conține numele fișierului și indexul inode-ului aferent. Fiecare intrare din director (indiferent de tipul acesteia: fișier, director, link symbolic) are un dentry.
Contine minim o intrare pt . (autoreferinta) si una pt .. (directorul parinte)

Ce este un sistem de fișiere virtual?
As far as Linux is concerned:
Avem mai multe tipuri de sisteme de fisiere. Exemple: ext, ext2...ext4, XFS, JFS, btrfs etc. Fiecare vine cu implementari diferite(Exemplu: modul in care block-urile de fisier sunt alocate sau felul in care directoarele sunt organizate), ceea ce conduce automat la diferente in modul de abordare a lucrului cu fisiere in cadrul unui sistem de operare. Un sistem de fisiere virtual elimina acest inconvenient, prin adaugarea unui **layer abstract pentru operatiile sistem-fisier**.

Ideile de baza:

- VFS(virtual file sistem) defineste o interfata generica pentru operatiile sistem-fisier. Toate programele care lucreaza cu fisiere folosesc operatiile puse la dispozitie de aceasta interfata generica
- Fiecare sistem de fisiere are cate o implementare pentru interfata VFS

Interfata VFS include operatii corespunzatoare pentru toate apelurile de sistem uzuale pentru lucrul cu sistemele de fisiere / subdirectoare precum: open(), read(), write(), lnameseek(), close(), mmap() etc.

Note: In cadrul WIndows, din VFS lipsesc unele operatii pe care le avem disponibile pe un sistem de fisiere traditional UNIX. (Microsoft = scumbags)

Ce este un dispozitiv virtual?
Un dispozitiv virtual pointeaza catre un device file care nu are un hardware asociat. In cadrul unui OS precum Linux sau Unix, putem creea un dispozitiv virtual folosind comanda ‘mknod’ si acesta are aceleasi caracteristici precum un dispozitiv real. Kernel-ul il identifica drept un periferic, insa acesta este doar un fisier/director.

Ce tipuri de dispozitive cunoașteți? Clasificați-le din orice punct de vedere cunoașteți
Dpdv UNIX putem distinge doua tipuri de device-uri:
1.Dispozitive caracter(aka serial devices): se ocupa cu gestionarea datelor ‘one character at a time’.

Exemple de dispozitive caracter sunt terminalele, tastaturile, mouse-uri, placi de sunet etc. Datele pot fi scrise sau citite o singura data.

In situatia in care un byte e citit de la tastatura/mouse, acelasi byte nu poate fi citit de alt program.

2.Dispozitive block(aka random access devices): sunt responsabile cu gestionarea datelor 'one block at a time'. Dimensiunea unui block este in general un multiplu de 512 exprimat in bytes(poate varia in functie de tipul dispozitivului, in general avem 1024) si avem drept exemplu diverse medii de stocare precum disk-uri sau casete(d-alea cu banda magnetica).

Aici datele sunt persistente si exprimate in sectiuni contigue si spre deosebire de dispozitivele de tip caracter, putem citi/scrie datele de mai multe ori.

Exemple de device name-uri:
/dev/hda, /dev/hdb, /dev/mouse, /dev/modem, /dev/zero. /dev/null, /dev/tty(consola)
Note: Toate dispozitivele sunt reprezentate sub forma de fisier(in Unix si Linux).

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

Dispozitiv tic caracter: unseekable(explicatie mai jos), n-are nevoie de buffering, accesul in cadrul acestui tip de device se face sub forma de stream 'one byte at a time', driverul asociat este mult mai simplu, necesita mai putin efort fiindca are de gestionat o singura pozitie(cea curenta). Apelurile read(), write() sunt blocante.

Dispozitiv tip block: E accesat printr-un cache, deci avem nevoie de un buffer, driver-ul asociat kernel-ului necesita mai multa atentie/efort avand responsabilitatea de a naviga printre chunks of blocks in cadrul mediului de stocare. Apelurile bread(), bwrite() pot fi asincrone.

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

Operatia de seek n-are sens pentru un dispozitiv de tip caracter deoarece tipul asta de device are o singura pozitie, cea curenta(fiindca datele vin sub forma de stream 'one character at a time'), asadar nu putem sa miscam un pointer in fata sau inapoi in fisier.

