

Faktické otázky:

MOBV:

Q: Koľko kapitol má kniha Evil by design od autora Chris Nodder ?

A: Kniha "Evil by Design" od autora Chris Nodder má 7 kapitol. Každá z týchto kapitol je pomenovaná podľa jedného z siedmich smrtných hriechov:

Pride: pýcha

Sloth: záhaľka

Gluttony: nenásytnosť

Anger: hnev

Envy: závišť

Lust: chtivosť

Greed: chamtivosť

Sources:

[moc01.pdf](#) (Page 82)

[moc01.pdf](#) (Page 83)

[moc01.pdf](#) (Page 81)

total time = 4594.01 ms / 584 tokens

CHAT:

Kniha "Evil by Design" od autora Chris Nodder má 8 kapitol. Každá kapitola sa zaoberá iným aspektom užívateľského rozhrania a psychológie, ktoré sa využívajú na ovplyvňovanie chovania používateľov.

total time = 2305.38 ms / 252 tokens

Q: V ktorom roku bol vydaný prvý prototyp aplikácie Musical.ly známy pod menom TikTok ?

Prvý prototyp aplikácie Musical.ly, ktorá je známa pod menom TikTok, bol vydaný v apríli 2014.

Sources:

- moc01.pdf (Page 92)
- moc01.pdf (Page 92)
- moc01.pdf (Page 95)

total time = 1458.31 ms / 448 tokens

CHAT:

Prvý prototyp aplikácie Musical.ly, ktorá sa neskôr stala známa ako TikTok, bol vydaný v roku 2014. Aplikácia bola vytvorená spoločnosťou ByteDance a pôvodne bola dostupná iba v Číne. V roku 2017 bola aplikácia Musical.ly zjednotená s TikTocom a odvtedy sa rozširovala po celom svete.

total time = 3182.23 ms / 130 tokens

Q: Aké elementy jednoduchosti poznáme podľa Dr. Fogga ?

Podľa Dr. BJ Fogga poznáme šesť elementov jednoduchosti, ktoré ovplyvňujú vykonanie akcie. Tieto elementy sú:

1. Čas: Koľko času treba na vykonanie akcie.
2. Peniaze: Koľko akcia stojí.
3. Fyzická náročnosť: Koľko práce treba vykonať.
4. Mentálna námaha: Koľko treba pritom rozmýšľať.
5. Sociálna deviácia: Ako je akcia akceptovaná spoločnosťou.
6. Nerutinnosť: Ako sa akcia podobá alebo nepodobá existujúcim rutinám.

Tieto elementy pomáhajú zhodnotiť, ako jednoduchá alebo zložitá je akcia, čo je kľúčové pre porozumenie a ovládnutie správania sa.

Sources:

- moc01.pdf (Page 48)
- moc01.pdf (Page 45)

- moc01.pdf (Page 44)

total time = 7022.66 ms / 678 tokens

CHAT:

Podľa Dr. Fogga poznáme tri hlavné elementy jednoduchosti:

1. **Jednoduchosť v užívateľskom rozhraní (User Interface Simplicity):** To znamená, že rozhranie by malo byť intuitívne a ľahko použiteľné. Užívateľ by mal rýchlo pochopiť, ako s aplikáciou pracovať, bez potreby dlhého učenia sa.
2. **Jednoduchosť v architektúre (Architectural Simplicity):** To sa týka vnútornej štruktúry aplikácie. Architektúra by mala byť čistá a prehľadná, čo usnadňuje údržbu a rozširovateľnosť softvéru.
3. **Jednoduchosť v kóde (Code Simplicity):** Kód by mal byť čitateľný a porozumiteľný. To zjednodušuje spoluprácu medzi vývojármi a znižuje pravdepodobnosť vzniku chýb.

Tieto tri elementy sú kľúčové pre vytvorenie efektívneho a udržateľného softvéru.

total time = 8475.43 ms / 288 tokens

Q: Čo znamená skratka PBL v kontexte gamifikácie?

V kontexte gamifikácie znamená skratka PBL Points, Badges, Leaderboards. Ide o tri základné herné prvky, ktoré sa často využívajú pri gamifikácii, aby sa zvýšila motivácia a angažovanosť používateľov. Konkrétne:

- Body (Points): Bodové systémy, ktoré používatelia získavajú za dosiahnutie určitých cieľov alebo úloh.
- Odznaky (Badges): Vizuálne odznaky alebo odznaky, ktoré používatelia získavajú za dosiahnutie určitých úspechov alebo úrovní.
- Rebríčky (Leaderboards): Zoznamy, ktoré zobrazujú poradie používateľov podľa ich dosiahnutých bodov alebo úspechov, čím sa podnecuje súťaživý duch.

Tieto prvky pomáhajú zvyšovať záujem a angažovanosť používateľov v rôznych neherných aktivitách, ako sú vzdelávanie, pracovné úlohy alebo používateľské aplikácie.

Sources:

- moc02.pdf (Page 6)
- moc02.pdf (Page 37)
- moc02.pdf (Page 6)

total time = 8243.61 ms / 540 tokens

CHAT:

V kontexte gamifikácie skratka PBL znamená **Problem-Based Learning** (učenie sa prostredníctvom riešenia problémov). PBL je metóda učenia, ktorá sa zameriava na riešenie reálnych alebo simulovaných problémov, aby študenti získali hlbšie porozumenie a praktické dovednosti. V gamifikácii sa tato metóda často používa na vytvorenie interaktívnych a angažujúcich učebných zážitkov, kde študenti riešia problémy, ktoré sú súvisiace s učivom, a tým sa učí efektívnejšie a zaujímavejšie.

total time = 5242.70 ms / 178 tokens

Q: Aká bola cena Picturephonu v roku 1964 a na koľko by to vyšlo dnes ?

Podľa informácií z prednášky, cena Picturephonu v roku 1964 bola 160 USD za mesiac. Dnes by to vyšlo približne na 1000 USD za mesiac.

Sources:

- moc01.pdf (Page 8)
- moc01.pdf (Page 9)
- Prednaska_4new.pdf (Page 3)

total time = 2145.25 ms / 389 tokens

CHAT:

V roku 1964 bola cena Picturephonu približne 16 500 dolárov. Aby sme zistili, koľko by to vyšlo dnes, môžeme použiť infláciu. Inflácia je mierou, ako sa zmenila hodnota peňazí v čase.

Podľa údajov z roku 2023 by suma 16 500 dolárov z roku 1964 bola ekvivalentná približne 150 000 dolárov. Je to však iba odhad, ktorý sa môže líšiť v závislosti od použitého inflačného kalkulačky a roka, na ktorý sa odkazujete.

Táto informácia je založená na všeobecných vedomostiach a nie je priamo z nahratých prednášok.

total time = 6125.10 ms / 227 tokens

OS:

Q: Aké režimy činnosti má RISC-V CPU?

RISC-V CPU má tri režimy činnosti:

1. Supervisor režim: V tomto režime beží kernel (jadro operačného systému). Supervisor režim má rozšírené práva a môže vykonávať operácie, ktoré sú v user režime zakázané, napríklad prístup k zariadeniam, nastavovanie adresného priestoru a prístup k špeciálnym registrom CPU.
2. User režim: V tomto režime bežia bežné používateľské programy. User režim má obmedzené práva a nemôže vykonávať operácie, ktoré by mohli ohroziť stabilitu alebo bezpečnosť systému.
3. Machine režim: Tento režim je určený pre najvyššiu úroveň kontroly nad hardvérom a je obvykle využívaný pri inicializácii systému alebo pri vykonávaní úloh, ktoré vyžadujú priamý prístup k hardvéru.

Tieto režimy umožňujú RISC-V CPU efektívne rozdeľovať práva a zabezpečiť systém pred neoprávneným prístupom.

Sources:

- o 04-system_call.pdf (Page 10)
- o 04-system_call.pdf (Page 11)
- o 04-system_call.pdf (Page 13)

total time = 9519.97 ms / 657 tokens

CHAT:

RISC-V CPU má niekoľko režimov činnosti, ktoré určujú úroveň prístupu k zdrojom a funkciám procesora. Hlavné režimy činnosti sú:

1. **User Mode (U-mode):** Tento režim je určený pre bežné aplikácie a poskytuje najnižšiu úroveň prístupu. Aplikácie v tomto režime nemajú prístup k kritickým systémovým zdrojom a operáciám.

2. **Supervisor Mode (S-mode):** Tento režim je určený pre operačné systémy. Poskytuje vyššiu úroveň prístupu ako U-mode a umožňuje správať systémové zdroje, ako sú pamäť a periférie.
3. **Machine Mode (M-mode):** Tento je najvyššou úrovňou prístupu a je určený pre inicializáciu a diagnostiku systému. Má plný prístup k všetkým zdrojom a funkciám procesora.
4. **Hypervisor Mode (H-mode):** Tento režim je určený pre virtuálne mašiny a hypervisory. Poskytuje úroveň prístupu medzi S-mode a M-mode, umožňujúc správať virtuálne mašiny.

Každý z týchto režimov má svoje vlastné registre a kontexty, ktoré umožňujú bezpečné a efektívne správanie sa s rôznymi úrovňami prístupu a funkciami procesora.

total time = 10661.04 ms / 346 tokens

Q: Akú veľkosť má stránka v stránkovaní RISC-V architektúry?

V RISC-V architektúre má stránka veľkosť 4 KiB. Stránky sú zarovnané na hranici 4 KiB, čo znamená, že začiatok každej stránky je násobkom 4 KiB (napríklad 0, 4 KiB, 8 KiB, 12 KiB atď.).

Sources:

- 03-virtual-memory-2.pdf (Page 34)
- 03-virtual-memory-2.pdf (Page 36)
- 03-virtual-memory-2.pdf (Page 37)

total time = 2901.24 ms / 420 tokens

CHAT:

4096 bajtov

total time = 557.36 ms / 36 tokens

Q: Akú má veľkosť v bajtoch i-uzol na disku v xv6?

I-uzol na disku v operačnom systéme xv6 má veľkosť 64 bajtov.

Sources:

- 08-fs-1.pdf (Page 25)
- 08-fs-1.pdf (Page 24)
- 03-virtual-memory-2.pdf (Page 47)

total time = 1425.77 ms / 524 tokens

CHAT:

V operačnom systéme xv6 má i-uzol (inode) veľkosť 64 bajtov. I-uzol obsahuje metadáta súboru, ako napríklad práva, veľkosť, časové údaje a ukazovatele na bloky dát, ktoré sú súčasťou súboru.

total time = 2653.62 ms / 103 tokens

Q: Čo obsahuje blok 45 v štruktúre disku xv6 ?

Blok 45 v štruktúre disku xv6 obsahuje blok bitmapy využitia blokov. Táto bitmapa slúži na mapovanie, ktoré bloky sú voľné a ktoré sú použité. Voľný blok je označený hodnotou 0, zatiaľ čo použitý blok je označený hodnotou 1.

Sources:

- 08-fs-1.pdf (Page 18)
- 06-threads.pdf (Page 79)
- 06-threads.pdf (Page 75)

total time = 2937.56 ms / 468 tokens

CHAT:

V operačnom systéme xv6 je štruktúra disku organizovaná do blokov, ktoré obsahujú rôzne systémové a užívateľské dáta. Blok 45 v štruktúre disku xv6 obsahuje **root súborový systém** (root filesystem). Tento súborový systém je základným súborovým systémom, ktorý sa nachádza na disku a obsahuje kritické systémové súbory a adresáre, ktoré sú potrebné pre spustenie a fungovanie operačného systému.

Root súborový systém v xv6 obsahuje napríklad:

- Adresár /, ktorý je koreňovým adresárom súborového systému.
- Kritické systémové súbory a skripty.

- Základné binárne súbory a knihovny.

Tento súborový systém je zásadný pre správnu funkčnosť operačného systému a jeho spustenie.

total time = 6661.99 ms / 234 tokens

Q: Akým príkazom sa spúšťa xv6 vo nástroji qemu ?

Xv6 sa spúšťa vo virtualizačnom nástroji Qemu pomocou príkazu:

make qemu

Tento príkaz spustí xv6 vo virtualizačnom prostredí Qemu, ktoré emuluje počítač s architektúrou RISC-V.

Sources:

- 02.partA_.pdf (Page 27)
- 01-1.pdf (Page 61)
- 02.partA_.pdf (Page 23)

total time = 2743.61 ms / 443 tokens

CHAT:

V nástroji QEMU sa operačný systém xv6 spúšťa pomocou príkazu make qemu. Tento príkaz sa vykoná v priečinku, kde je zdrojový kód xv6, a spustí emulátor QEMU s xv6. Uistite sa, že máte nainštalovaný QEMU a všetky potrebné závislosti pre kompiláciu xv6.

total time = 3370.11 ms / 127 tokens

PPDS:

Čo znamená skratka CUDA?

Skratka CUDA znamená Compute Unified Device Architecture. Ide o platformu a API vyvinutú spoločnosťou NVIDIA, ktorá umožňuje programátorom využívať výpočtové zdroje GPU (Graphics Processing Unit) pre všeobecné výpočty mimo grafických aplikácií. CUDA umožňuje paralelné programovanie, čím sa zvyšuje výkonnosť pri riešení komplexných výpočtových úloh.

Sources:

- 2025-07.cuda.pdf (Page 13)

- 2025-07.cuda.pdf (Page 7)
- 2025-07.cuda.pdf (Page 10)

total time = 3502.45 ms / 609 tokens

CHAT:

CUDA je skratka pre **Compute Unified Device Architecture**. Je to platforma a API (Application Programming Interface) vyvinutá spoločnosťou NVIDIA, ktorá umožňuje programátorom využívať výpočtové zdroje grafických procesorov (GPU) pre riešenie výpočtovo náročných úloh. CUDA umožňuje paralelné programovanie, čo znamená, že môže rýchlo spracovávať veľké množstvá dát a úloh súbežne, čo je zvlášť užitočné v oblastiach ako je vedecký výskum, strojové učenie, simulácie a ďalšie.

CUDA je široko používaná v rôznych odvetviach, vrátane akademického výskumu, priemyslu a v rozvoji softvéru.

total time = 5731.09 ms / 367 tokens

Koľko jadier tvorí warp v architektúre CUDA ?

V architektúre CUDA tvorí warp 32 jadier. Warp je základná vykonávacia jednotka CUDA programov a skupina týchto 32 jadier je spracovávaná spoločne.

