PB152 Operačné systémy

Petr Ročkai, preklad Zuzana Baranová

Časť A: Úvodné informácie

Dívate sa na poznámky k prednáškam pre PB152. Všetko, čo potrebujete vedieť na úspešné absolvovanie predmetu (mimo seminára, ktorý tvorí samostatný predmet - PB152cv) je obsiahnuté v tomto dokumente. Prednášky pokrývajú rovnaké materiály v inom formáte (slidy, ktoré vidíte v tomto dokumente sú rovnaké slidy, ktoré sú použité v prednáškach).

Testy a skúška

3

- 3 krátke priebežné testy (každý na 4 prednášky)
  - 2 opakovacie otázky na prednášku (8 dokopy)
  - musíte úspešne urobiť 2 z 3 testov (inak X)
- skúška na záver semestra
  - časť 1: opakovacie otázky
  - časť 2: hĺbkové porozumenie
- všetky testy sú jedno-priechodové (k odpovediam sa nedá vracať)

Na úspešné absolvovanie predmetu potrebujete zložiť aspoň 2 z 3 priebežných testov, ktoré budú prístupné v IS. Každý z týchto testov bude pokrývať 4 prednášky - na každej prednáške dostanete 2 opakovacie otázky (ktoré budú uvedené na konci každej prednášky). Môžete urobiť jednu chybu (tzn. musíte odpovedať na aspoň 7 z 8 otázok správne, aby sa vám test počítal ako úspešný). Ak neuspejete v 2 testoch z 3, dostanete známku X.
Ak úspešne zložíte 2 (alebo všetky 3) testy, môžete pristúpiť ku skúške, ktorá bude mať dve časti - prvá bude rovnaká ako priebežné testy, akurát bude obsahovať 12 otázok, jednu z každej prednášky. Takisto môžete urobiť 1 chybu. Ak budete úspešní, pokračujete k druhej časti, ktorá je zameraná na hĺbkové pochopenie látky, a ktorá rozhodne o vašej výslednej známke. Musíte úspešne zložiť obe časti skúšky naraz - ak neuspejete v jednej, musíte znova skladať obe.
Všetky testy (tzn. priebežné testy a obe časti skúšky) budú prebiehať v IS-e a budú sa vám zobrazovať po jednej otázke, na ktorú musíte odpovedať, než sa môžete pozrieť na ďalšiu otázku. K odovzdaným otázkam sa nebudete môcť vrátiť. Cvičné testy budú tiež nastavené týmto spôsobom, aby ste si mohli vyskúšať, ako si najlepšie zorganizovať čas.

Priebežné testy

4

- 8 otázok, 15 minút, najviac 1 chyba
- cvičný test 1 týždeň pred tým, neobmedzený počet pokusov
- kedy – medzi 16:00 a 16:30 v nasledujúce dni:
  - 9. apríl
  - 7. máj
  - 4. jún

Priebežné testy budú prebiehať v piatky poobede. Budete mať 4 týždne na naštudovanie si príslušných 4 prednášok, potom 1 týždeň na opakovanie (s pomocou cvičného testu, ak chcete). Cvičný test bude prístupný počnúc predchádzajúcim piatkom, až do momentu kedy začína priebežný test. Cvičný test si môžete skúšať koľkokrát chcete.
Dostanete pripomienku emailom o každom priebežnom teste jeden týždeň pred tým, než sa bude konať (v rovnakom čase, kedy sa zverejní cvičný test) a ráno v deň konania testu.

Skúška

5

- časť 1: 12 opakovacích otázok, 20 minút
  - 10 alebo menej = F, 11+ = pokračujete na časť 2
- časť 2: 12 otázok, 90 minút
  - určte pravdivosť zložitejších tvrdení
  - +1 / -0.5 bodov za otázku
  - 6+ = E, 7+ = D, 8+ = C, 9+ = B, 10+ = A
- cvičné testy v máji, neobmedzené množstvo pokusov

V máji bude sprístupnená cvičná verzia oboch častí. Obe si budete môcť vyskúšať ľubovoľne krát, avšak výber otázok (a odpovedí) bude obmedzený, v porovnaní so skutočnou skúškou.
V druhej časti bude každá otázka zložená zo 4 tvrdení (krátky odstavec) o nejakom aspekte operačných systémov. Existujú dva možné scenáre: buď

- 3 tvrdenia sú nepravdivé a 1 je pravdivé (v tomto prípade vyberte pravdivé tvrdenie), alebo
- 3 tvrdenia sú pravdivé a 1 je nepravdivé (v tomto prípade vyberte nepravdivé tvrdenie).