'sudo cat /dev/input/mice' you'll see why

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

Functia returneaza file descriptor-ul socket-ului nou creat, care contine adresa/port-ul asociat peer-ului (aka aplicatia careia ii permitem sa se conecteze)

Adr IP locala + ce port gaseste SO-ul disponibil primul

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

Un apel de sistem read() sau write() intoarce numarul de bytes cititi/scrisi sau 0 in cazul end-of-file la read().

Ce operații se pot face pe fișiere? De ce avem două buffere?

Operatiile care sunt definite in structura de file_operations din cadrul kernelului, din care se pot aminti cateva mai populare: read(), llseek(), poll(), ioctl(), mmap(), write() etc.

- open,
- close,
- read,
- write,

- ioctl.

Avem user space buffer si kernel buffer cu rolul de a limita numarul de accesari directe pe fisierele de pe disc.

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

Nu modifica cursorul: operatiile de schimbare ale atributelor unui fisier, operatiile de open/close/create.

Modifica cursorul: operatiile de pozitionare intr-un fisier(obviously), scrierea/citirea si trunchierea.

Trunchierea nu modifica valoarea cursorului! +1 +2

Ce operații asupra fișierelor modifică/nu modifică dimensiunea fișierului?

Modifica: write daca scrii peste dimensiunea curenta, truncate

Nu modifica: lseek, read

Read, pozitionare, inchidere/deschidere, ?schimbarea de atribute ale fisierului?

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

Dimensiunea fisierului e retinuta in structura inod-ului(pe langa owner id, group id, flag-uri de sistem/user, device id etc), iar file pointer-ul rezida in cadrul kernel-ului, acesta avand rolul de I/O managing in File Descriptor Table. (le gasesti de obicei la /proc/__PID__/fd)

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

Un buffer e de send, altul de receive.

Ce este o întrerupere? Când este livrată o întrerupere?

Basically e un semnal/notificare catre un proces/kernel ca un eveniment exceptional a avut loc.

Sunt de doua tipuri: hardware si software. Cele hardware atentioneaza CPU-ul ca un device extern necesita atentie din partea sistemului de operare, iar cele software sunt generate de catre kernel si au drept identificator un nr intreg cu nume simbolice de forma SIGxxxx.

Cu ce diferă port-mapped I/O de memory-mapped I/O?

Cele doua metode de mapare sunt complementare in cadrul I/O. MMIO e mapat in acelasi spatiu de adrese ca programul in sine si e accesat in acelasi mod, iar PMIO are un spatiu de adrese separat si pentru acces se foloseste un set dedicat de instructiuni.

Ce este o operație asincronă?

Operatiile asincrone au drept caracteristica faptul ca pot fi executate in alta ordine fata de cea in care apar in cod.

Ce este o operație neblocantă?

O operatie neblocanta permite executarea altor operatii, chiar daca inca nu a ajuns la finalul executiei.

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

Socketii UNIX sunt mecanisme de IPC(interprocess communication) si permit schimb de date bidirectional intre procese care ruleaza in acelasi sistem, in timp de socketii de retea sunt utili in cazul comunicatiei intre procese situate in retele/sisteme diferite.

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

Pe langa cea evidenta, adica pipe-ul anonim nu are un nume si este accesibil prin intermediul a doi file descriptori generati de functia pipe(), in timp ce un pipe cu nume primeste numele de la user prin care este referentiat de catre reader/writer, un named pipe poate avea mai multe procese care pot comunica prin acesta, dar unul anonim permite numai comunicarea unidirectionala intre un proces parinte si unul copil.(master-slave)

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

La lucrul cu fisiere pe disc, apelurile read/write nu acceseaza direct spatiul de pe disc, ci copiaza date dintr-un buffer din user-space intr-unul din kernel-space denumit 'buffer cache'. Scopul este de a nu genera timpi morti cauzati de operatii repetate si incete pe disc.

PAstreaza intr-un fel de hash blocurile recent citite de pe disk. Mai ai si page cache, unde pastrezi pagini, nu blocuri.

De ce operația write pe fișiere este foarte rar blocantă?

E rar blocanta fiindca procesul de copiere a continutului buffer-ului din user-space in kernel-space dureaza foarte putin. Cazuri in care devine blocanta sunt rare, precum atunci cand nu exista pagini libere si unele trebuie sa fie eliberate si cazul in care write-ul are ca target mijlocul unui fisier, ceea ce presupune citirea continutului din jurul pozitiei.