Sources:

- 2025-07.cuda.pdf (Page 33)
- 2025-07.cuda.pdf (Page 7)
- 2025-07.cuda.pdf (Page 11)

total time = 1900.28 ms / 514 tokens

CHAT:

V architektúre CUDA je warp základnou jednotkou, ktorá sa skladá z 32 vlákien (threads). Tieto vlákna sú synchronizované a vykonávajú rovnaké inštrukcie v danom čase, ale môžu pracovať s rôznymi údajmi. Warp je kľúčovým konceptom v CUDA, pretože umožňuje efektívne využitie paralelizmu v GPU. Ak máte ďalšie otázky ohľadom CUDA alebo iných tém z prednášok, neváhajte sa opýtať!

total time = 4145.01 ms / 154 tokens

Aká operácia v MPI distribuuje dáta všetkým procesom v komunikátore?

Operácia v MPI, ktorá distribuuje dáta všetkým procesom v komunikátore, je **MPI_Bcast**. Táto operácia rozposiela správu z jedného procesu (obvykle procesu s rankom 0) všetkým ostatným procesom v danom komunikátore.

Pre viac informácií o základných funkciách MPI, pozrite si stránku 13 v dokumente MPI_ppds_2025_08.pdf.

Sources:

- MPI_ppds_2025_08.pdf (Page 13)
 - MPI_ppds_2025_08.pdf (Page 4)
 - MPI_ppds_2025_08.pdf (Page 25)
- total time = 3625.06 ms / 577 tokens

CHAT:

V MPI (Message Passing Interface) operácia, ktorá distribuuje dáta všetkým procesom v komunikátore, je **MPI_Bcast** (broadcast). Táto operácia umožňuje jednomu procesu (obvykle procesu s rankom 0) poslať dáta všetkým ostatným procesom v danom komunikátore. MPI_Bcast je účinný spôsob rozosielania informácií, keď je potrebné, aby všetky procesy mali k dispozícii rovnaké dáta.

total time = 3612.78 ms / 141 tokens

Ako sa volá najvýkonnejší počítač na Slovensku ?

Na základe informácií z prednášok, ktoré máte k dispozícii, nie je možné určiť najvýkonnejší počítač na Slovensku. Prednáška sa zameriava na superpočítače zverejnené v rokoch 2022 a 2025, ale konkrétny superpočítač na Slovensku nie je spomenutý.

Ak by ste mali ďalšie informácie alebo konkrétne zdroje o superpočítačoch na Slovensku, môžete sa na ne odvolať. V opačnom prípade by bola odpoveď na vašu otázku mimo rozsahu poskytnutých informácií.

Sources:

- Prednaska_4new.pdf (Page 6)
- Prednaska_4new.pdf (Page 5)

- Prednaska_4new.pdf (Page 5)

total time = 5007.33 ms / 721 tokens

CHAT:

Na Slovensku je najvýkonnejším počítačom superpočítač **Salomon**. Tento superpočítač je umiestnený v Centrum informačných a komunikačných technológií Slovenskej akadémie vied (CIVIT) a je určený pre vedecké výpočty a výskum. Salomon je schopný vykonávať veľmi náročné výpočty a je využívaný v rôznych vedeckých disciplínach, ako je fyzika, chemia, biológia a iné.

total time = 3572.72 ms / 151 tokens

Kedy sa paralelný algoritmus nazýva cenovo optimálny ?

Paralelný algoritmus sa nazýva cenovo optimálny, ak jeho cena ($C(n, p)$) je asymptoticky rovnaká ako časová zložitosť najlepšieho sekvenčného algoritmu ($T_1(n)$). To znamená, že platí:

$$[C(n, p) = O(T_1(n))]$$

Toto znamená, že paralelný algoritmus je cenovo optimálny, ak jeho náklady na vykonanie sú v poradí veľkosti časovej zložitosti najlepšieho sekvenčného algoritmu pre daný problém.

Sources:

- Prednaska_5_2_25.pdf (Page 9)
- PPDS_Prednaska-5_1_25.pdf (Page 8)
- Prednaska_5_2_25.pdf (Page 9)

total time = 4403.37 ms / 673 tokens

CHAT:

Paralelný algoritmus sa nazýva cenovo optimálny, ak jeho časová zložitosť je optimálna vzhľadom na počet procesorov, ktoré sa na jeho vykonávaní podieľajú. To znamená, že algoritmus je cenovo optimálny, ak jeho časová zložitosť je najlepšia možná pre daný počet procesorov a daný problém. V ideálnom prípade by časová zložitosť paralelného algoritmu mala byť lineárna voči počtu procesorov, teda ($T(p) = O(T(1)/p)$), kde ($T(p)$) je časová zložitosť algoritmu s (p) procesormi a ($T(1)$) je časová zložitosť algoritmu s jedným procesorom.

Cenová optimalita je dôležitá, pretože ukazuje, že algoritmus efektívne využíva dostupné zdroje a dosahuje maximálnu rýchlosť zvyšovaním počtu procesorov. To je zvlášť dôležité v súčasných výpočtových systémoch, kde je dostupné veľké množstvo procesorov a je potrebné efektívne ich využívať.

total time = 8520.09 ms / 303 tokens

UPD:

Napiš rizikovú rovnicu v kontexte počítačovej bezpečnosti.

Rizikovú rovnicu v kontexte počítačovej bezpečnosti môžeme vyjadriť nasledovne:

$$[R = T \cdot V \cdot C]$$

kde:

- (T) predstavuje informáciu o hrozbách, konkrétne pravdepodobnosť realizácie hrozby.
- (V) predstavuje informáciu o zraniteľnostiach, teda slabých miestach systému, ktoré môžu byť využité pri útoku.
- (C) predstavuje hodnotu aktíva, teda odhad dopadov útoku na systém alebo organizáciu.

Táto rovnica pomáha kvantifikovať riziko a určiť, ako vážne môže byť riziko pre systém alebo organizáciu.

Sources:

- p21-01.pdf (Page 29)
- p21-01.pdf (Page 33)
- p21-10.pdf (Page 34)

total time = 5204.11 ms / 173 tokens

CHAT:

V kontexte počítačovej bezpečnosti môžeme definovať rizikovú rovnicu ako nástroj na kvantifikáciu a porovnanie rôznych rizík, ktoré môžu ohroziť systém alebo organizáciu. Riziková rovnica je zvyčajne definovaná ako:

$$[\text{Riziko} = \text{Práca} \times \text{Útok} \times \text{Zraniteľnosť}]$$

1. **Práca (Asset):** Reprezentuje hodnotu alebo dôležitosť zdroja, ktorý môže byť ohrozený. To môže byť údaje, systémy, služby alebo iné cenné zdroje.

2. **Útok (Threat):** Reprezentuje možnosť, že sa niekto pokúsi zneužiť zraniteľnosť. To môže byť vnútorný alebo vonkajší útočník.
3. **Zraniteľnosť (Vulnerability):** Reprezentuje slabiny alebo chyby v systéme, ktoré môžu byť využité útočníkom.

Riziková rovnica umožňuje bezpečnostným odborníkom identifikovať a porovnávať rôzne riziká, aby mohli prioritizovať opatrenia na ich zníženie. Napríklad, ak máte systém s vysokou hodnotou (práca), ktorý je vystavený viacerým útokom (útok) a má viacero zraniteľností (zraniteľnosť), riziko je vysoké a je potrebné zvážiť opatrenia na jeho zníženie.

total time = 10937.69 ms / 590 tokens

Aké kategórie hrozieb opisuje STRIDE metodika ?

STRIDE metodika zameriava sa na šesť hlavných kategórií hrozieb:

1. Spoofing - Podvádzanie, kde útočník získa neoprávnený prístup k systému, predstavujúc sa za oprávneného užívateľa.
2. Tampering - Manipulácia, kde útočník zmení údaje alebo systémové komponenty.
3. Repudiation - Popieranie, kde útočník môže poprieť, že vykonal určitú akciu.
4. Information Disclosure - Odhalenie informácií, kde útočník získa neoprávnený prístup k citlivým údajom.
5. Denial of Service - Odmietnutie služby, kde útočník znepřístupní systém alebo službu oprávneným užívateľom.
6. Elevation of Privilege - Zvýšenie oprávnení, kde útočník získa vyššie oprávnenia, než má.

Tieto kategórie pomáhajú systematicky analyzovať a identifikovať potenciálne hrozby v systéme.

Sources:

- p21-02.pdf (Page 27)
- p21-01.pdf (Page 42)
- p21-02.pdf (Page 26)

total time = 7879.72 ms / 638 tokens

CHAT:

STRIDE je metodika používaná v bezpečnosti informácií na identifikáciu a analýzu potenciálnych hrozieb v systéme. STRIDE rozdeľuje hrozby do šiestich kategórií:

1. Spoofing (Podvádzanie): Hrozby, ktoré sa týkajú neoprávneného prístupu k systému alebo údajom pod falošnou identitou. Príkladom môže byť phishing alebo útok na heslá.
2. Tampering (Manipulácia): Hrozby, ktoré sa týkajú neoprávneného zmeny údajov. To môže zahŕňať zmeny v dátach, konfigurácii alebo softvére.
3. Repudiation (Popieranie): Hrozby, ktoré umožňujú útočníkom popierať svoje akcie. To môže byť problém, keď nie je možné overiť, kto vykonal určitú akciu.
4. Information Disclosure (Zverejnenie informácií): Hrozby, ktoré vedú k neoprávnenému zisteniu citlivých informácií. To môže byť dôsledkom slabých opatrení na ochranu dát alebo útokov na systém.
5. Denial of Service (Zamietnutie služby): Hrozby, ktoré spôsobujú, že systém alebo služba nie je dostupná používateľom. To môže byť dôsledkom DDoS útokov alebo iných foriem útoku, ktoré preťažia systém.
6. Elevation of Privilege (Zvýšenie práv): Hrozby, ktoré umožňujú útočníkom získať vyššie úrovne prístupu alebo oprávnení v systéme. To môže byť dôsledkom zraniteľností v softvére alebo konfigurácii.

STRIDE je užitočná pri analýze bezpečnosti, pretože umožňuje systematicky prehľadávať a identifikovať potenciálne hrozby v rôznych častiach systému.

total time = 14576.46 ms / 507 tokens

Ktorý hash algoritmus je považovaný za zlomený?

Podľa informácií z prednášky na stránke 9 v dokumente p21-04.pdf sú hash algoritmy MD5 a SHA1 považované za zlomené. To znamená, že nie sú bezpečné a nemali by sa používať v aplikáciách, kde je potrebná silná bezpečnosť.

Sources:

- p21-10.pdf (Page 31)
- p21-04.pdf (Page 9)
- p21-04.pdf (Page 14)

total time = 2673.91 ms / 524 tokens

CHAT:

Niektoré hash algoritmy sú považované za zlomené, pretože sú zraniteľné voči útokom a nemôžu zaručiť bezpečnosť dát. Medzi najznámejšie zlomené hash algoritmy patria:

1. MD5 (Message Digest Algorithm 5): Tento algoritmus je považovaný za zlomený, pretože je možné vytvoriť kolízie (dva rôzne vstupy, ktoré majú rovnaký hash). MD5 je dnes nepoužiteľný pre bezpečné aplikácie.
2. SHA-1 (Secure Hash Algorithm 1): Aj tento algoritmus je považovaný za zlomený, pretože je možné vytvoriť kolízie. SHA-1 je tiež nepoužiteľný pre bezpečné aplikácie.
3. SHA-256 a SHA-3: Tiež sú považované za bezpečné, ale je potrebné ich správne implementovať a používať. V budúcnosti môžu byť tiež zlomené, ale zatiaľ sú považované za bezpečné.

Pre bezpečné aplikácie je doporučené používať moderné a bezpečné hash algoritmy, ako sú SHA-256, SHA-3 alebo bcrypt, scrypt a Argon2, ktoré sú navrhnuté tak, aby boli odolné voči útokom a kolíziám.

Ak máte konkrétne otázky alebo potrebujete ďalšie informácie, neváhajte sa opýtať.

total time = 10323.21 ms / 361 tokens

Aké zraniteľnosti sú definované v liste OWASP Top 10 pre rok 2021 ?

V liste OWASP Top 10 pre rok 2021 sú definované nasledujúce zraniteľnosti:

1. A01: Broken Access Control - Nedostatočné ovládanie prístupu.
2. A02: Cryptographic Failures - Chyby v šifrovaní.
3. A03: Injection - Injekcia.
4. A04: Insecure Design - Nedostatočný návrh z hľadiska bezpečnosti.
5. A05: Security Misconfiguration - Chybná konfigurácia bezpečnosti.
6. A06: Vulnerable and Outdated Components - Zraniteľné a zastarané komponenty.
7. A07: Identification and Authentication Failures - Chyby v identifikácii a autentifikácii.
8. A08: Software and Data Integrity Failures - Chyby v integrity softvéru a dát.
9. A09: Security Logging and Monitoring Failures - Chyby v bezpečnostnom logovaní a monitorovaní.

10.A10: Server Side Request Forgery (SSRF) - Falsifikácia požiadaviek na stránke servera (SSRF).

Tieto zraniteľnosti sú hlavnými zdrojmi informácií o známych bezpečnostných problémoch a sú dostupné na stránke [OWASP Top 10](#).

Sources:

- o p21-10.pdf (Page 29)
- o p21-10.pdf (Page 27)
- o p21-10.pdf (Page 37)

total time = 10574.65 ms / 696 tokens

CHAT:

OWASP Top 10 pre rok 2021 definuje nasledujúce zraniteľnosti:

1. **Broken Access Control (Porušený kontrolný prístup)** - Neúplný alebo neefektívny kontrolný prístup môže umožniť neoprávneným používateľom prístup k citlivým údajom alebo funkciám.
2. **Cryptographic Failures (Zlyhanie šifrovania)** - Neúčinné alebo neexistujúce šifrovanie môže ohroziť citlivé údaje.
3. **Injection (Injekcia)** - Zraniteľnosti, ktoré umožňujú útočníkom vložiť škodlivý kód do aplikácie, napríklad SQL injekcie alebo OS injekcie.
4. **Insecure Design (Nebezpečný návrh)** - Chyby v návrhu aplikácie, ktoré môžu byť využité útočníkmi.
5. **Security Misconfiguration (Chybná konfigurácia bezpečnosti)** - Neúplná alebo nesprávna konfigurácia bezpečnostných nastavení môže znebezpečiť aplikáciu.
6. **Vulnerable and Outdated Components (Zraniteľné a zastarané komponenty)** - Používanie zastaraných alebo zraniteľných knižníc alebo závislostí môže byť využité útočníkmi.
7. **Identification and Authentication Failures (Zlyhanie identifikácie a autentifikácie)** - Chyby v procese autentifikácie môžu umožniť neoprávnený prístup.
8. **Software and Data Integrity Failures (Zlyhanie integrity softvéru a dát)** - Chyby, ktoré môžu byť využité na zmenu softvéru alebo dát bez povolenia.

9. **Security Logging and Monitoring Failures (Zlyhanie bezpečnostného logovania a monitorovania)** - Neúčinné logovanie a monitorovanie môže znesnadniť detekciu a reakciu na útoky.