Všetky testy a skúšky budú v češtine, pričom technické výrazy budú tiež zmienené v angličtine (v zátvorke). Ak chcete skladať skúšku v angličtine, kontaktujte ma najneskôr do 19. marca.

Semináre

6

- samostatný, voliteľný kurz (kód PB152cv)
- venuje sa operačným systémom z praktického hľadiska
- vyskúšate si veci, o ktorých tu budeme rozprávať
- ponúka dodatočnú prax s programovaním v C

Seminár poskytuje dobrý spôsob získania nejakých praktických skúseností s operačnými systémami. Zameriava sa hlavne na interakciu programov so službami OS, ale tiež pokryje problémy na užívateľskej úrovni, ako virtualizácia, inštalácia OS a písanie shell-ových skriptov.

Študijné materiály

7

- poznámky k prednáškam sú hlavným učebným textom
  - obsahujú slidy použité v prednáške
  - zväčša samostatné s minimom závislostí
  - anglická verzia je už prístupná v IS-e
  - preložené kapitoly budú zverejnené týždenne
- nahrávky prednášok
  - z behu 2019, ako podporný materiál

Čítate poznámky k prednáškam ku kurzu PB152 Operačné Systémy. Toto je váš hlavný učebný zdroj; je založený na prednáškových slidoch, ale s dodatočnými detailmi, ktoré sa do formátu slidov nevošli. Tieto poznámky by mali byť samostatné, v zmysle že sa spoliehajú iba na znalosti, ktoré máte z iných kurzov, ako PB150 Architektury výpočetních systémů (alebo PB151) alebo PB071 Principy nízkoúrovňového programování. Podobne, ovládať témy obsiahnuté v týchto poznámkach je

dostatočné na úspešné absolvovanie skúšky.

Knihy

8

- existuje niekoľko dobrých kníh o OS
- je dobrý nápad si ich prečítať

- A. Tanenbaum: Modern Operating Systems
- A. Silberschatz et al.: Operating System Concepts
- L. Skočovský: Principy a problémy OS UNIX
- W. Stallings: Operating Systems, Internals and Design
- mnoho ďalších, kľudne sa poobzerajte po iných

Vyššie zmienené knihy zvyčajne pokrývajú výrazne viac materiálu, než je možné obsiahnuť v jednosemestrálnom kurze. Štúdium operačných systémov je však veľmi dôležité pre mnoho pod-odvetví informatiky, a tiež pre mnoho programovacích disciplín. Akýkoľvek čas naviac, ktorý na tejto téme strávite pravdepodobne nebude na škodu.

Témy

9

1. Anatómia OS
2. Systémové knižnice a API
3. Jadro (Kernel)
4. Súborové systémy
5. Základné zdroje a multiplexing
6. Súbežnosť a zamykanie

V prvej polovici semestra sa budeme zaoberať základnými komponentami a abstrakciami, ktoré sú prítomné v univerzálnych (general-purpose) operačných systémoch. Prvá prednáška poskytne iba základný prehľad celého operačného systému a pokúsi sa načrtnúť, ako do seba jednotlivé časti zapadajú. Druhou prednáškou pokryjeme základné programovacie rozhrania, ktoré OS poskytuje, sprostredkovávané prevažne systémovými knižnicami.

Témy (pokračovanie)

10

7. Ovládače zariadení
8. Sieťové vrstvy (Network Stack)
9. Interprety príkazov & užívateľské rozhrania
10. Užívatelia a práva
11. Virtualizácia & kontajnery
12. Operačné systémy na zvláštne účely

V druhej polovici semestra začneme ovládačmi zariadení, ktoré všeobecne tvoria dôležitú časť operačných systémov, keďže sprostredkovávajú komunikáciu medzi aplikačným softvérom a hardvérovými perifériami pripojenými k počítaču. Podobne, vrstvy sieťového rozhrania umožňujú programom komunikovať s ďalšími počítačmi (a so softvérom, ktorý beží na týchto ďalších počítačoch), ktoré sú pripojené k počítačovej sieti.