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

By default read() se blocheaza pana cand cel putin un byte e disponibil spre a fi returnat catre aplicatie, dar e posibil sa ii schimbi comportamentul in non-blocking daca esti suficient de ambitios si ii setezi flag-ul corespunzator file descriptorului (O_NONBLOCK).

As spune ca exista un buffer(Block I/O layer) al discului unde se poate face read-ahead, sa se aduca in acel buffer date din zona accesata. In momentul in care se face read se citeste din buffer. Insa, daca nu sunt date suficiente apelul de sistem devine blocant.

Care este rolul unui device driver?

Rolul sau este de a permite unui device hardware sa comunice cu sistemul de operare.(Ex: transfer de date)

Ce rol are controller-ul hardware?

Administreaza corespunzator un periferic (cunoaste specificitatile perifericului respectiv).

Care este rolul DMA-ului (Direct Memory Access)?

DMA-ul faciliteaza transferul de date I/O direct din si in memorie fara interventia procesorului, acesta avand rolul doar de a initia transferul.

Când are sens să folosim polling în loc de întreruperi?

Ar avea sens sa preferam polling-ul daca evenimentele pe care le gestionam sunt sincrone, frecvente(majoritatea ciclurilor de polling genereaza hit-uri) si timpul cat faci polling este limitat(cat mai mic posibil fiindca altfel irosesti cicli de procesare). Altfel, daca evenimentele sunt asincrone, rare si au prioritate crescuta in cadrul aplicatiei preferam intreruperile.

Ce înseamnă zero-copy? Ce mecanism/apel folosește zero-copy?
Zero-copy presupune transferul unei informatii intre 2 buffere fara trecerea prin user-space.
Linux: splice(), sendfile()
Windows: TransmitFile()

Ce rol are mecanismul de TCP offload engine?
Rolul sau este de a usura munca CPU-ului in a gestiona protocolul TCP/IP. TOE-urile sunt concepute de producatorii de placi de retea, fiind implementate direct in hardware. In Linux nu prea ai suport pentru asa ceva deoarece conceptul paraseste ideea de a conferi kernel-ului accesul permenent si total la intreg setul de resurse al sistemului at any time.

Care este sursa primară pentru care un apel send() pe un socket TCP se blochează?
TCP - send se blocheaza ptc receiverul are bufferul plin sau daca send bufferul e plin

Care este sursa primară pentru care un apel send() pe un socket UDP se blochează?
Ai un buffer unde primesti mesaje. Daca socketul spre care faci send nu face recvfrom() pentru a-si goli acel buffer, tu nu mai ai cum sa ii trimiti nimic pana nu incepe sa le receptioneze.

Ce garanții ni se oferă în momentul în care apelul send() se întoarce în user space?
Datele au fost copiate intr-un send buffer.

Cu ce diferă afișarea folosind printf() față de folosirea write()?
Cred ca se refera la faptul ca printf() (nu sprintf() sau fprintf()) are ca standard printatul la stdout.
Lui write trebuie sa ii specifici fd-ul corespunzator lui stdout(1), de exemplu.
ssize_t write(int *fd***, const void ****buf***, size_t** *count***); int printf(const char ****format***, ...);**

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

Ce rol are apelul / comanda sync?
The **sync** command forces an immediate write of all cached data to disk.
Sync muta din buffer cache-ul din kernel space pe disk. Noi cand facem write facem write in acest buffer cache, nu pe disk direct. Apelul sync duce datele din acest buffer pe disk.

Care este rolul apelului ioctl / DeviceIoControl?
IOCTL. **IOCTL** is referred as Input and Output Control, which is **used** to talking to device drivers. This system call, available in most driver categories. The major **use** of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.

De ce în general doar utilizatorul root are permisiuni de scriere (uneori doar root are permisiuni de citire) pe intrările din /dev?
? Daca nu esti atent si modifici de capul tau prin /dev files, un device hardware ar putea deveni unusable ? (Cred)
Pentru ca accesul direct la hardware se face doar din modul privilegiat
Daca oricine ar putea face modificari pe modul de functionare ale device driverelor, atunci ar putea aparea probleme de securitatea, se pot crea vulnerabilitati si comunicarea cu dispozitivele de I/O ar fi nefunctionala. Dorim sa pastram intacta comunicarea cu aceste

dispozitive. Se specifica faptul ca oricum procesorul prioriteaza comunicarea cu Dispozitivele de I/O fara a exista interferente cu ce scriu alte aplicatii in memoria RAM.