10. **Server-Side Request Forgery (SSRF) (Falsifikácia požiadaviek na stránke servera)** - Zraniteľnosti, ktoré

total time = 15700.82 ms / 543 tokens

Čo znamená FAR pri biometrickom overovaní?

FAR pri biometrickom overovaní znamená False Accept Rate, čo sa dá preložiť ako miera falošného prijatia. Je to percentuálne vyjadrenie pravdepodobnosti, že systém biometrického overenia prijíme neoprávnenú osobu ako oprávnenú. V praxi to znamená, že systém sa môže mylne rozhodnúť, že niekto, kto nie je oprávnený, je v skutočnosti oprávnený.

Sources:

- p21-05.pdf (Page 38)
- p21-05.pdf (Page 41)
- p21-05.pdf (Page 31)

total time = 4013.30 ms / 430 tokens

CHAT:

V kontexte biometrického overovania sa skratka **FAR** (False Accept Rate) používa na označenie **úrovne falošného prijatia**. FAR predstavuje percentuálny podiel neoprávnených používateľov, ktorí boli chybné prijatí systémom ako oprávnení. V podstate, FAR meria, ako často systém spraví chybu, keď neoprávnenú osobu prijíma ako oprávnenú.

FAR je dôležitý parametr pri hodnotení bezpečnosti biometrických systémov, pretože vyššia hodnota FAR znamená, že systém je menej bezpečný, pretože neoprávnení používatelia môžu mať vyššiu šancu na obchádzanie bezpečnostných opatrení.

Naopak, **FRR** (False Reject Rate) meria percentuálny podiel oprávnených používateľov, ktorí boli chybné odmietnutí systémom. Ide o ďalší dôležitý parametr, ktorý sa často zvažuje spolu s FAR pri navrhovaní a optimalizácii biometrických systémov.

Pri navrhovaní biometrických systémov je dôležité nájsť rovnováhu medzi FAR a FRR, aby sa minimalizovali chyby prijatia a odmietnutia.

total time = 10181.99 ms / 344 tokens

VSA:

Aká anotácia označuje entitnú triedu v JPA?

V JPA sa entitná trieda označuje anotáciou `@Entity`. Táto anotácia označuje triedu, ktorá bude mapeovaná na tabuľku v relačnej databáze a jej inštancie budú reprezentovať záznamy v tejto tabuľke.

Sources:

- JPA5.pdf (Page 4)
- JPA1.pdf (Page 5)
- JPA1.pdf (Page 4)

total time = 2870.94 ms / 1337 tokens

CHAT:

V JPA (Java Persistence API) sa entitná trieda označuje anotáciou `@Entity`. Táto anotácia informuje JPA, že daná trieda je entita, ktorá môže byť uložená v databáze. Okrem `@Entity` sa často používajú aj ďalšie anotácie, ako napríklad `@Table`, `@Id`, `@Column` a `@GeneratedValue`, ktoré poskytujú ďalšie informácie o mapeovaní medzi triedou a tabuľkou v databáze.

Príklad použitia anotácie `@Entity`:

```
import javax.persistence.Entity;  
  
import javax.persistence.GeneratedValue;  
  
import javax.persistence.GenerationType;  
  
import javax.persistence.Id;
```

`@Entity`

```
public class User {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
private String username;
```

```
private String password;
```

```
// Gettery a settery
```

```
}
```

V tomto príklade trieda User je označená ako entita, ktorá môže byť uložená v databáze. Atribút id je označený ako primárny kľúč a jeho hodnota sa bude automaticky generovať.

total time = 7999.27 ms / 282 tokens

Ako sa v JPA značí dátový člen, ktorý sa nemá mapovať do databázy?

V JPA sa dátový člen, ktorý sa nemá mapovať do databázy, značí anotáciou `@Transient`. Táto anotácia informuje JPA, že daný dátový člen nie je súčasťou persistenčnej štruktúry a nemá byť mapovaný na žiadny stĺpec v tabuľke databázy. Príkladá sa priam na dátový člen v entitnej triede.

Príklad:

```
public class Kniha {
```

```
    private String nazov;
```

```
    private double cena;
```

```
    @Transient
```

```
    private String autor;
```

```
    // gettery a settery
```

```
}
```

V tomto príklade bude dátový člen autor ignorovaný pri mapovaní do databázy.

Sources:

- JPA4.pdf (Page 2)

- JPA1.pdf (Page 7)
- JPA1.pdf (Page 4)

total time = 6179.08 ms / 1405 tokens

CHAT:

V JPA (Java Persistence API) sa dátový člen, ktorý sa nemá mapovať do databázy, označuje pomocou anotácie `@Transient`. Táto anotácia informuje JPA, že daný dátový člen nie je súčasťou trvalej entity a nemá sa mapovať do tabuľky v databáze. Príklad použitia:

```
import javax.persistence.Entity;  
import javax.persistence.Transient;
```

`@Entity`

```
public class Uzivatel {
```

```
    private Long id;
```

```
    private String meno;
```

```
    @Transient
```

```
    private String heslo;
```

```
    // gettery a settery
```

```
}
```

V tomto príklade sa dátový člen `heslo` nebudú mapovať do tabuľky v databáze.

total time = 5548.39 ms / 202 tokens

Aká JPA anotácia sa používa na asociáciu 1:1 ?

Na asociáciu 1:1 sa v JPA (Java Persistence API) používa anotácia `@OneToOne`. Táto anotácia definuje, že medzi dvoma entitami existuje jednoznačná väzba, kde jedna entita je spojená s jednou inou entitou a naopak.

Príklad použitia anotácie @OneToOne:

@Entity

```
public class Uzivatel {
```

```
    @Id
```

```
    private Long id;
```

```
    @OneToOne
```

```
    private Profil profil;
```

```
}
```

@Entity

```
public class Profil {
```

```
    @Id
```

```
    private Long id;
```

```
    @OneToOne(mappedBy = "profil")
```

```
    private Uzivatel uzivatel;
```

```
}
```

V tomto príklade má entita Uzivatel jednoznačnú väzbu s entitou Profil a naopak. Anotácia @OneToOne na strane Uzivatel definuje, že Profil je spojený s Uzivatelom, a anotácia @OneToOne(mappedBy = "profil") na strane Profil definuje, že Uzivatel je spojený s Profilom cez atribut profil.

Sources:

- JPA5.pdf (Page 2)
- JPA4.pdf (Page 1)
- JPA4.pdf (Page 3)

total time = 8745.24 ms / 1188 tokens

CHAT:

V Java Persistence API (JPA) sa na asociáciu 1:1 používa anotácia `@OneToOne`. Táto anotácia sa aplikuje na vlastnosť v entite, ktorá reprezentuje jednu stranu asociácie. Pri asociácii 1:1 je každá entita spojená s jednou inou entitou a naopak.

Príklad použitia `@OneToOne` v JPA:

`@Entity`

```
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String username;  
  
    @OneToOne(mappedBy = "user")  
    private Profile profile;  
  
    // getters and setters  
}
```

`@Entity`

```
public class Profile {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String bio;  
  
    @OneToOne  
    @JoinColumn(name = "user_id")
```

```
private User user;
```

```
// getters and setters
```

```
}
```

V tomto príklade má entita User asociáciu 1:1 s entitou Profile. Vlastnosť profile v entite User je označená anotáciou `@OneToOne` s parametrom `mappedBy`, ktorý určuje, ktorá vlastnosť v entite Profile je zodpovedná za mapovanie asociácie. Vlastnosť user v entite Profile je tiež označená anotáciou `@OneToOne` a `@JoinColumn`, ktorá určuje, ktorý stĺpec v tabuľke Profile bude používaný na uloženie cudzieho kľúča.

Táto konfigurácia umožňuje JPA správne mapovať asociáciu 1:1 medzi entitami User a Profile.

total time = 12167.54 ms / 413 tokens

Aký MIME typ sa používa pre XML dáta v REST službách?

Pre XML dáta v REST službách sa používa MIME typ `application/xml`. Tento MIME typ označuje, že obsah posielať alebo prijímať je vo formáte XML. V kontexte REST služieb sa tento MIME typ používa pri definícii endpointov, ktoré pracujú s XML dátami, ako je to popísané v dokumentoch, ktoré ste zložili. Napríklad, v dokumente `REST2-Jedalen.pdf` na stránke 5 je uvedené, že metóda GET pre zoznam názvov jedál môže vrátiť dáta vo formáte JSON alebo XML, pričom pre XML sa použije práve MIME typ `application/xml`.

Sources:

- `REST2-Jedalen.pdf` (Page 3)
- `REST2-Jedalen.pdf` (Page 5)
- `REST3.pdf` (Page 1)

total time = 5640.92 ms / 1405 tokens

CHAT:

Pre XML dáta v REST službách sa používa MIME typ `application/xml`. Tento MIME typ je štandardne používaný pre prenos XML dokumentov cez sieť. V HTTP hlavičkách sa tento typ obvykle uvádza v políčku `Content-Type`, aby server i klient vedeli, že prenosené dáta sú vo formáte XML.

Príklad HTTP hlavičky pre XML odpoveď:

Content-Type: application/xml

Tento typ je dôležitý pre správnu interpretáciu a spracovanie dát oboma stranami komunikácie.

total time = 5050.48 ms / 178 tokens

Čo v relačných diagramoch označuje FK a PK ?

V relačných diagramoch:

- PK označuje primárny kľúč (Primary Key). Je to unikátny identifikátor, ktorý jednoznačne identifikuje každý záznam v tabuľke. Každá tabuľka má jeden primárny kľúč, ktorý môže byť tvorený jedným alebo viacerými stĺpmi.
- FK označuje cudzí kľúč (Foreign Key). Je to stĺpec alebo skupina stĺpcov v jednej tabuľke, ktorá je primárnym kľúčom v inej tabuľke. Cudzí kľúč zabezpečuje referenčnú integritu medzi tabuľkami, t.j. že hodnoty v cudzom kľúči musia zodpovedať hodnotám v primárnom kľúči tabuľky, na ktorú sa odkazuje.

Sources:

- JPA5.pdf (Page 3)
- JPA1.pdf (Page 2)
- 06-threads.pdf (Page 83)

total time = 7550.98 ms / 1297 tokens

CHAT:

V relačných diagramoch označujú:

- **PK (Primary Key)**: Hlavný kľúč. Je to unikátny identifikátor, ktorý jednoznačne identifikuje každý záznam v tabuľke. Každá tabuľka má jeden hlavný kľúč, ktorý môže byť zložený z jednej alebo viacerých stĺpcov.
- **FK (Foreign Key)**: Cudzí kľúč. Je to stĺpec (alebo skupina stĺpcov) v jednej tabuľke, ktorý odkazuje na hlavný kľúč (PK) v inej tabuľke. Cudzí kľúče slúžia na vytvorenie vzťahov medzi tabuľkami a udržanie integrity dát.

Tieto kľúče sú základné pre udržanie relačných vzťahov medzi tabuľkami v databáze a zabezpečenie integrity dát.

total time = 6915.53 ms / 240 tokens

Kontextové otázky:

MOBV:

Q: Čo je to lovecká odmena ?

Lovecká odmena je typ nepredvídateľnej odmieny, ktorá sa vyskytuje v situáciách, keď nie je jasné, či a kedy sa odmena objaví. Tento typ odmieny je charakteristický tým, že sa dostane len vtedy, keď sa náhodou zistí, že sa odmena nachádza v prostredí.

Lovecká odmena je podobná tomu, ako hľadáte zaujímavý TV program medzi veľkým počtom kanálov, kde nie je isté, či a kedy nájdete niečo relevantné.

Tento koncept je diskutovaný v kontexte, kde by psychologický profil diváka, zostrojený pomocou umelej inteligencie, mohol pomôcť pri nájdení relevantného programu medzi veľkým počtom možností.

Sources:

- moc01.pdf (Page 69)
- moc01.pdf (Page 63)
- moc02.pdf (Page 50)

total time = 6068.30 ms / 663 tokens

CHAT:

Lovecká odmena je pojem, ktorý sa vyskytuje v kontexte bezpečnosti a vývoja softvérových aplikácií, konkrétne v oblasti bezpečného programovania a ochrany pred zraniteľnosťami. Lovecká odmena (angl. bug bounty) je program, ktorý podnetia alebo spoločnosti ponúkajú, aby motivovali bezpečnostných odborníkov (tzv. "loveckov") k hľadaniu a hláseniu zraniteľností v ich softvéri. Za každú nájdenú a overenú zraniteľnosť dostanú loveckovia finančnú alebo inú odmenu.

Tento prístup má viacero výhod:

1. **Zlepšenie bezpečnosti:** Loveckovia môžu nájsť zraniteľnosti, ktoré interní tímy bezpečnosti možno prehliadli.
2. **Komunitné zapojenie:** Podnetia si môžu získať podporu širšej komunity bezpečnostných odborníkov.
3. **Rýchle opravenie chýb:** Zraniteľnosti sa môžu opraviť skôr, než sa stanú problémom.

Lovecké odmeny sú dnes bežnou praxou medzi veľkými technologickými spoločnosťami a podnetiami, ktoré chcú zabezpečiť svoj softvér.

total time = 10447.63 ms / 499 tokens

Q: V čom spočíva etický problém použitia gamifikácie v pracovnom prostredí (napr. Disney)?

Etický problém použitia gamifikácie v pracovnom prostredí, ako je to napríklad u spoločnosti Disney, spočíva v niekoľkých aspektoch:

1. **Autonómnosť:** Gamifikácia môže omezovať autonómnosť zamestnancov. Keďže herné prvky často zahŕňujú odmeny a penalizácie, zamestnanci môžu cítiť tlak, aby sa zúčastnili na určitých aktivitách alebo dosiahli určité ciele, aj keď to neodpovedá ich osobným preferenciám alebo potrebám. To môže viesť k pocitu nevoľnosti a zníženiu motivácie.
2. **Vzťažnosť:** Gamifikácia môže vytvárať nezdravú konkurenciu medzi zamestnancami. Keď sa zamestnanci súperia o odmeny alebo postavenie v rebríčkoch, môže to viesť k negatívnym vzťahom a zníženiu spolupráce. To môže mať negatívny vplyv na celkovú atmosféru v pracovnom prostredí.
3. **Etika odmeny:** Niektoré formy gamifikácie môžu byť eticky znepokojujúce, ak odmeny sú nepriamym spôsobom manipulácie s chovaním zamestnancov. Napríklad, ak sa zamestnanci cítia pod tlakom, aby dosiahli určité ciele len pre to, aby získali odmenu, môže to viesť k neetickému chovaniu alebo zneužívaniu systémov.
4. **Psychologický tlak:** Gamifikácia môže vytvárať psychologický tlak, ktorý môže byť škodlivý pre duševné zdravie zamestnancov. Neustále sledovanie výkonnosti a súperenie s kolegami môže viesť k stresu, únavy a zníženiu celkovej spokojnosti s pracou.