Súvisiace kurzy

11

- PB150/PB151 Architektury výpočetních systémů
- PB153 Operační systémy a jejich rozhraní
- PA150 Principy operačních systémů
- PV062 Organizace souborů
- PB071 Principy nízkoúrovňového programování
- PB173 Tematicky zaměřený vývoj aplikací v jazyce C/C++

Existuje niekoľko kurzov, ktoré sa buď prelínajú s týmto, tvoria prerekvizity, ktoré vyučujú znalosti potrebné pre tento kurz, či kurzy, ktoré rozširujú, čo sa tu naučíte. Zoznam vyššie je neúplný. Kurz PB153 je alternatívou k tomuto kurzu. Predpokladá sa, že väčšina študentov bude mať PB071 paralelne s týmto kurzom, napriek tomu, že znalosti C nebudú potrebné pre teóriu, ktorú preberieme. Základy jazyka C však budú nevyhnutné pre voliteľný seminár (PB152cv).

Organizácia semestra

12

- všeobecne, jedna prednáška = jedna téma
- s najväčšou pravdepodobnosťou bude 13 prednášok
-
- prednáška bude opakovacia
- 3 priebežné testy: 2.4., 30.4., 28.5.

Časť B: Prehľad semestra

Táto sekcia poskytuje vysokoúrovňový prehľad tém, ktoré budú pokryté v jednotlivých prednáškach. Berte to ako rozšírený obsah alebo zbierku abstraktov, jeden pre každú z nadchádzajúcich prednášok.

2 Systémové knižnice a API

14

- POSIX: Portable Operating System Interface
- UNIX: (takmer) všetko je súbor
- najmenší spoločný základ programov: C
- z pohľadu užívateľa: objekty, archívy, zdieľané knižnice
- prekladač, linker

Systémové knižnice a ich rozhrania (API) poskytujú najpriamejší prístup k službám operačného systému. V druhej prednáške si rozoberieme ako programy pristupujú k týmto službám a ako systémové knižnice vstupujú do programovacieho jazyka C. Tiež si rozoberieme základné artefakty, ktoré tvoria programy: objektové súbory, archívy, zdieľané knižnice a ako tieto vznikajú: ako sa zo súboru obsahujúceho zdrojový kód jazyka C stane nakoniec spustiteľný súbor skrze preklad (kompiláciu) a linkovanie.
V priebehu tejto prednášky budeme používať ako náš preferovaný zdroj pre čerpanie príkladov štandard POSIX, keďže ide o najpoužívanejšie systémové rozhranie. Navyše preň existuje dostatok dokumentácie a iných zdrojov, ako online tak offline.

3 Jadro (Kernel)

15

- privilegovaný režim CPU
- proces bootovania
- vynútenie hraníc
- typy jadra: mikro, mono, exo, ...
- systémové volania

V tretej prednáške sa sústredíme na jadro (kernel), pravdepodobne najdôležitejšiu (a často najkomplikovanejšiu) časť operačného systému. Začneme pekne na začiatku, procesom bootovania: ako sa jadro načíta do pamäte, inicializuje hardvér a spustí súčasti užívateľského priestoru (teda všetko, čo nie je jadro) operačného systému.
Potom si povieme o vynútení hraníc: ako jadro kontroluje užívateľské procesy, aby nezasahovali do seba navzájom a do hardvérových zariadení. Naznačíme, ako toto vynútenie umožňuje zdieľanie počítača niekoľkými užívateľmi, bez toho aby sa navzájom narušovali (alebo aspoň aby narušenie bolo minimálne).
Ďalšou zaujímavou témou bude ako je také jadro navrhnuté a čo je a nie je súčasťou jadra. Rozoberieme niektoré kompromisy jednotlivých typov jadra, hlavne čo sa týka bezpečnosti a korektnosti vo vzťahu k výkonnosti.
Nakoniec sa pozrieme na mechanizmus systémových volaní, čo je spôsob, akým komunikuje užívateľský priestor s jadrom (kernelom), a ktorým posiela požiadavky na rôzne nízkoúrovňové systémové služby.

4 Súborové systémy

16

- prečo a ako
- abstrakcia nad zdieľaným blokovým úložiskom
- hierarchia adresárov
- všetko je súbor
- i-uzly, adresáre, pevné (hard) & symbolické/mäkké (soft) odkazy

Nasledujú súborové systémy, ako rozšírená abstrakcia nad stálym blokovým úložiskom, ktoré nám poskytujú hardvérové úložné zariadenia. V prvom rade sa zamyslíme, prečo sú súborové systémy potrebné a prečo sú tak všadeprítomné v operačných systémoch, a pozrieme sa ako fungujú vnútri. Konkrétne sa zameriame na tradičný UNIX-ový súborový systém, ktorý nám poskytne dôležitý vhľad do architektúry operačného systému ako celku, a do dôležitých aspektov sémantiky POSIX-ových súborov.