Intrebari – teste moodle

Ce operatie NU necesita apel de sistem?
copierea unui sir in alt sir (crearea unui proces, alocarea memoriei, inchiderea unui socket, citirea din fisier DA)

Care este ordinea corecta in stiva software?
aplicatie, middleware, biblioteci, sistem de operare, hardware

Ce situatie are cel mai putin nevoie de S.O.?
un sistem cu un singur proces

Care este neajunsul unui apel de sistem?
overhead

Ce este un descriptor de fisiere?
un index intr-o tabela

Care dintre afirmatiile despre apeluri de biblioteca si de sistem este adevarata?
un apel de sistem cauzeaza mai mult overhead decat un apel de biblioteca

Cate tabele de descriptori de fisiere se gasesc la nivelul SO?
cate una per proces

Care este diferenta dintre o biblioteca si sistemul de operare?
sistemul de operare ruleaza in kernel mode, biblioteca nu
bibliotecile sunt linkate cu executabile/procese, sistemul de operare nu

Care dintre urmatoarele apeluri NU are niciun efect asupra dimensiunii fisierului?
read, lseek (write, truncate, open DA)

Care dintre urmatoarele apeluri NU are niciun efect asupra cursorului de fisier?
truncate (read, lseek, open, write DA)

Care dintre urmatoarele definitii este cea mai potrivita pentru tabela descriptorilor de fisiere?
struct channel *fd_table[1024];

Ce apel duce la crearea unei noi structuri de tip channel/open file?
open

Ce apel este folosit pentru a recupera informatii despre un proces care si-a incheiat executia?
wait

Ce argument primeste apelul de creare a unui proces?
calea catre fisierul executabil

Cum gestioneaza SO accesul la procesor pentru mai multe procese?
se aloca o cuanta de timp pentru fiecare proces

Ce este un proces zombie?
un proces care si-a incheiat executia si nu a fost asteptat de procesul parinte

Ce intoarce apelul fork() in procesul copil?
0

Care dintre urmatoarele apeluri creeaza un proces nou?
fork(), popen()

Cate procese vor fi create noi dupa 2 apeluri fork() consecutive? Nu includem procesul initial.
3

Ce se schimba intr-un proces in urma apelului exec()?
spatiul de adrese

Ce cauzeaza tranzitia unui proces din RUNNING in READY?
procesului ii expira cuanta, procesul cedeaza voluntar procesorul (yield)

Ce este planificatorul de executie?
o functie

Cate procese se pot gasi in RUNNING?
egal cu numarul de procesoare

Ce planifica planificatorul de executie?
thread-uri

Care dintre urmatoarele cauzeaza segmentation fault?
accesarea unei zone invalide/nealocate de memorie, accesarea unei zone cu permisiuni insuficiente

Care sunt caracteristicile zonelor dinamice din spatiul virtual de adrese al unui proces?
nu sunt definite in executabil, au dimensiune variabila

Cate spatii virtuale de adrese sunt in cadrul unui SO?
egal cu numarul de procese

Cum este reprezentat la nivelul SO un spatiu virtual de adrese?
o lista cu zone de memorie

Care sunt apelurile de sistem pe care le efectueaza malloc()?
mmap, brk

Care dintre urmatoarele zone de memorie NU au permisiunile read & write?
text, rodata (read-only) (bss, data – read&write)

Ce componenta software gestioneaza heap-ul?
biblioteca standard C

Cum se dezaloca memorie de pe stiva?
se dezaloca automat la iesirea din functie

Cate pagini fizice sunt intr-un sistem?
dimensiunea memoriei RAM / dimensiunea unei pagini

Care sunt conditiile pentru a genera page fault?
intrarea din tabela de pagini are bitul “invalid” activat, intrarea din tabela de pagini are permisiuni neconforme

Care este dimensiunea uzuala a unei pagini de memorie pe sistemele x86 si ARM?
4KB

Ce NU contine o intrare din tabela de pagini?
adresa paginii virtuale (permsiunile paginii, adresa paginii fizice, daca pagina este valida sau nu DA)