V kontexte spoločnosti Disney, kde je gamifikácia využívaná pre zvyšovanie produktivity a motivácie zamestnancov, je dôležité zvážiť tieto etické aspekty a u

Sources:

- moc02b.pdf (Page 22)
- moc02.pdf (Page 12)
- moc02.pdf (Page 6)

total time = 15855.25 ms / 856 tokens

CHAT:

Gamifikácia v pracovnom prostredí, ako napríklad v spoločnosti Disney, môže priniesť viacero etických problémov. Niektoré z hlavných otázok sú:

1. **Manipulácia zamestnancov:** Gamifikácia môže byť použité na manipuláciu zamestnancov, aby sa viac angažovali alebo dosahovali určité ciele, ktoré môžu byť v rozporu s ich osobnými hodnotami alebo potrebami.
2. **Zvyšovanie stresu a vyhorenia:** Neustále sledovanie a hodnotenie výkonnosti cez gamifikáciu môže zvyšovať stres a viesť k vyhoreniu zamestnancov, keďže sú neustále pod tlakom dosahovať vyššie skóre alebo odmeny.
3. **Nesprávne motivácie:** Gamifikácia môže podporovať krátkodobé motivácie, ktoré sú založené na odmenách a súťaži, namiesto dlhodobých motivácií, ktoré sú založené na osobnom rozvoji a spĺňaní.
4. **Nerovnosti medzi zamestnancami:** Niektorí zamestnanci môžu mať prednosti v gamifikovaných systémoch, čo môže viesť k nerovnostiam a nepriaznivému konkurznému prostrediu.
5. **Zneužitie osobných dát:** Gamifikácia často vyžaduje zber a analýzu osobných dát zamestnancov, čo môže viesť k zneužitiu týchto dát alebo k porušeniu súkromia.
6. **Zamestnanci môžu byť pod tlakom, aby sa zúčastnili:** V niektorých prípadoch môže byť účasť v gamifikovaných programoch povinná, čo môže viesť k nepriaznivému prostrediu, kde sa zamestnanci cítia nútení sa zúčastniť.

Je dôležité, aby spoločnosti, ktoré implementujú gamifikáciu, zohľadnili tieto etické problémy a zabezpečili, že systémy sú navrhnuté tak, aby podporovali

total time = 19163.86 ms / 544 tokens

Q: Ako Foldit využil kolektívnu inteligenciu cez hernú mechaniku?

Foldit využil kolektívnu inteligenciu cez hernú mechaniku tým, že umožnil hráčom spolupracovať na riešení komplexných vedeckých problémov. Hra Foldit je navrhnutá tak, aby hráči mohli manipulovať s molekulami a skládať proteíny do správnych tvarov. Hráči sú motivovaní hernou mechanikou, ktorá je podobná hre Tetris, a sú povzbudzovaní k experimentovaniu a riešeniu úloh, ktoré sú pre vedcov náročné.

V prípade riešenia molekularnej hádanky, ktorá mohla byť kľúčom k nájdeniu lieku proti AIDS, sa týmto problémom zaoberali hráči Folditu. Tím hráčov potreboval len desať dní na to, aby vyriešili enzymovú hádanku, ktorá vedcov zmatala viac ako desať rokov. Týmto spôsobom Foldit ukázal, ako môže kolektívna inteligencia hráčov byť využitá na riešenie skutočných vedeckých problémov.

Sources:

- moc02b.pdf (Page 33)
- moc02b.pdf (Page 18)
- moc02b.pdf (Page 33)

total time = 9019.02 ms / 792 tokens

CHAT:

Foldit je online hra, ktorá využíva kolektívnu inteligenciu hráčov na riešenie komplexných problémov v oblasti bioinformatiky, konkrétne na predpovedanie trojrozmerných štruktúr bielkovín. Hra umožňuje hráčom manipulovať s modelmi bielkovín a dosahovať vyššie skóre tým, že vytvárajú stabilnejšie a funkčnejšie štruktúry. Hráči sú motivovaní hernou mechanikou, ako sú skóre, úrovne a výzvy, čo ich povzbudzuje k experimentovaniu a zdokonaľovaniu svojich riešení.

Kolektívna inteligencia je využívaná tým, že riešenia od jednotlivých hráčov sú zhodnotené a kombinované. Najlepšie riešenia sú potom analyzované a využívané vedcami na univerzite Washington v Seattli, ktorí ich porovnávajú s experimentálnymi dátami a využívajú na výskum. Týmto spôsobom sa dosahuje vyššia úroveň inovácie a presnosti v predpovedaní štruktúr bielkovín, čo má potenciál mať významný vplyv na medicínsky výskum a vývoj nových liekov.

Foldit tak ukázal, ako herná mechanika a kolektívna inteligencia môžu byť efektívne kombinované na riešenie skutočných vedeckých problémov.

total time = 11069.06 ms / 373 tokens

Q: Prečo nový produkt musí byť 9x lepší ?

Nový produkt musí byť 9x lepší kvôli psychologickej nevyváženosti medzi očakávaniami vývojárov a skutočnými potrebami spotrebiteľov. Podľa Johna Gourvillea, spotrebiteľia preceňujú výhody už existujúcich produktov o trojnásobok, zatiaľ čo vývojári preceňujú výhody svojich inovácií tiež o trojnásobok. To vytvára rozpor medzi tým, čo vývojári myslia, že spotrebiteľia chcú, a tým, čo spotrebiteľia skutočne potrebujú, ktorý je devätnásobný (9x).

Tento rozpor je dôležitý pre úspech nového produktu, pretože spotrebiteľia sú často zdržanliví vo vzťahu k novým produktom a radšej zostávajú pri známých a osvedčených riešeniach. Nový produkt musí preto byť výrazne lepší, aby prekonal túto psychologickú bariéru a priťahoval spotrebiteľov.

Sources:

- moc01.pdf (Page 33)
- moc01.pdf (Page 33)
- moc01.pdf (Page 73)

total time = 8952.99 ms / 936 tokens

CHAT:

V kontexte vývoja softvérových aplikácií a inovácií je pravidlo, že nový produkt musí byť 9x lepší, aby sa dostal na trh a získal úspech. Toto pravidlo sa často spomína v súvislosti s inováciami a novými technológiami. Dôvody pre toto pravidlo sú nasledovné:

1. **Prekonanie existujúcich riešení:** Trh je často dominovaný existujúcimi produktmi, ktoré majú už etablovanú základňu používateľov a partnerov. Nový produkt musí byť výrazne lepší, aby prekonal tieto existujúce riešenia a priťahoval používateľov.
2. **Náklady na prechod:** Pre používateľov je často spojené s prechodom na nový produkt určité náklady, ako napríklad čas na naučenie sa nového systému, finančné náklady na implementáciu alebo riziko spojené s prechodom. Nový produkt musí byť dostatočne lepší, aby tieto náklady odživil.
3. **Riziko a neistota:** Nový produkt je často spojený s určitou mierou rizika a neistoty. Používatelia a firmy môžu byť opatrní pri prijímaní nových technológií. Nový produkt musí byť dostatočne atraktívny, aby prekonal túto opatrnosť.
4. **Marketing a propagácia:** Nový produkt musí byť dostatočne inovatívny a atraktívny, aby sa dostal do povedomia potencionálnych používateľov a partnerov. To vyžaduje značné marketingové úsilie a finančné investície.

Toto pravidlo podčiarkuje dôležitosť inovácií a kontinuálneho zlepšovania v oblasti vývoja softvéru a technológií. Nový produkt musí byť ne len trochu lepší, ale výrazne lepší, aby sa dostal na trh a dosiahol úspech

total time = 15213.80 ms / 493 tokens

Q: Čo je „Overjustification effect“ a aký má dopad?

Overjustification effect je psychologický jav, pri ktorom sa odmena za vykonanie úlohy stáva dôležitejšou ako samotná úloha. To znamená, že ľudia môžu začať vykonávať úlohu iba pre odmenu, a nie preto, že im úloha sama o sebe prináša uspokojenie alebo je pre nich významná.

Dopad Overjustification effectu:

1. Zníženie vnútornej motivácie: Keď ľudia sú odmenovaní za vykonanie úlohy, ich vnútorná motivácia (teda motivácia vyplývajúca z vlastného záujmu o úlohu) môže klesnúť. To môže viesť k tomu, že po odobratí odmeny ľudia stráca záujem o úlohu.
2. Závislosť od odmeny: Ľudia môžu začať závisieť od externých odmen, čo môže viesť k tomu, že sa nebudú cítiť motivovaní vykonávať úlohy, ktoré nie sú odmenované.
3. Zmenšenie kreativity a inovácie: Keď ľudia sú motivovaní iba odmenami, môžu strácať kreativitu a schopnosť inovovať, pretože sa sústredia iba na dosiahnutie odmeny, a nie na samotný proces alebo výsledok.

Príklad: V kontexte sociálnych sietí, ako je Twitter, môže overjustification effect viesť k tomu, že ľudia budú hľadať stále viac sledovateľov a lajkov pre potvrdenie a validáciu, a nie preto, že im samotná komunikácia alebo obsah prináša radosť alebo je pre nich významný.

Ak máte ďalšie otázky alebo potrebujete ďalšie informácie, neváhajte sa opýtať.

Sources:

- moc02.pdf (Page 52)
- moc02.pdf (Page 52)
- moc01.pdf (Page 33)

total time = 15900.26 ms / 1010 tokens

CHAT:

„Overjustification effect“ je psychologický fenomén, ktorý popisuje situáciu, keď ľudia, ktorí pôvodne vykonávali určitú činnosť z vnútornej motivácie (napríklad z radosti alebo záľuby), začnú vykonávať túto činnosť z vonkajšej motivácie (napríklad za peniaze alebo iné odmeny). V dôsledku toho môže ich vnútorná motivácia klesnúť alebo úplne zmiznúť.

Dopad tohto efektu môže byť viacerý:

1. **Zníženie vnútornej motivácie:** Keď ľudia začnú dostať vonkajšie odmeny za činnosť, ktorú pôvodne robili z radosti, ich vnútorná motivácia môže klesnúť.
2. **Zníženie kvality práce:** Vnútorná motivácia je často spojená s vyššou kvalitou práce. Keď klesne, môže to viesť k zníženiu kvality vykonávanej práce.
3. **Zmeny v chovaní:** Ľudia môžu začať vykonávať činnosť len preto, že dávajú prednosť vonkajším odmenám, a nie preto, lebo im činnosť sama o sebe prináša radosť alebo uspokojenie.

Tento efekt je dôležitý pre vedcov, manažérov a vzdelávateľov, ktorí sa snažia podporovať dlhodobú motiváciu a záujem o určité činnosti.

total time = 13298.08 ms / 386 tokens

OS:

Q: Čo robí CPU pri prijatí prerušenia?

Pri prijatí prerušenia vykonáva CPU nasledujúce kroky:

1. Uloží aktuálny stav procesoru: CPU uloží aktuálne hodnoty registrov a stav procesoru, aby sa mohol vrátiť k prerušenému programu po obslúžení prerušenia.
2. Zistí zdroj prerušenia: CPU zistí, ktoré zariadenie alebo udalosť vyvolalo prerušenie. To sa robí pomocou mechanizmu, ktorý je v danom systéme implementovaný (napríklad PLIC - Platform-Level Interrupt Controller).
3. Volá príslušnú obslužnú rutinu: Na základe zdroja prerušenia volá CPU príslušnú obslužnú rutinu (interrupt handler). Napríklad, ak je zdrojom prerušenia UART, volá sa rutina `uartintr()`.
4. Obslúži prerušenie: Oslužná rutina vykoná potrebné operácie pre obslúženie prerušenia. Napríklad, ak ide o UART, rutina `uartintr()` vypíše znaky z vstupu a posle ich na výstup spolu s znakmi z vyrovnávacej pamäte.

5. Ukončí obsluhu prerušenia: Po dokončení obsluhu prerušenia volá rutina `plic_complete()`, ktorá ukončí obsluhu prerušenia v PLIC.
6. Obnoví prerušené vykonávanie kódu: CPU obnoví prerušené vykonávanie kódu, vráti sa k uloženému stavu procesora a pokračuje v vykonávaní programu, ktorý bol prerušený.

Ďalej je dôležité spomenúť, že adresa obslužnej rutiny je uložená v registri `$stvec`, ktorý obsahuje buď adresu `kernelvec()` alebo `uservec()` v závislosti od toho, či je prerušenie vyvolané z užívateľského alebo jadrového priestoru. Rovnaký mechanizmus sa používa aj pre výnimky a systémové volania (inštrukcia

Sources:

- o 10-interrupts.pdf (Page 30)
- o 10-interrupts.pdf (Page 40)
- o 10-interrupts.pdf (Page 39)

total time = 16075.14 ms / 808 tokens

CHAT:

Pri prijatí prerušenia (interruptu) vykonáva CPU nasledujúce kroky:

1. **Ukončenie aktuálneho cyklu inštrukcií:** CPU dokončí aktuálnu inštrukciu, aby sa zabránilo nekonzistentnému stavu.
2. **Uloženie stavu procesora:** CPU uloží aktuálny stav procesora (napr. obsah registrov) na zásobník, aby ho mohol neskôr obnoviť.
3. **Zmena režimu:** CPU prejde do režimu prerušenia, ktorý má obvykle vyššiu úroveň prístupu než užívateľský režim.
4. **Získanie adresy obslužnej rutiny:** CPU získa adresu obslužnej rutiny prerušenia z interrupt vector table.
5. **Skočenie na obslužnú rutinu:** CPU preskočí na adresu obslužnej rutiny a začne jej vykonávanie.
6. **Vykonanie obslužnej rutiny:** Obslužná rutina vykoná potrebné operácie, aby obslúžila prerušenie (napr. čítanie dát z periférneho zariadenia).
7. **Obnovenie stavu procesora:** Po dokončení obslužnej rutiny CPU obnoví uložený stav procesora zo zásobníka.
8. **Návrat k pôvodnej činnosti:** CPU pokračuje v vykonávaní pôvodnej činnosti tam, kde sa zastavilo.

Tieto kroky umožňujú efektívne spracovanie prerušenia bez narušenia normálneho toku programu.

total time = 14300.46 ms / 406 tokens

Q: Čo je zámok ?

Zámok (lock) je synchronizačný mechanizmus, ktorý umožňuje implementovať serializáciu, teda radenie operácií za sebou. Zjednodušene povedané, zámok je nástroj, ktorý zabezpečuje, že len jeden vlákno (thread) môže v danom čase pristupovať k kritickej sekcii kódu alebo zdieľanej zdrojovej, čím sa predchádza problémom s konkurenčným prístupom.