5 Základné zdroje a multiplexing

17

- virtuálna pamäť, procesy
- zdieľanie procesorov & plánovanie
- procesy vs vlákna
- prerušenia, hodiny

Jednou zo základných úloh operačného systému je správa rôznych zdrojov, počnúc najzákladnejšími: jadrami procesora a pamäťou RAM. Keďže tieto prostriedky sú veľmi dôležité pre všetky procesy, budeme im venovať celú prednášku. Pozrieme sa na základné jednotky pridelenia zdrojov: vlákna pre procesor a procesy pre pamäť. Tiež sa budeme venovať mechanizmom, ktoré kernel používa na pridelenie a ochranu daných zdrojov, konkrétne virtuálnu pamäť a plánovač úloh (scheduler).

6 Súbežnosť a zamykanie

18

- komunikácia medzi procesmi
- prístup k zdieľaným zdrojom
- vzájomné vylúčenie
- uviaznutie (deadlock) a prevencia uviaznutia

Plánovanie a rozdeľovanie času procesora úzko súvisí s ďalšou dôležitou témou, ktorá je ústredná pri návrhu operačných systémov: súbežnosť. Túto tému prejdeme na pomerne vysokej, všeobecnej úrovni, keďže detaily sú často komplikované a hlavne späté s danou architektúrou a vyžadujú dôkladné pochopenie hardvéru (SMP, hierarchia cache pamätí), aj jadra.

7 Ovládače zariadení

19

- ovládače na úrovni užívateľa vs jadra
- prerušenia ap.
- GPU
- PCI ap.
- blokové úložisko
- sieťové zariadenia, wifi
- USB
- bluetooth

Ďalšou z centrálnych úloh operačného systému je sprostredkovať prístup k hardvérovým zariadeniam. Časť kódu, ktorá umožňuje tento prístup k hardvéru sa zaoberá najmä softvérovými rozhraniami a API – tomuto sa tiež hovorí 'abstrakcia hardvéru'. Aby však táto abstrakcia mohla fungovať, často je potrebné množstvo 'lepidla', ktoré je špecifické pre konkrétne zariadenie (alebo aspoň triedu daného zariadenia) – lepidla tiež známeho ako ovládače zariadení.
Jednou z dôležitých otázok bude súhra medzi ochranou na úrovni procesora a priamym prístupom k hardvéru a jej význam pre ovládače. Uvidíme, že pri niektorých (ale nie všetkých) typoch hardvéru môžu komunikáciu medzi hardvérom a vyššími úrovňami systému (vrstva abstrakcie hardvéru, aplikačný softvér, atď.) rozumne sprostredkovávať iba privilegované programy (v jadre alebo jeho blízkosti).

8 Sieťové vrstvy (Network Stack)

20

- TCP/IP
- preklad mena na IP adresu (name resolution)
- API soketov
- firewall a filtrovanie paketov
- sieťové súborové systémy

Napriek tomu, že budete mať samostatný kurz zameraný na siete, strávime jednu z prednášok štúdiom sietí: sieťová komunikácia je neoddeliteľnou súčasťou moderných operačných systémov a rozhodnutia týkajúce sa sietí často ovplyvňujú ďalšie časti systému. Pozrieme sa na všadeprítomné vrstvy TCP/IP, ako táto architektúra zapadá do operačného systému, a ktoré rozhrania (API) môžu aplikácie používať, aby dokázali využiť sieťové služby. Tiež si ukážeme ďalšiu funkcionalitu úzko spätú so sieťami, ktorá je často hlboko integrovaná v operačných systémoch: filtrovanie paketov a sieťové súborové systémy.

9 Interprety príkazov & užívateľské rozhrania

21

- interaktívne systémy
- v minulosti: konzoly a terminály
- textové terminály, RS-232
- bash a iné Bourne shell-y, POSIX
- grafické: X11, Wayland, OS X, Windows, Android, iOS

Ďalšia prednáška je zameraná na interakciu človeka s počítačom, čo je nepochybne podstatnou súčasťou dojmu z používania počítača a tým aj podstatnou časťou väčšiny univerzálnych (general-purpose) operačných systémov. Aj počítače, ktoré nekomunikujú s človekom priamo (fyzicky), obvykle poskytujú nejakú formu rozhrania, väčšinou sprostredkovanú sieťovo.
Najprv sa pozrieme na 'tradičné' textové rozhrania, ktoré sú stále hojne využívané medzi systémovými a sieťovými inžiniermi a programátormi, ale tiež sa do určitej miery budeme venovať grafickej podpore v moderných zariadeniach (nevynímajúc smartfóny).