Ce situatie va cauza intotdeauna ca un page fault sa conduca la livrarea unei exceptii de acces la memorie (SIGSEGV) procesului curent?
pagina nu este parte dintr-o zona din spatiul virtual de adrese al procesului

Cand se modifica valoarea PTBR (page table base register)?
la schimbarea de context intre doua procese

2 procese sunt pornite din acelasi executabil. Care dintre zonele executabilului vor putea fi partajate prin intermediul tabelelor de pagini, de spatiile virtuale ale celor doua procese?
text, rodata

Care sunt cele doua mecanisme care reduc overhead-ul temporal, respectiv spatial cauzat de implementarea memoriei virtuale cu tabele de pagini?
TLB, respectiv paginare multi-level/ierarhica

Care sunt cele doua permisiuni care nu pot fi reprezentate simultan intr-o intrare in tabela de pagini din ratiuni de securitate?
scriere si executie

Care dintre urmatoarele variante reprezinta code pointeri?
adresa de retur a functiei, pointer de functie

Ce apel este folosit intr-un atac de tipul code reuse pentru a tranzita la un atac de tipul code injection?
mprotect

Ce contine un shellcode?
cod obiect

La ce intrebare raspunde continutul stivei la un moment dat?
ce am facut?

Care este avantajul implementarii unei aplicatii in format multi-process in loc de multi-threaded?
izolare

Ce au diferit doua thread-uri apartinand aceluiasi proces?
stiva

Ce informatii se aloca la crearea unui nou thread?
o stiva, un TCB (thread control block)

Ce zone de memorie sunt proprii fiecarui thread?
stiva, thread-local storage

Unde se gaseste zona TLS pentru thread-urile create de un program?
pe stiva fiecarui thread

Un proces contine 8 thread-uri de tip user-levvel. Cate core-uri poate folosi maxim procesul?
1

In ce context functioneaza corect functiile thread-safe, respectiv functiile reentrante?
cele thread-safe in context multithreaded, cele reentrante in context de handler de semnal si multi-threaded

Ce primitiva de sincronizare are in implementarea functiei lock folosita o instructiune de tip atomic_cmpxchg?
spinlock

In ce situatie NU este cert atomica instructiunea add [ebp – 32], eax?
pe un sistem multiprocesor

Care sunt cele doua metode aferente unei primitive de ordonare? (NU de acces exclusiv)?
notify, wait

Ce rol are prefixul lock in fata unei instructiuni x86?
serializeaza accesul la magistrala de lucru cu memoria

Care sunt cele doua moduri de adresare a registrelor I/O pe x86?
port-mapped I/O, memory-mapped I/O

Ce operatie pe ce dispozitiv are implementarea “return 0”?
citire din dev/null

Cand o operatie de read pe un descriptor se blocheaza?
cand buffer-ul corespunzator din kernel este gol

Care sunt cele doua tipuri de canale de I/O care au ca endpoint, respectiv un dispozitiv si un proces?
fisier, socket

Ce informatie NU se gaseste in inode-ul (FCB-ul) unui fisier?
cursorul de fisier, numele fisierului (timpi de acces, owner, dimensiune, inode number, pointeri la blocurile de date DA)

Ce efect are comanda rm?
decrementeaza numarul de link-uri al unui fisier

Cati octeti poate citi un apel neblocant read care solicita N octeti?
intre 0 si N inclusiv

Se ruleaza doua comenzi: cp a.dat b.dat si cp a.dat c.dat. De ce a doua dureaza mai putin decat prima?
dupa prima comanda, fisierul a.dat se gaseste in buffer cache

Ce comanda ajunge sa citeasca cel putin un dentry?
cat, rm, stat, ls, find

Care este numarul minim de link-uri detinut de un inode de tip director?
2

Care dintre urmatoarele ocupa cel mai putin spatiu?
hard link

In ce zona din layout-ul sistemului de fisier se gasesc dentry-urile?
dzone (zona de date)

Cate buffere are asociate fiecare socket?
doua: unul pentru send si unul pentru receive

Cand se blocheaza apelul send() pe socket?
cand buffer-ul socket-ului este plin

Cati socket-uri TCP pot exista la un moment dat in sistem?
oricate, in limita resurselor sistemului