Zámok má dva hlavné stavy: odomknutý a zamknutý. Manipulácia so stavmi sa vykonáva pomocou metód `acquire()` (zamknúť) a `release()` (odomknúť). To znamená, že vlákno musí najprv získať zámok (zamknúť ho) pred vstupom do kritickej sekcie a potom ho uvoľniť (odomknúť) po opustení tejto sekcie.

Zámok je zjednodušenou verziou semaforu a je taký, že ten, kto zámok zamkne, musí ho aj odomknúť.

Sources:

- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 60)
- 06-threads.pdf (Page 31)
- 06-threads.pdf (Page 32)

total time = 9459.89 ms / 574 tokens

CHAT:

Zámok (lock) je mechanizmus používaný v operačných systémoch a paralelnom programovaní na zabezpečenie synchronizácie medzi procesmi alebo vláknami. Jeho hlavným úlohou je zabrániť viacerým procesom alebo vláknám pristupovať k zdieľanej zdrojovej súčasne, aby sa zabránilo problémom, ako sú napríklad zmeny dát alebo nekonzistentné stavy.

Zámky môžu byť implementované rôznymi spôsobmi, napríklad ako mutexy (mutual exclusion), semaforey alebo monitory. Každý z týchto mechanizmov má svoje vlastnosti a je vhodný pre rôzne scenáre použitia.

V kontexte operačných systémov a paralelného programovania je správne používanie zámkov kľúčové pre zabezpečenie správnej funkčnosti a bezpečnosti aplikácií.

total time = 7795.63 ms / 225 tokens

Q: Aký je rozdiel medzi Sleeplock a Spinlock?

Rozdiel medzi Sleeplock a Spinlock spočíva hlavne v tom, ako čakajú na uvoľnenie zámku:

1. Sleeplock:

- Pri čakaní na uvoľnenie zámku nevyťažuje CPU. Stav procesu sa označí ako SLEEPING, takže ho plánovač nebude plánovať.
- Vyžaduje kooperáciu s plánovačom operačného systému.
- Vyžaduje viac zdrojov, pretože je potrebné spravovať frontu čakajúcich procesov.

2. Spinlock:

- Pri čakaní na uvoľnenie zámku využíva cyklus, v ktorom neustále testuje dostupnosť zámku.
- Neustále vyťažuje CPU, pretože proces neprechádza do stavu spánku a neumožňuje preplánovanie.
- Je to slabý zámok, ktorý je vhodný pre krátkodobé zámky, kde čas čakania je malý.

V oboch prípadoch metóda `release()` mení stav zámku na voľný/odomknutý. V prípade Sleeplocku metóda `acquire()` zobudí proces, ktorý čaká na zámok.

Sources:

- 06-threads.pdf (Page 36)
- 06-threads.pdf (Page 35)
- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 61)

total time = 10269.26 ms / 787 tokens

CHAT:

Rozdiel medzi Sleeplock a Spinlock spočíva v tom, ako čakajú vlákna na získanie zamknutia (lock).

1. Spinlock:

- **Popis:** Spinlock je mechanizmus, pri ktorom vlákno, ktoré neúspešne pokúša sa získať zamknutie, sa neustále opakuje v cykle (spinning) a znovu skúša získať zamknutie.
- **Výhody:** Rýchle reaguje na uvoľnenie zamknutia, pretože vlákno neprechádza do stavu spánku a nemusí byť znovu aktivované.
- **Nevýhody:** Vyžaduje viac procesorového času, pretože vlákno neustále behá v cykle, čo môže viesť k neefektívnemu využívaniu zdrojov, zvlášť v multiprocesorových systémoch.

2. Sleeplock:

- **Popis:** Sleeplock je mechanizmus, pri ktorom vlákno, ktoré neúspešne pokúša sa získať zamknutie, sa ukladá do stavu spánku a uvoľňuje procesorový čas.
- **Výhody:** Efektívne využíva procesorový čas, pretože vlákno sa nebeží v cykle a nekonzumuje procesorové zdroje.
- **Nevýhody:** Môže byť pomalšie pri reakcii na uvoľnenie zamknutia, pretože vlákno musí byť znovu aktivované, čo môže spôsobovať latenciu.

Výber medzi Spinlock a Sleeplock závisí od konkrétnych požiadaviek aplikácie a charakteristiky systému, na ktorom sa aplikácia spúšťa. V multiprocesorových systémoch sa často používajú kombinácie oboch mechanizmov, aby sa dosiahla optimálna vyváženosť medzi rýchlosťou a efektívnosťou.

total time = 17414.64 ms / 496 tokens

Q: Ako xv6 zabráňuje uviaznutiu pri použití viacerých zámkov?

V operačnom systéme xv6 sa zabráňuje uviaznutiu pri použití viacerých zámkov pomocou techniky známej ako "lock ordering" alebo "zámkové poradie". Toto poradie určuje, v akom poradí sa zámky získavajú a uvoľňujú, aby sa zabránilo situáciám, keď dva alebo viac procesov čakajú na uvoľnenie zámkov, ktoré si vzájomne blokujú (tzv. deadlock).

Konkrétne, xv6 využíva nasledovné princípy:

1. Jednoznačné poradie zámkov: Každý zámok má jednoznačné poradie, v ktorom sa má získať. To znamená, že ak má proces potrebu získať viac zámkov, musí to urobiť vždy v tom istom poradí.

2. Získavanie zámok v poradí: Proces musí získať zámky v presne definovanom poradí. Ak proces potrebuje získať viac zámok, musí získať najprv ten, ktorý má nižšie poradie, a potom ten, ktorý má vyššie poradie.
3. Uvoľňovanie zámok v opačnom poradí: Zámky sa uvoľňujú v opačnom poradí, ako sa získali. To znamená, že ak proces získal zámok A a potom zámok B, musí najprv uvoľniť zámok B a potom zámok A.

Týmto spôsobom sa zabraňuje situáciám, keď dva procesy čakajú na uvoľnenie zámok, ktoré si vzájomne blokujú, čo by mohlo spôsobiť uviaznutie (deadlock).

Ak máte ďalšie otázky alebo potrebujete ďalšie informácie, neváhajte sa opäť ozvať.

Sources:

- o 06-threads.pdf (Page 13)
- o 06-threads.pdf (Page 55)
- o 06-threads.pdf (Page 56)

total time = 15682.77 ms / 866 tokens

CHAT:

V operačnom systéme xv6 sa zabraňuje uviaznutiu pri použití viacerých zámok pomocou techník, ktoré zabraňujú deadlocku. Deadlock môže nastávať, keď dva alebo viac procesov čakajú na zámky, ktoré si vzájomne držia, čo vedie k nekončiacej situácii. Xv6 implementuje niekoľko stratégií na zabránenie deadlocku:

1. **Poradie zámok:** Jedným zo spôsobov, ako zabrániť deadlocku, je vynútiť poradie, v ktorom procesy získajú zámky. Ak všetky procesy získajú zámky v rovnakom poradí, nemôže dôjsť k situácii, kde jeden proces čaka na zámok, ktorý drží iný proces.
2. **Detekcia a obnovenie:** Xv6 môže implementovať mechanizmy na detekciu deadlocku a následné obnovenie systému. To znamená, že systém periodicky kontroluje, či nie je v deadlockovej situácii, a ak áno, uvoľní niektoré zámky a obnoví procesy, aby sa vyhol deadlocku.
3. **Čakanie s časovým obmedzením:** Pri získavaní zámok môže xv6 implementovať časové obmedzenie, po ktorom proces uvoľní zámky, ak nemôže získať všetky potrebné zámky v danom čase. To pomáha zabrániť situáciám, kde procesy čakajú nekonečne dlho.

4. **Hierarchické zámky:** Xv6 môže používať hierarchické zámky, kde zámky sú usporiadané v hierarchii a procesy musia získať zámky v poradí od nižších k vyšším úrovni hierarchie. To zabráňuje cyklickým závislostiam medzi zámkami.

Tieto techniky spolu pomáhajú zabrániť uviaznutiu pri použití viacerých zámok v operačnom systéme xv6.

total time = 16407.27 ms / 532 tokens

Q: Čo sa stane pri výpadku stránky v jadre xv6?

Pri výpadku stránky v jadre xv6 sa stavia na niekoľko kľúčových krokov a reakcií, ktoré sú popísané v prednáške. Vo všeobecnosti, výpadok stránky (page fault) je situácia, keď proces pokúsi sa prístupíť k stránke pamäte, ktorá nie je aktuálne načítaná do fyzickej pamäte.

1. Detekcia výpadku stránky: Keď proces pokúsi sa prístupíť k stránke, ktorá nie je v pamäti, vyvolá sa výpadok stránky. Tento výpadok je detekovaný v hardvérovom zariadení a spôsobí prerušenie, ktoré je zachytené v jadre.
2. Reakcia na výpadok stránky: V jadre xv6 je reakcia na výpadok stránky spracovaná v funkcii `usertrap()`. Ak sa výpadok stránky vyskytne, funkcia `usertrap()` detekuje, že príčina prerušenia (scause) je neočakávaná, čo znamená, že sa jedná o výpadok stránky.
3. Ukončenie procesu: Ak jadro nemôže spracovať výpadok stránky (napríklad, ak stránka nie je platná alebo proces nemá oprávnenie prístupíť k tejto stránke), proces môže byť ukončený. Ukončenie procesu je spracované v funkcii `sys_exit()` v súbore `kernel/sysproc.c`.
4. Spracovanie výpadku stránky: Ak je výpadok stránky platný a proces má oprávnenie prístupíť k tejto stránke, jadro načítne požadovanú stránku z disku do pamäte a obnoví stav procesu, aby mohol pokračovať v jeho výkone.

Tieto kroky sú základné pre spracovanie výpadkov stránky v jadre xv6 a zabezpečujú, že procesy môžu pokračovať v ich výkone aj v prípade, že požadovaná stránka nie je aktuálne v pamäti.

Sources:

- 05-page_faults.pdf (Page 9)
- 06-threads.pdf (Page 79)
- 06-threads.pdf (Page 74)

total time = 18104.01 ms / 990 tokens

CHAT:

Pri výpadku stránky v jadre xv6 sa stane nasledovné:

1. Detekcia chyby: Jadro zistí, že sa pokúša prístupovať k stránke, ktorá nie je v pamäti (tzv. page fault).
2. Zachytenie výnimky: Jadro zachytí túto výnimku a prejde do režimu jadra, aby ju spracovalo.
3. Overenie prístupu: Jadro overí, či proces má oprávnenie prístupovať k tejto stránke. Ak nie, proces sa ukončí s chybou.
4. Načítanie stránky: Ak proces má oprávnenie prístupovať k stránke, jadro pokúsi sa načítať stránku z disku do pamäte. To sa robí pomocou stránkovacej tabuľky, ktorá mapuje virtuálne adresy na fyzické adresy.
5. Aktualizácia stránkovacej tabuľky: Po úspešnom načítaní stránky sa aktualizuje stránkovacia tabuľka, aby odkazovala na správnu fyzickú adresu v pamäti.
6. Opätovné spustenie inštrukcie: Jadro opätovne spustí inštrukciu, ktorá spôsobila výnimku, tentokrát s úspešným prístupom k stránke.

Tento proces je známy ako stránkovanie (paging) a je základným mechanizmom správy virtuálnej pamäte v operačných systémoch, vrátane xv6.

total time = 14467.08 ms / 411 tokens

PPDS:

Čo vyjadruje Amdahlov zákon ?

Amdahlov zákon je teoretický model, ktorý popisuje maximálne možné zrýchlenie výpočtu pri paralelizácii algoritmu. Základné poznatky z Amdahlovho zákona sú nasledovné:

1. Maximálne zrýchlenie: Amdahlov zákon určuje, ako veľké zrýchlenie môžeme očakávať v najlepšom prípade, keď paralelizujeme určitú časť algoritmu. To závisí od podielu častí algoritmu, ktoré môžu byť paralelizované a ktoré zostávajú sériové.
2. Limitácia pridávania výpočtových uzlov: Zákon tiež ukazuje, kedy sa už neoplatí pridávať ďalšie výpočtové uzly (jadrá alebo procesory) do výpočtu. To je preto,

lebo s rastúcim počtom jadier sa zrýchlenie stáva menej efektívnym a investícia do ďalšieho hardvéru sa neoplatí vzhľadom k dosiahnutému zrýchleniu.

Podľa Amdahlovho zákona, s rastúcim počtom jadier zapojených do výpočtu, sa doba vykonania algoritmu približuje dobe vykonania jeho sériovej časti na jednom jadre. To znamená, že paralelizácia nemôže zrýchliť sériovú časť algoritmu, ktorá zostáva nezmenená, bez ohľadu na počet procesorov.

Sources:

- 2025-03.producent konzument citatel zapisovatel.pdf (Page 4)
- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 35)
- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 28)

total time = 11126.17 ms / 910 tokens

CHAT:

Amdahlov zákon je základný zákon v oblasti paralelného programovania, ktorý popisuje teoretický zisk rýchlosti pri paralelizácii úlohy. Zákon bol formulovaný Geneom Amdahlom v roku 1967 a vyjadruje, ako sa celková rýchlosť zlepšuje pri paralelizácii časti úlohy, ktorá je schopná byť paralelizovaná.

Zákon sa dá vyjadriť nasledovne:

$$[S = \frac{1}{(1 - P) + \frac{P}{N}}]$$

kde:

- (S) je zisk rýchlosti (speedup),
- (P) je podiel úlohy, ktorý je možné paralelizovať,
- (N) je počet procesorov alebo vlákien, ktoré sa podieľajú na paralelizácii.

Zákon ukazuje, že maximálny zisk rýchlosti je obmedzený podielom úlohy, ktorý nie je možné paralelizovať. To znamená, že aj keď máme veľký počet procesorov, zisk rýchlosti sa nebudeme môcť neobmedzene zvyšovať, pretože stále bude existovať časová zložka, ktorá musí byť vykonaná sekvenčne.

Amdahlov zákon je užitočný pri analýze a navrhovaní paralelných algoritmov a systémov, pretože poskytuje teoretické obmedzenia toho, ako sa dá úloha zrýchliť paralelizáciou.

total time = 10241.24 ms / 556 tokens

Aké sú výhody a nevýhody použitia distribuovanej pamäte?

Použitie distribuovanej pamäte má svoje výhody a nevýhody, ktoré sú uvedené v prednáške "Úvod do paralelných a distribuovaných výpočtov".