10 Užívatelia a práva

22

- viac-užívateľské (multi-user) systémy
- izolovanie, vlastníctvo
- práva v súborových systémoch
- čo sme schopní dosiahnuť

Existujú dva podstatné prípady použitia počítačov (a tým operačných systémov), kde je dôležitá vysokoúrovňová kontrola prístupu a správa práv: prvý, keď je jeden počítač zdieľaný viacerými užívateľmi (čo je ten typickejší prípad), ale v modernom svete tiež prípad, keď na svojich zariadeniach spúšťame nedôveryhodné alebo čiastočne dôveryhodné programy (práva pre aplikácie na smartfónoch, webové stránky, ktoré spúšťajú javascript na vašom laptope atď.).

11 Virtualizácia & kontajnery

23

- multiplexovanie zdrojov
- izolovanie
- viacero jadier (kernelov) na jednom systéme
- hypervízory typu 1 a typu 2
- `virtio`

Počítač, spolu so svojím operačným systémom, je prirodzenou štandardnou 'jednotkou' (unit) výpočtových zdrojov – vhodne zapuzdruje zdroje samotné, ako aj softvérové vrstvy a konfiguráciu systému. Bohužiaľ, počítače – ako fyzické zariadenia – sú nepraktické a neskladné: treba ich zaobstarať, uložiť na police v klimatizovaných miestnostiach, pripojiť k zdroju napájania, k sebe navzájom a do väčšej siete. Ich fyzické komponenty sú náchylné na opotrebovanie a poruchy a musia byť pravidelne vymieňané a opravované.
Virtualizácia nám umožňuje oddeliť logické aspekty počítača – nainštalovaný softvér, dátové úložisko a konfiguráciu – od fyzickej krabice. Tým môžeme zlepšiť využitie hardvéru, oddeliť údržbu hardvéru od softvérových prvkov a všeobecne si zjednodušiť život (väčšinou).
V tejto prednáške sa bližšie pozrieme na moderné virtuálne stroje založené na báze hypervízorov a na to, ako sú implementované v súčasnej generácii operačných systémov.

12 Operačné systémy na zvláštne účely

24

- univerzálne (general-purpose) vs špecializované (special-purpose)
- embedded systémy
- real-time systémy
- high-assurance systémy - zamerané na vysokú bezpečnosť a spoľahlivosť (seL4)

V priebehu kurzu sa väčšinou budeme baviť o univerzálnych (tzv. general-purpose) operačných systémoch: používaných v osobných počítačoch a serveroch. Posledná prednáška bude venovaná špecializovaným systémom, ktoré sa používajú na zvláštne účely: systémy vyvinuté pre práčky, satelity, či pre vozidlá vysielané na Mars. Zbežne sa tiež pozrieme na systémy zamerané na vysokú spoľahlivosť (high-assurance systems), ktorých cieľom je extrémne vysoká spoľahlivosť a/alebo bezpečnosť.

Časť 1: Anatómia OS

V prvej prednáške si najprv položíme otázku "Čo je operačný systém", a odpovieme si sériou krátkych, aj keď vo veľkej miere neuspokojivých, odpovedí. Keďže ide o komplexný systém, operačný systém je zložený z množstva komponent. Každá z týchto komponent je samostatne popísateľná jednoduchšie, než celý operačný systém, a preto sa pokúsime porozumieť operačnému systému ako súboru dielčích častí.

Obsah prednášky

26

1. Komponenty
2. Rozhrania
3. Klasifikácia

Po rozbore definície operačného systému sa pozrieme detailnejšie na jeho komponenty a na rozhrania medzi týmito komponentami. Nakoniec sa pozrieme na klasifikáciu operačných systémov, čo je ďalší uhol pohľadu, ktorý nám môže pomôcť s pochopením toho, čo je operačný systém.

Čo je operačný systém?