In ce situatie apelul recv() pe socket intoarce 0?
cand celalalt capat al socket-ului a inchis conexiunea

Care este frecventa de apeluri cea mai probabila cand shell-ul creeaza un proces din comanda ls?
fork, exec, wait, exit

Care dintre urmatoarele apeluri creeaza un pipe?
popen

Care este valoarea descriptorului de fisier intors de primul apel de deschidere a unui canal/fisier?
3

Care sunt valori posibile pentru cursorul de fisier imediat dupa deschiderea unui fisier?
0, dimensiunea fisierului

Ce NU contine structura de canal I/O / open file din kernel?
descriptorul de fisier (cursorul de fisier, pointer catre metadatele fisierului, permisiunile de deschidere ale fisierului DA)

Care este functia apelata o data si care se intoarce de doua ori?
fork

Ce efect are operatorul & in shell?
nu se mai apeleaza wait() imediat dupa crearea procesului

Cate zone de tip .text/.code sunt in spatiul virtual de adrese al unui proces?
1 + numar de biblioteci

Cate stive sunt in spatiul virtual de adrese al unui proces?
numar de thread-uri

Care apel sigur NU extinde/diminueaza dimensiunea unei zone din spatiul de adrese al unui proces?
mprotect() (malloc, strdup, free, strcpy DA)

Fie zona virtuala Z, definita de: S = adresa de start, L = dimensiunea zonei, P = permisiunile zonei. Accesam adresa virtuala A care este la offset-ul O de adresa de start S a zonei. Ce conditie duce la primirea segmentation fault?
A – S > L, S + O > S + L

Cate tabele de mapare a memoriei exista intr-un SO?
cate una per proces

Doua zone de memorie ale aceluiasi proces se gasesc la adrese virtuale de start VS1 si VS2. Adresele fizice de start sunt PS1 si PS2. Stim ca VS1 < VS2. Care afirmatie este adevarata?
PS1 != PS2

Ce este cheia in tabela de mapare pentru memorie?
adresa de start a zonei virtuale

Care dintre urmatoarele variabile sunt proprii fiecarui thread?
o variabila locala nemarcata static

Unde NU se gasesc variabile partajate thread-urilor?
pe stiva (pe heap, in datele din biblioteca standard C, in .bss, in .data DA)

1000 de thread-uri incrementeaza valoarea unei variabile initializate la 0. Care este valoarea la final a variabilei?
mai mic sau egal decat 1000

Care dintre urmatoarele functii sunt reentrante?
char *strcpy(char *dest, char *src)

Care dintre comenzi creeaza un hard link b.txt?
touch b.txt, mv a.txt, b.txt, ln a.txt b.txt, cp a.txt b.txt, ln -s a.txt b.txt, mkdir b.txt

Ce contine inode-ul unui symlink?
o cale

Cand apare un danglink link?
cand stergem un fisier referit de un symbolic link

Un inode are 2 blocuri de cate 4096 octeti fiecare. Care dimensiune NU poate fi asociata fisierului?
4096

Care dintre urmatoarele optiuni ne vor ajuta sa adaugam continut la sfarsitul unui fisier existent?
Deschidem fisierul cu O_WRONLY | O_APPEND; apelam write()
Deschidem fisierul cu O_RDWR si executam un lseek la sfarsit inainte de a apela write()

Care din afirmatiile de mai jos este adevarata?
Microkernel-ul are un TCP (Trusted Computing Base) mai mic si deci, mai sigur

SO NU are rolul de a:
actualiza utilitarele de sistem (asigura securitatea sistemului, mediaza accesul la resursele hardware, gestioneaza resursele hardware ale sistemului de calcul DA)

Care NU este atribuit unui proces?
versiunea de compilator (PID, user, tabela de descriptori de fisier DA)

Intr-un kernel monolitic:
Management-ul memoriei este facut direct de catre kernel

Ce afiseaza codul urmator atunci cand fork() reuseste?

```
int i = 0;
if (fork() == 0) i++;
else sleep(1);
printf("%d\n", i);
```

Va afisa 1 scris de copil, 0 scris de parinte

Care este diferenta dintre un proces zombie si unul orfan?

Un proces orfan este un proces al carui parinte a murit, iar unul zombie este un proces care a murit, dar la care parintele nu a facut wait.