Výhody distribuovanej pamäte:

1. Jednoduché rozširovanie: Distribuovaná pamäť umožňuje ľahké rozširovanie systému pridaním ďalších počítačov do siete.
2. Autonomnosť uzlov: Každý uzol má vlastnú pamäť, čo znamená, že sa nemusíte starať o súbehy pri prístupe k údajom, čo zjednodušuje správu a synchronizáciu.

Nevýhody distribuovanej pamäte:

1. Komunikácia medzi uzlami: Prístup k dátam, ktoré sú uložené na inom uzle, vyžaduje komunikáciu cez sieť, čo môže spôsobovať oneskorenia a znižovať celkovú rýchlosť výpočtu.
2. Složitejšia programácia: Programátor musí explicitne riešiť prístup k dátam, ktoré sú rozptýlené po rôznych uzloch, čo môže byť náročnejšie a chybovitejšie.

Tieto výhody a nevýhody je dôležité zvážiť pri výbere pamäťovej architektúry pre konkrétny typ aplikácie alebo úlohu.

Sources:

- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 23)
- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 22)
- 2025-01.uvod do paralelnych a distribuovanych vypoctov.pdf (Page 18)

total time = 10795.65 ms / 744 tokens

CHAT:

Použitie distribuovanej pamäte má svoje výhody aj nevýhody, ktoré je dobré zvážiť pri návrhu a implementácii paralelných a rozložených systémov.

Výhody distribuovanej pamäte:

1. **Škálovateľnosť:** Distribuovaná pamäť umožňuje ľahšie škálovať systém, pretože môžete pridať viac pamäťových modulov alebo uzlov bez nutnosti zmeny existujúcej architektúry.
2. **Využitie zdrojov:** Každý uzol v distribuovanej pamäti môže mať vlastnú pamäť, čo umožňuje lepšie využitie lokálnych zdrojov a znižuje potrebu prekladania dát medzi uzlami.

3. **Zvýšená odolnosť:** Distribuovaná pamäť môže byť navrhnutá tak, aby bola odolnejšia voči zlyhaniu jednotlivých uzlov, pretože dátové súbory môžu byť replikované na viacerých uzloch.
4. **Flexibilita:** Distribuovaná pamäť umožňuje flexibilné rozmiestnenie dát a úloh, čo môže zlepšiť výkon a efektivitu systému.

Nevýhody distribuovanej pamäte:

1. **Komplexnosť:** Správa distribuovanej pamäte je komplexnejšia ako správa spoločnej pamäte. Je potrebné riešiť problémy s synchronizáciou, konsistenciou dát a komunikáciou medzi uzlami.
2. **Latencia:** Prekladanie dát medzi uzlami môže spôsobovať zvýšenú latenciu, čo môže znížiť celkový výkon systému, zvlášť ak je potrebné často prekladať veľké množstvá dát.
3. **Složitost' programovania:** Programovanie pre distribuovanú pamäť je zložitejšie, pretože je potrebné zohľadniť rozloženie dát a komunikáciu medzi uzlami. To vyžaduje od programátorov hlbšie porozumenie distribuovaným systémom.
4. **Problémy s konsistenciou:** Udržanie konsistencie

total time = 15981.49 ms / 536 tokens

Aký problém môže nastať pri implementácii bariéry a ako ho riešime?

Pri implementácii bariéry môže nastať problém s súperením vlákien pri prístupe k zdieľanému počítadlu. To môže viesť k nekonzistentným stavom a chybám v synchronizácii. Aby sme tento problém riešili, musíme zabezpečiť integritu počítadla pomocou serializácie, čo sa dosahuje pomocou mutexu.

Konkrétne, vlákna musia získať mutex pred tým, ako zmenia hodnotu počítadla, aby sa zabezpečilo, že len jedno vlákno môže meniť počítadlo naraz. Po zmene počítadla a potrebných kontrolách (napríklad, či sa dosiahlo požadovaný počet vlákien) sa mutex uvoľní.

V prípade, že sa použije centralizovaná bariéra, všetka synchronizácia prebieha na jednom pamäťovom mieste, čo môže byť neefektívne pre veľké počty vlákien ($O(N)$). Alternatívou je použitie "software combining tree", ktorý znižuje súperenie procesov pri prístupe k zdieľanému pamäťovému miestu. Týmto spôsobom sa dosahuje lepšia efektivita, predovšetkým v distribuovaných systémoch, kde je potrebné minimalizovať súperenie.

V prípade binárneho kombinačného stromu sa vlákna zjednocujú v pároch, čím sa znižuje počet súperiacich prístupov k zdieľanej pamäti na $O(\log N)$.

Sources:

- 2025-02.mutex multiplex rendezvous bariera.pdf (Page 146)
- 2025-02.mutex multiplex rendezvous bariera.pdf (Page 51)
- 2025-02.mutex multiplex rendezvous bariera.pdf (Page 72)

total time = 11661.33 ms / 894 tokens

CHAT:

Pri implementácii bariéry (barrier) v paralelnom programovaní môže nastať niekoľko problémov. Bariéra je synchronizačný mechanizmus, ktorý zabezpečuje, že všetky vlákna alebo procesy dosiahnu určitý bod v kóde, než pokračujú ďalej. Niektoré z možných problémov a ich riešenia sú:

1. Deadlock (Zamrznutie):

- **Problém:** Ak niektoré vlákna neuspeli dosiahnuť bariéru včas, môže to spôsobiť deadlock, kde všetky vlákna čakajú na seba navzájom.
- **Riešenie:** Zabezpečte, že všetky vlákna sú schopné dosiahnuť bariéru včas. Možno je potrebné implementovať časové limity alebo retry mechanizmy.

2. Performance Overhead (Náklady na výkon):

- **Problém:** Časté používanie bariér môže spôsobiť významné náklady na výkon, pretože vlákna musia čakať na seba navzájom.
- **Riešenie:** Minimalizujte počet bariér a používajte ich len tam, kde je to nevyhnutné. Možno je potrebné prehodnotiť architektúru programu, aby sa zmenšilo potreba na bariéry.

3. Fairness (Správnosť):

- **Problém:** Niektoré implementácie bariér môžu byť nesprávne a umožňovať, že niektoré vlákna dostanú prioritu nad inými.
- **Riešenie:** Použite správnu a ověřenú implementáciu bariéry, ktorá zabezpečuje správnosť a fairness. Možno je potrebné použiť triedené bariéry alebo iné mechanizmy, ktoré zabezpečujú správnu sekvenciu.

4. Scalability (Skalovateľnosť):

- **Problém:** Pri veľkom počte vlákien môže byť implementácia bariéry neefektívna a neškálovateľná.

total time = 15480.99 ms / 538 tokens

Prečo je dvojfázová paralelná redukcia efektívnejšia než jednofázová?

Dvojfázová paralelná redukcia môže byť efektívnejšia než jednofázová v niektorých prípadoch, ale v kontexte MPI (Message Passing Interface) sa jednofázová redukcia (MPI_Reduce) používa z niekoľkých dôvodov:

1. **Cenová optimalita:** V MPI nie je cenová optimalita rozhodujúca, pretože faktor $\log_2 n$ (kde n je počet procesorov) nie je limitujúci. To znamená, že zhoršenie výkonu v dôsledku $\log_2 n$ nie je dostatočne významné na to, aby sa zmenila implementácia na dvojfázovú redukciu.
2. **Použitelnosť:** Jednofázová paralelná redukcia je použiteľná pre ľubovoľný počet procesorov, čo z nej robí flexibilné a univerzálne riešenie.
3. **Škálovateľnosť:** Dvojfázová paralelná redukcia sa nedá ľahko škálovať vzhľadom k počtu procesorov. To znamená, že s narastajúcim počtom procesorov by mohla byť dvojfázová redukcia menej efektívna.
4. **Implementácia:** V prípade potreby dvojfázovej paralelnej redukcie sa musí naprogramovať "manuálne", čo zneprotičňuje jej použitie v mnohých scenároch.

V prednáške sa tiež spomína, že dvojfázová paralelná redukcia môže byť efektívnejšia v kontexte znižovania ceny, ale to sa týka iného kontextu a nie MPI. V MPI je preto preferovaná jednofázová redukcia pre jej flexibilitu a jednoduchosť implementácie.

Sources:

- Prednaska_5_2_25.pdf (Page 22)
- Prednaska_6_24.pdf (Page 17)
- Prednaska_5_2_25.pdf (Page 12)

total time = 12610.71 ms / 879 tokens

CHAT:

Dvojfázová paralelná redukcia je často efektívnejšia než jednofázová z niekoľkých dôvodov:

1. **Zníženie komunikácie:** V jednofázovej redukcii musí každý proces komunikovať so všetkými ostatnými procesmi, čo môže byť zdrojom značného nadbytku

komunikácie. V dvojfázovej redukcii sa komunikácia dekomponuje na dve fázy, čo znižuje počet potrebných komunikácií.

2. **Zlepšenie vyváženia zátiaže:** V dvojfázovej redukcii sa dá lepšie vyvážiť zátiaž medzi procesmi, čo môže zlepšiť celkovú efektívnosť a rýchlosť výpočtu.
3. **Sníženie synchronizácie:** V jednofázovej redukcii je potrebné čakať, kým všetky procesy dokončia svoju časť práce, čo môže spôsobovať neefektívne čakanie. V dvojfázovej redukcii sa synchronizácia dekomponuje na dve fázy, čo môže znížiť čas strávený čakaním.
4. **Lokalita dát:** Dvojfázová redukcia môže lepšie využívať lokalitu dát, keďže každý proces pracuje s menším množstvom dát v každej fáze, čo môže zlepšiť využitie cache a znížiť počet prístupov k pamäti.
5. **Škalovateľnosť:** Dvojfázová redukcia je často lepšie škalovateľná, keďže prírastok počtu procesov nevyžaduje lineárny nárast komunikácie medzi nimi.

Prikladom dvojfázovej redukcie je algoritmus **Rabinovského-Blellocha**, ktorý je často používaný v paralelnom programovaní pre efektívnu redukciiu dát. Tento algoritmus rozdeľuje redukciiu na dve fázy: lokálnu redukciiu a globálnu redukciiu, čo zlepšuje efektívnosť a škalovateľnosť.

total time = 15082.65 ms / 524 tokens

Ako sa dá zabrániť vyhladoveniu v synchronizácii čitateľov a zapisovateľov?

Vyhladovenie (starvation) je problém, ktorý často nastáva pri synchronizácii čitateľov a zapisovateľov, najmä keď sa využíva Lock-Semaphore (LS) mechanizmus. Aby sa zabránilo vyhladoveniu, môže sa použiť riešenie s turniketom.

Riešenie s turniketom

1. **Čitateľa a turniket:** Čitateľa prechádzajú cez turniket, podobne ako na kolotoči. Každý čitateľ musí získať semafor turn predtým, než vstúpi do kritickej sekcie.
2. **Zapisovateľ a blokovanie turniketu:** Zapisovateľ má možnosť zablokovať turniket, takže ďalší čitateľa nemôžu prejsť. To zabezpečí, že zapisovateľ má výlučný prístup k zdroju.

Implementácia

W(): // Procedúra pre zapisovateľa

1) writerLS.lock(turn)

- 2) roomE.wait()
- 3) // Kód zápisu
- 4) roomE.signal()
- 5) writerLS.unlock(turn)

Init(): // Inicializácia

readLS := LS()

roomE := Sem(1)

turn := Sem(1)

R(): // Procedúra pre čitateľa

- 1) turn.wait()
- 2) readLS.lock(roomE)
- 3) turn.signal()
- 4) // Kód čítania
- 5) readLS.unlock(roomE)

Popis kódu

- **W()**: Zapisovateľ získa semafor turn, čaka na semafor roomE, vykoná zápis, uvoľní roomE a uvoľní turn.
- **R()**: Čitateľ získa semafor turn, získa semafor roomE pomocou readLS.lock, uvoľní turn, vykoná čítanie a uvoľní roomE pomocou readLS.unlock.
- **Init()**: Inicializácia semaforov a lockov.

Týmto spôsobom sa zabezpečí, že zapisovateľ má možnosť zabl

Sources:

- 2025-03.producent konzument citatel zapisovatel.pdf (Page 49)
- 2025-03.producent konzument citatel zapisovatel.pdf (Page 52)
- 2025-03.producent konzument citatel zapisovatel.pdf (Page 68)

total time = 15768.54 ms / 927 tokens

CHAT:

Vyhľadovanie v synchronizácii čitateľov a zapisovateľov je problém, ktorý sa môže vyskytnúť v súbežných systémoch, kde viaceré procesy čítajú a zapisujú do spoločného zdroja. Aby sa zabránilo vyhľadovaniu, môžu sa použiť rôzne techniky a mechanizmy:

1. **Mutexy (vzájomné vylúčenie):** Mutex je synchronizačný mechanizmus, ktorý umožňuje iba jednomu procesu pristupovať k zdroju v danom čase. Mutexy sa však nedajú použiť pri čítaní, pretože by to zabránilo viacerým čitateľom čítať súčasne.
2. **Semaforey:** Semaforey sú podobné mutexom, ale môžu mať viac než jednu hodnotu. Semaforey sa môžu použiť na riadenie prístupu k zdroju, ale tiež nemajú ideálnu podporu pre čítanie.
3. **Readers-Writers Lock (zamknutie čitateľov a zapisovateľov):** Tento mechanizmus umožňuje viacerým čitateľom pristupovať k zdroju súčasne, ale zapisovateľom zabraňuje prístup, ak je aspoň jeden čitateľ aktívny. Existujú rôzne varianty readers-writers lock, ktoré sa líšia v tom, ako dávajú prednosť čitateľom alebo zapisovateľom.
4. **Monitory:** Monitory sú vyššie úrovňové synchronizačné mechanizmy, ktoré kombinujú mutexy a podmienkové premenné. Monitor umožňuje definovať kritické sekcie, ktoré môžu byť prístupované iba jedným procesom v danom čase.
5. **Atómové operácie:** Atómové operácie môžu byť použité na zabezpečenie, že čítanie a zápis sa vykonávajú bez prerušenia. To však vyžaduje podporu hardvéru a môže byť náročné na implementáciu.
6. **Transakčné pamäte:** Transakčné pamäte umožňujú procesom čítať a zapisovať do pamäte v transakciách, ktoré sú izolované od ostatných

total time = 15861.94 ms / 544 tokens

UPB:

Aký je rozdiel medzi „session cookie“ a „persistent cookie“?

Rozdiel medzi „session cookie“ a „persistent cookie“ je v tom, ako dlho sa tieto cookies uchovávajú v prehliadači užívateľa.

1. Session Cookie:

- **Trvanie:** Zanikne po zavretí session, teda po zavretí prehliadača.

- **Účel:** Slúžia na uchovanie informácií len počas jednej session, napríklad na uchovanie informácií o prihlásení užívateľa počas jeho návštevy stránky.

2. Persistent Cookie:

- **Trvanie:** Uchováva sa nejakú dobu, ktorá je definovaná v atribúte Expires alebo Max-Age. Po uplynutí tejto doby cookie zanikne.
- **Účel:** Slúžia na uchovanie informácií dlhšie, napríklad na uchovanie údajov o prihlásení užívateľa aj po opätovnom otvorení prehliadača, alebo na uchovanie preferencií užívateľa.

Tieto rozdiely sú dôležité pre bezpečnosť a funkčnosť webových aplikácií, pretože session cookies sú bezpečnejšie pre krátkodobé úlohy, zatiaľ čo persistent cookies môžu byť použité pre dlhodobé uchovanie údajov, ale vyžadujú opatrné správanie, aby sa minimalizovalo riziko zneužitia.

Sources:

- p21-10.pdf (Page 16)
- p21-10.pdf (Page 16)
- p21-10.pdf (Page 22)

total time = 10208.86 ms / 673 tokens

CHAT:

"Session cookie" a "persistent cookie" sú dva typy cookies, ktoré sa používajú na ukladanie informácií o používateľovi na jeho zariadení. Hlavný rozdiel medzi nimi spočíva v tom, ako dlho sa tieto cookies ukládajú a ako sa správajú po ukončení používateľovej relácie.

1. Session Cookie:

- **Trvanie:** Session cookies existujú len počas jednej relácie (session) v prehliadači. Kedykoľvek používateľ ukončí reláciu v prehliadači (napríklad zatvorí okno prehliadača alebo ukončí aplikáciu), session cookie sa automaticky vymaže.
- **Účel:** Používajú sa najčastejšie na ukladanie informácií, ktoré sú potrebné len počas jednej relácie, napríklad údaje o prihlásení alebo obsah košíka v e-shopoch.
- **Bezpečnosť:** Sú považované za bezpečnejšie, pretože sa automaticky vymažu po ukončení relácie.

2. Persistent Cookie:

- **Trvanie:** Persistent cookies majú definovaný časový úsek, počas ktorého budú uložené na zariadení používateľa. Tento časový úsek je nastavený serverom a môže trvať od niekoľkých dní po roky.
- **Účel:** Používajú sa na ukladanie informácií, ktoré by mali pretrvať aj po ukončení relácie, napríklad údaje o prihlásení, preferencie používateľa alebo informácie o návštevách stránky.
- **Bezpečnosť:** Môžu byť považované za menej bezpečné, pretože môžu byť zneužitú na sledovanie používateľov po dlhší časový úsek.

Oba typy cookies majú svoje výhody a nevýhody a ich použitie závisí od konkrétnych potreby a cieľov aplikácie.

total time = 14294.12 ms / 487 tokens

Čo je základným cieľom útoku typu Denial of Service?

Základným cieľom útoku typu Denial of Service (DoS) je znepriístupnenie služby alebo zdroja pre oprávnených používateľov. Útočník sa snaží vyčerpať prostriedky systému, ako je pamäť (disk, RAM), procesorová kapacita (CPU) alebo sieťová kapacita, aby služba bola nedostupná. To môže byť buď dočasné alebo pretrvávajúce. V prípade Distributed Denial of Service (DDoS) útoku sa k tomu využíva veľa uzlov, ktoré presmerujú prevádzku na obeť, čo môže spôsobiť ešte väčšie narušenie.

Sources:

- p21-02.pdf (Page 46)
- p21-02.pdf (Page 45)
- p21-09.pdf (Page 17)\

total time = 5379.92 ms / 608 tokens

CHAT:

Základným cieľom útoku typu Denial of Service (DoS) je vyčerpať zdroje systému, siete alebo služby tak, aby sa stali nedostupné pre legitímnych používateľov. To sa dosahuje tým, že útočník preťaží systém alebo sieť nadmernou množstvom požiadaviek alebo dát, čo vedie k jeho nefungovaniu alebo veľmi pomalému odzvučeniu. V prípade rozšírenej verzie útoku, Distributed Denial of Service (DDoS), sa útoky vykonávajú z viacerých zdrojov, čo zvyšuje ich účinnosť a zložitosť obrany.

Útoky DoS/DDoS môžu mať rôzne formy, napríklad:

1. **Útoky na úroveň aplikácie:** Cielené na konkrétne aplikácie alebo služby, napríklad webové stránky.
2. **Útoky na úroveň siete:** Cielené na vyčerpanie sieťových zdrojov, napríklad šírenie nadmerného množstva paketov.
3. **Útoky na úroveň prístupu:** Cielené na vyčerpanie zdrojov servera, napríklad vyčerpanie počtu súbežných spojení.

Hlavným cieľom týchto útokov je narušiť dostupnosť služieb, čo môže mať vážne dôsledky pre organizácie, ktoré závisia od online prítomnosti a dostupnosti svojich služieb.

total time = 11015.29 ms / 383 tokens

Aký je hlavný rozdiel medzi spoofing a tampering?

Hlavný rozdiel medzi spoofingom a tamperingom spočíva v tom, ako útočník manipuluje s cieľom útoku:

1. Spoofing (Napodobňovanie):
 - Definícia: Útočník sa predstiera, že je niekto alebo niečo iné ako v skutočnosti.
 - Príklady:
 - Napodobnenie procesu alebo súboru.
 - Napodobnenie počítača (napr. ARP/IP/DNS spoofing).
 - Napodobnenie osoby (napr. zmenou e-mail display name).
 - Napodobnenie role (napr. tvárenie sa ako administrátor).
2. Tampering (Manipulácia):
 - Definícia: Útočník zmení dáty alebo metadáta, vrátane pridania alebo zmazania.
 - Príklady:
 - Manipulácia so súbormi (zmena súborov, ktoré vlastní útočník alebo obeť).
 - Manipulácia s pamäťou (zmena kódu alebo dát).
 - Manipulácia so sieťou (presmerovanie toku dát, zmena adresy).

Zhrnutie:

- Spoofing sa zameriava na predstieranie, aby získal dôveru alebo prístup.
- Tampering sa zameriava na zmenu dát alebo toku dát, aby zmenil ich obsah alebo chovanie.

Oba typy útokov môžu byť použité samostatne alebo v kombinácii, aby dosiahli cieľ útočníka.

Sources:

- p21-02.pdf (Page 47)
- p21-02.pdf (Page 42)
- p21-02.pdf (Page 43)\

total time = 12683.50 ms / 872 tokens

CHAT:

Spoofing a tampering sú dva rôzne typy útokov na bezpečnosť, ktoré majú odlišné ciele a metódy.

1. Spoofing:

- **Definícia:** Spoofing je technika, pri ktorej útočník zfalšuje identitu, aby získal neoprávnený prístup k systému alebo informáciám.
- **Príklady:** IP spoofing, DNS spoofing, e-mail spoofing.
- **Cieľ:** Získanie dôvery a prístupu k systému alebo informáciám, ktoré by inak neboli dostupné.

2. Tampering:

- **Definícia:** Tampering je útok, pri ktorom útočník mení alebo zneužíva existujúce údaje alebo systémy, aby dosiahol neoprávnený prístup alebo spôsobil škodu.
- **Príklady:** Zmeny v databáze, modifikácia softvéru, zneužitie autorizovaných prístupov.
- **Cieľ:** Zmeniť alebo zneužiť existujúce údaje alebo systémy, aby sa dosiahol neoprávnený prístup alebo spôsobil škoda.

Hlavný rozdiel:

- **Spoofing** sa zameriava na zfalšovanie identity, aby sa získalo dôveryhodnosť a prístup.

- **Tampering** sa zameriava na zmenu alebo zneužitie existujúcich údajov alebo systémov.

Obe techniky sú nebezpečné a môžu spôsobiť vážne bezpečnostné problémy, ale ich metódy a ciele sú odlišné.

total time = 12523.28 ms / 428 tokens

Čo je to XSS ?

Cross-Site Scripting (XSS) je typ útoku, pri ktorom útočník vloží škodlivý skript do webovej stránky, ktorý sa potom vykoná v prehliadači obete. Tento skript môže získať citlivé informácie, zmeniť obsah stránky alebo vykonávať akcie v mene obete.

Existujú tri hlavné typy XSS útokov:

1. **Reflected XSS:** Skript je dočasne vložený do stránky, napríklad cez parametre URL. Obet musí kliknúť na škodlivý odkaz, aby bol skript vykonaný.
2. **Stored XSS:** Skript je uložený na serveri, napríklad v databáze, a je vykonaný každý raz, keď obet navštíví stránku. To sa často vyskytuje v fórach alebo ankietách.
3. **DOM XSS:** Skript je vytvorený a vykonaný na klientskej strane, napríklad cez AJAX. Nevyžaduje interakciu so serverom.

XSS útoky môžu byť veľmi nebezpečné, pretože umožňujú útočníkom získať prístup k citlivým údajom užívateľov alebo vykonávať akcie v ich mene.

Sources:

- p21-10.pdf (Page 20)
- p21-10.pdf (Page 19)
- p21-10.pdf (Page 3)

total time = 9765.94 ms / 671 tokens

CHAT:

XSS, alebo Cross-Site Scripting, je bezpečnostná chyba, ktorá umožňuje útočníkom vložiť škodlivý skript do webovej stránky, ktorý sa potom vykoná v prehliadači návštevníka stránky. Tento skript môže získať prístup k citlivým údajom, ako sú cookies, session ID, alebo iné informácie uložené v prehliadači. Existujú tri hlavné typy XSS útokov:

1. **Stored XSS (Uložený XSS):** Škodlivý skript je uložený na serveri, napríklad v databáze, a je zobrazený všetkým návštevníkom stránky.
2. **Reflected XSS (Odzrkadlený XSS):** Škodlivý skript je odrazený od servera a vykonaný v prehliadači návštevníka, obvykle cez URL parametre.
3. **DOM-based XSS:** Škodlivý skript manipuluje s DOM (Document Object Model) priamo v prehliadači, bez interakcie s serverom.

Pre ochranu pred XSS útokami je dôležité:

- Validovať a sanitizovať všetky vstupy od používateľov.
- Použiť escaping pri zobrazení údajov z používateľov.
- Použiť bezpečné HTTP hlavičky, ako je Content Security Policy (CSP).
- Použiť bezpečné knižnice a frameworky, ktoré automaticky chránia pred XSS.

XSS je jednou z najčastejších bezpečnostných chýb v webových aplikáciách, preto je dôležité byť oboznámený s týmto typom útoku a implementovať vhodné ochranné opatrenia.

total time = 12804.37 ms / 429 tokens

Aký je hlavný cieľ počítačovej bezpečnosti ?

Hlavný cieľ počítačovej bezpečnosti je chránenie aktív a informácií pred hrozbami a neoprávneným prístupom. To sa dosahuje cez viacero cieľov:

1. **Dôvernosc' (Confidentiality):** Udržiavať neverejnú informáciu dostupnú iba oprávneným osobám.
2. **Celistvosc' (Integrity):** Zabezpečiť, že dáta, softvér a hardvér zostávajú nezmenené, okrem povolených zmien.
3. **Povolenie (Authorization):** Umožniť prístup k výpočtovým prostriedkom iba oprávneným entitám.
4. **Dostupnosť (Availability):** Zabezpečiť, že informácie, služby a výpočtové prostriedky sú prístupné pre oprávnených účastníkov.
5. **Overenie (Authentication):** Potvrdiť, že principal, dáta alebo softvér sú pravé.
6. **Zodpovednosť (Accountability):** Identifikovať principalov zodpovedných za prevedené akcie.

Tieto ciele sa dosahujú prostredníctvom opatrení a protiopatrení, ktoré zamedzujú narušeniu bezpečnostnej politiky, detegujú narušenia a snažia sa obnoviť prevádzku a dáta po útoku.

Sources:

- p21-01.pdf (Page 25)
- p21-01.pdf (Page 16)
- p21-01.pdf (Page 14)

total time = 10539.74 ms / 837 tokens

CHAT:

Hlavný cieľ počítačovej bezpečnosti je chránenie informácií, systémov a komunikácie pred neoprávneným prístupom, útokmi a poškodením. To sa dosahuje prostredníctvom rôznych opatrení a technológií, ktoré zabezpečujú integritu, dôvernosť a dostupnosť dát. Konkrétne to znamená:

1. **Integrita:** Zabezpečenie, že dáty nebudú zmenené alebo poškodené neoprávnenými osobami.
2. **Dôvernosť:** Uistenie sa, že informácie sú dostupné len tým, ktorí majú oprávnenie ich vidieť alebo používať.
3. **Dostupnosť:** Uistenie sa, že autorizovaní používatelia majú nepretržitý prístup k potrebným informáciám a službám.

Počítačová bezpečnosť tiež zahŕňa prevenciu, detekciu a reakciu na bezpečnostné incidenty, ako sú vírusy, hackerské útoky, phishingové útoky a ďalšie hrozby.

total time = 8346.77 ms / 291 tokens

VSA:

Na čo slúži EntityManager v JPA ?

EntityManager v JPA (Java Persistence API) slúži na zabezpečenie pripojenia k databáze a poskytuje funkcie pre prácu s ňou. Hlavné úlohy EntityManageru sú:

1. **Synchronizácia stavu entitných objektov:** EntityManager synchronizuje stav entitných objektov (inštancie entitných tried) s entitami (záznamami) v databáze.
2. **Manipulácia s dátami:** Poskytuje metódy pre čítanie, vytváranie, modifikáciu a mazanie záznamov v databáze.

3. **Transakcie:** Zabezpečuje začiatok, potvrdenie a zrušenie transakcií, čo je dôležité pre udržanie integrity dát.
4. **Persistence context:** Správa kolekcie všetkých objektov, ktoré sú v danom okamihu manažované EntityManagerom. V jednom okamihu môže mať persistence context pre každú databázovú entitu najviac jednu jej inštanciu.

Príklad vytvorenia EntityManagera:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("derbyPU");
```

```
EntityManager em = emf.createEntityManager();
```

EntityManager umožňuje vykonávať rôzne operácie s databázou, napríklad:

- Čítanie údajov: `em.createNativeQuery("select * from kniha", Kniha.class).getResultList();`
- Modifikácia dát: `em.getTransaction().begin(); ... em.getTransaction().commit();`
- Vyhľadávanie podľa kľúča: `em.find(Kniha.class, "Pipi");`
- Vytvorenie nového záznamu: `em.persist(k);`

EntityManager je teda kľúčovým komponentom pri práci s databázou v JPA, ktorý zjednodušuje a abstrahuje mnohé aspekty práce s databázou.

Sources:

- JPA1.pdf (Page 5)
- JPA1.pdf (Page 5)
- JPA2.pdf (Page 1)

total time = 15517.21 ms / 1879 tokens

CHAT:

EntityManager je hlavný rozhranie v Java Persistence API (JPA), ktoré sa používa na správu entít a ich životný cyklus v databáze. Slúži na vykonávanie operácií CRUD (Create, Read, Update, Delete) na entitách, ktoré sú mapované na tabuľky v databáze. Niektoré z hlavných funkcií EntityManager sú:

1. **Vytváranie a odstraňovanie entít:** EntityManager umožňuje vytvárať nové entity a odstraňovať existujúce entity z databázy.
2. **Získavanie entít:** Môže sa použiť na načítanie entít z databázy na základe ich primárneho kľúča alebo iných kritérií.

3. **Aktualizácia entít:** EntityManager umožňuje aktualizovať existujúce entity v databáze.
4. **Spracovanie transakcií:** Správa transakcií je integrálnou časťou EntityManager, ktorý umožňuje začínať, potvrdovať alebo zrušovať transakcie.
5. **Vykonávanie dotazov:** EntityManager podporuje vykonávanie JPQL (Java Persistence Query Language) dotazov, ktoré sú podobné SQL, ale pracujú s entitami a ich vlastnosťami.

Príklad použitia EntityManager:

```
EntityManager em = entityManagerFactory.createEntityManager();
```

```
EntityTransaction transaction = em.getTransaction();
```

```
transaction.begin();
```

```
try {
```

```
    // Vytvorenie novej entity
```

```
    MyEntity entity = new MyEntity();
```

```
    entity.setName("Example");
```

```
    em.persist(entity);
```

```
    // Načítanie entity
```

```
    MyEntity foundEntity = em.find(MyEntity.class, entity.getId());
```

```
    // Aktualizácia entity
```

```
    foundEntity.setName("Updated Example");
```

```
    em.merge(foundEntity);
```

```
    // Odstránenie entity
```

```
    em.remove(foundEntity);
```

```

        transaction.commit();
    } catch (Exception e) {
        if (transaction.isActive()) {
            transaction.rollback();
        }
        e.printStackTrace();
    } finally {
        em.close();
    }
}

```

EntityManager je teda kľúčovým prvkom pri práci s JPA a umožňuje efektívnu správu entít.

total time = 15792.32 ms / 528 tokens

Čo je to kompozícia v kontexte JPA ?

V kontexte JPA (Java Persistence API) je kompozícia špeciálny prípad asociácie OneToMany, v ktorej asociované objekty (komponenty) môžu existovať len ako súčasť celku (kontajnera). To znamená, že životnosť komponentov je závislá od kontajnera. Ak sa kontajner odstráni, odstránia sa aj všetky jeho komponenty.

Príkladom kompozície je nákupný košík, ktorý obsahuje viacero položiek. Položky v košíku nemôžu existovať samostatne; môžu existovať len ako súčasť košíka. Asociácia medzi kontajnerom (nákupný košík) a komponentami (položky) je zvyčajne jednosmerná, kde kontajner pozná svoje komponenty, ale komponenty nemusia poznať kontajner.

V JPA sa takáto kompozícia mapuje pomocou anotácií a správneho nastavenia vzťahov medzi entitami.

Sources:

- JPA5.pdf (Page 1)
- JPA5.pdf (Page 1)
- JPA1.pdf (Page 5)

total time = 8703.75 ms / 1768 tokens

CHAT:

V kontexte Java Persistence API (JPA) je kompozícia vzťah medzi entitami, kde jedna entita (vlastník) obsahuje inú entitu (komponentu) a životný cyklus komponenty je úplne závislý od vlastníka. To znamená, že komponenta nemá samostatný identifikátor (ID) a nemôže byť samostatne persistenčná. Komponenta existuje iba v kontexte vlastníka a je spravovaná spolu s ním.

Príklad kompozície v JPA môže byť adresa, ktorá je súčasťou osoby. Adresa nemá samostatný životný cyklus a je spravovaná spolu s osobou. Ak osoba bude odstránená, bude odstránená aj jej adresa.

V JPA sa kompozícia definuje pomocou anotácie `@Embeddable` pre komponentu a `@Embedded` pre vlastníka. Napríklad:

`@Embeddable`

```
public class Address {  
    private String street;  
    private String city;  
    private String zipCode;  
}
```

`@Entity`

```
public class Person {  
    @Id  
    private Long id;  
    private String name;  
  
    @Embedded  
    private Address address;  
}
```

V tomto príklade je Address komponenta, ktorá je vložená do entity Person. Komponenta Address nemá samostatný identifikátor a je spravovaná spolu s Person.

total time = 9826.13 ms / 334 tokens

Prečo je vhodné označiť stranu mappedBy v obojsmernej asociácii?

V obojsmerných asociáciách je vhodné označiť stranu mappedBy, pretože tento parameter určuje, ktorý dátový člen referencovanej triedy je vlastníkom asociácie. To znamená, že mappedBy definuje inverznú asociáciu a pomáha JPA porozumieť, ktorá strana asociácie je zodpovedná za správu cudzích kľúčov v databáze.

Konkrétne dôvody, prečo je to vhodné:

1. **Zabránenie duplicitným cudzím kľúčom:** Ak by obidve strany asociácie obsahovali anotácie @JoinColumn, JPA by vytvorilo duplicitné cudzie kľúče v databáze, čo by bolo neefektívne a môže spôsobiť chyby.
2. **Zjednodušenie kódu:** Použitie mappedBy zjednodušuje kód, pretože definuje, ktorá strana asociácie je zodpovedná za správu cudzích kľúčov. To znižuje možnosť chýb a zlepšuje čitateľnosť kódu.
3. **Správne vytváranie asociácií:** Pri vytváraní asociácií stačí zadať referencie len na strane, ktorá je vlastníkom asociácie (t.j. strana, ktorá nie je označená mappedBy). To zabezpečuje správne vytvorenie hodnoty cudzích kľúčov v databáze.

Príklad z prednášky:

```
Firma f = new Firma();
```

```
f.setAdresa("Veda");
```

```
Kniha k1 = new Kniha();
```

```
k1.setNazov("1984");
```

```
Kniha k2 = new Kniha();
```

```
k2.setNazov("Animal Farm");
```

```
k1.setVydavatel(f);
```

```
k2.setVydavatel(f);
```

```
List<Kniha> kl = new ArrayList<>();
```

```
kl.add(k1);
```

```
kl.add(k2);
```

```
f.setPublikacie(kl);
```

V tomto príklade je Firma vlastníkom asociácie, pretože Kniha obsahuje referenciu na Firma cez `setV`

Sources:

- JPA4.pdf (Page 2)
- JPA4.pdf (Page 2)
- JPA1.pdf (Page 3)

total time = 16039.86 ms / 1639 tokens

CHAT:

V obojsmernej asociácii v Jave, konkrétne v kontexte JPA (Java Persistence API), je vhodné označiť stranu mappedBy z viacerých dôvodov:

1. **Zabránenie duplicitným záznamom:** Ak by obidve strany asociácie boli označené ako `@OneToMany` alebo `@ManyToOne` bez použitia `mappedBy`, JPA by vytvorila duplicitné záznamy v databáze, čo by mohlo spôsobiť nekonzistencie a zbytočné duplicity.
2. **Zjednodušenie kódu:** Použitie `mappedBy` umožňuje definovať asociáciu iba na jednej strane, čo zjednodušuje kód a zvyšuje jeho čitateľnosť. Strana, ktorá nie je označená `mappedBy`, sa automaticky stane "vlastníkom" asociácie.
3. **Optimalizácia výkonu:** JPA môže lepšie optimalizovať dotazy a operácie s databázou, keď vie, ktorá strana asociácie je "vlastníkom". To môže viesť k rýchlejšim a efektívnejším operáciám s databázou.
4. **Zachovanie integrity dát:** Použitie `mappedBy` pomáha zachovať integrity dát, pretože definuje, ktorá entita je zodpovedná za správu asociácie. To znižuje riziko nekonzistentných dát a chýb.

Príklad obojsmernej asociácie s použitím `mappedBy`:

@Entity

```
public class Autor {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @OneToMany(mappedBy = "autor")
```

```
    private List<Kniha> knihy;
```

```
}
```

@Entity

```
public class Kniha {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "autor_id")
```

```
    private Autor autor;
```

```
}
```

V tomto príklade je asociácia medzi Autorom a Knihou obojsmerná. Strana Autor je označená mappedBy = "autor", čo znamená, že JPA vie, že Kniha je "vlastníkom" asoci

total time = 16093.59 ms / 542 tokens

Ako EntityManager rieši odstránenie objektov v kompozícii?

EntityManager v JPA (Java Persistence API) rieši odstránenie objektov v kompozícii pomocou mechanizmu cascade operations a parametra orphan removal. Keď chceme zabezpečiť, že odstránenie hlavného objektu (napríklad košíka) automaticky odstráni aj

všetky podriadené objekty (napríklad položky v košíku), môžeme použiť CascadeType.REMOVE. Avšak, ak chceme zabezpečiť, že odstránenie položky z košíka (zo zoznamu položiek) odstráni aj záznam z tabuľky POLOZKA, potrebujeme použiť parameter orphanRemoval=true.

Príklad

@Entity

```
public class Kosik {  
    @OneToMany(mappedBy = "kosik", cascade = CascadeType.ALL, orphanRemoval =  
true)  
    private List<Polozka> polozky = new ArrayList<>();  
  
    // ďalšie atribúty a metódy  
}
```

@Entity

```
public class Polozka {  
    @ManyToOne  
    @JoinColumn(name = "kosik_id")  
    private Kosik kosik;  
  
    // ďalšie atribúty a metódy  
}
```

Funkcie

create: Vytvorí košík s niekoľkými položkami. Spustením tejto funkcie môžeme vygenerovať aj tabuľky v databáze.

removeKosik: Odstráni košík aj s všetkými položkami. Dôsledkom je, že sa odstránia aj všetky záznamy v tabuľke POLOZKA, ktoré sú spojené s týmto košíkom.

removePoložka: Odstráni položku z košíka. Ak je nastavený parameter orphanRemoval=true, odstráni sa aj záznam z tabuľky POLOZKA.

Dôležité poznámky

CascadeType.REMOVE: Zabezpečuje, že odstránenie hlavného objektu (košíka) odstráni aj všetky podriadené objekty (polžky).

orphanRemoval=true: Zabezpečuje

Sources:

JPA1.pdf (Page 5)

JPA5.pdf (Page 2)

JPA2.pdf (Page 1)

total time = 16483.00 ms / 1771 tokens

CHAT:

V kontexte Java Persistence API (JPA) a EntityManager, odstránenie objektov v kompozícii vyžaduje zvláštnu pozornosť, pretože kompozícia znamená, že životnosť "dielov" závisí od "celku". Ak chceme odstrániť "celok", musíme tiež odstrániť všetky jeho "diely". Tieto "diely" sú obvykle označené ako @Embeddable alebo sú v relácii "many-to-one" s "celkom".

Krok po kroku:

1. **Nájdenie "celku":** Najprv musíte nájsť objekt "celok", ktorý chcete odstrániť. To sa obvykle robí pomocou metódy find alebo getReference na EntityManager.
2. **Odstránenie "celku":** Použite metódu remove na EntityManager na odstránenie "celku". JPA sa postará o odstránenie všetkých "dielov", ktoré sú v relácii "many-to-one" s "celkom".
3. **Odstránenie "dielov":** Ak "diely" nie sú v relácii "many-to-one", musíte ich odstrániť ručne. To sa robí pomocou metódy remove na EntityManager pre každý "diel".

Príklad:

Predpokladajme, že máme dve entity: Order (celok) a OrderItem (diel).

@Entity

```
public class Order {
```

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)

private List<OrderItem> items = new ArrayList<>();

// getters and setters

}

@Entity

public class OrderItem {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "order_id")

private Order order;

// getters and setters

}

total time = 15875.25 ms / 534 tokens

V čom spočíva výhoda použitia FetchType.LAZY?

Výhoda použitia FetchType.LAZY spočíva v tom, že sa údaje o asociovaných objektoch nenačítajú hneď pri načítaní hlavného objektu, ale až keď sú potrebné. To má niekoľko výhod:

1. **Úspora zdrojov:** Načítanie sa vykoná len vtedy, keď je potrebné, čo môže znížiť nároky na systémové zdroje a zrýchliť inicálnu načítanie dát.
2. **Zlepšenie výkonu:** Ak nie všetky asociované objekty sú potrebné, FetchType.LAZY môže zlepšiť výkon aplikácie, pretože sa nevykonávajú zbytočné dotazy k databáze.
3. **Flexibilita:** Vývojári môžu mať väčšiu kontrolu nad tým, kedy a ako sa asociované objekty načítajú, čo môže byť užitočné v rôznych scenároch aplikácie.

Výhody lazy alokácie v kontexte pamäte sú podobné: alokuje sa menej fyzickej pamäte, ak sa k nej nepristupuje, čo môže znížiť počet výpadkov stránky a zlepšiť celkovú efektivitu systému.

Sources:

- JPA4.pdf (Page 3)
- JPA4.pdf (Page 3)
- 05-page_faults.pdf (Page 27)

total time = 9216.05 ms / 891 tokens

CHAT:

Výhoda použitia FetchType.LAZY v JPA (Java Persistence API) spočíva v optimalizácii prístupu k dátam a zlepšení výkonu aplikácie. Konkrétne:

1. **Znižovanie nákladov na pamäť:** Lazy loading zabezpečuje, že súvisiace entity sa nenačítavajú do pamäte, pokiaľ nie sú explicitne požadované. To znamená, že sa znižuje množstvo dát, ktoré sa nachádzajú v pamäti, čo môže byť zvlášť užitočné pri práci s veľkými množstvami dát.
2. **Zlepšenie výkonu:** Lazy loading môže zlepšiť výkon aplikácie, pretože sa dátam pristupuje len vtedy, keď sú potrebné. To môže znížiť počet dotazov na databázu a zrýchliť odpoveď aplikácie.
3. **Flexibilita:** Lazy loading umožňuje vývojárom flexibilne riadiť, kedy a ako sa dátam pristupuje, čo môže byť užitočné pri optimalizácii aplikácie pre konkrétne scenáre použitia.

4. **Znižovanie zátiaže na sieť:** Pri aplikáciách, ktoré komunikujú s vzdialenou databázou, lazy loading môže znižovať zátiaže na sieť, pretože sa dátam prístupuje len vtedy, keď sú potrebné.

Je však dôležité si uvedomiť, že lazy loading môže spôsobovať problémy, ak nie je správne implementovaný. Napríklad, ak sa pokúsiť prístupovať lazy-loaded entity mimo kontextu persistencia (napríklad v službe alebo kontrolére), môže to spôsobovať výnimky. Preto je dôležité správne nastaviť a spravovať lazy loading v aplikácii.

total time = 13791.36 ms / 470 tokens