27

- softvér, ktorý sa stará o fungovanie hardvéru
- a zjednodušuje písanie ďalšieho softvéru

Tiež

- univerzálne pomenovanie pre nízkoúrovňový softvér
- vrstva abstrakcie nad daným zariadením
- hranice nie sú vždy zrejmé

Náš prvý (veľmi približný) pokus o definíciu OS sa týka jeho vzťahu k hardvéru. Keďže sa nachádza medzi hardvérom a zvyškom softvéru, v istom zmysle je zodpovedný za fungovanie hardvéru. Moderný hardvér je zriedka sám osebe schopný byť užitočný. Musí byť naprogramovaný, pričom základná vrstva týchto programov pochádza z operač-

ného systému.

Čo nie je (súčasťou) OS?

28

- firmvér: (veľmi) nízkoúrovňový softvér
  - výrazne viac viazaný na hardvér než OS
  - často beží na pomocných procesoroch (auxiliary processors)
- aplikačný softvér
  - beží nad operačným systémom
  - to, kvôli čomu si obstarávate počítač
  - napr. hry, tabuľkové kalkulátory, úprava fotografií, ...

Jedným z prístupov pochopenia toho, čo operačný systém je, môže byť pozrieť sa na príbuzné veci, ktoré operačný systém nie sú, a nie sú ani žiadnou jeho časťou. Pod operačným systémom sa nachádza ešte jedna v podstate softvérová vrstva, prevažne známa ako firmvér. V typickom počítači je prítomných veľa častí firmvéru, ale väčšina je spúšťaná na pomocných (auxiliary) procesoroch – napr. procesoroch vo WiFi karte, v grafickom subsystéme, v pevnom disku, ap. Na rozdiel od toho, operačný systém beží na hlavnom procesore. Existuje jedna časť firmvéru, ktorá typicky beží na hlavnom CPU: na starších systémoch sa jej hovorilo BIOS, na moderných systémoch je známa proste ako "firmvér" (the firmware).
Opačným smerom, teda nad operačným systémom, sa nachádza množstvo aplikačného softvéru. Napriek tomu, že časť tohto typu softvéru môže byť priamo pribalená (bundled) k operačnému systému, nie je, prísne vzaté, jeho súčasťou. Aplikačný softvér sú programy, ktoré používate na nejakú činnosť, ako textové editory a procesory, programovacie vývojové prostredia (tzv. IDE – integrated development environment), počítačové hry alebo webové aplikácie (Facebook). A tak ďalej.

Čo robí operačný systém?

29

- interaguje s užívateľom
- spravuje a prideľuje (multiplexuje) hardvér
- spravuje ďalší softvér
- organizuje a spravuje dáta
- poskytuje služby iným programom
- vynucuje bezpečnosť

Úlohy a povinnosti operačného systému sú pomerne rôznorodé. Na jednu stranu sa stará o základnú interakciu s užívateľom: či už cez interpret príkazov, grafické užívateľské rozhranie, alebo systém pre dávkové spracovanie úloh zadaných formou diernych štítkov. Potom tu je hardvér, ktorý musí byť spravovaný a zdieľaný medzi jednotlivými programami a užívateľmi. Ďalšou z úloh, ktoré má na starosti operačný systém je inštalácia dodatočného (aplikačného) softvéru.
Významnou úlohou je tiež organizácia a spravovanie dát: o to sa starajú súborové systémy. Opäť to zahŕňa kontrolu prístupu a zdieľanie príslušného hardvéru, ktorý ukladá konkrétne bity a bajty, medzi užívateľmi.
Treťou stranou, s ktorou operačný systém komunikuje, je aplikačný softvér. Okrem užívateľa a hardvéru potrebujú aplikačné programy na svoju funkciu služby operačného systému, keďže potrebujú s užívateľom komunikovať a hardvérové zdroje používať. Operačný systém je zodpovedný za oboje.

Časť 1.1: Komponenty

V tejto časti si rozoberieme, z čoho sa operačný systém skladá, pre lepšie porozumenie toho, čo operačný systém je.

Z čoho je OS zložený?

31

- jadro (kernel)
- systémové knižnice
- systémoví démoni / služby
- užívateľské rozhranie
- pomocné funkcie systému (utilities)

V podstate každý OS tieto obsahuje.

Operačné systémy sa skladajú z niekoľkých komponent, pričom niektoré sú dôležitejšie ako iné. V podstate všetky hore uvedené komponenty sú prítomné, v nejakej podobe, v každom operačnom systéme (výnimku môžu tvoriť maličké špecializované systémy). Jadro (kernel) je najdôležitejšia a najnižšia vrstva operačného systému, zatiaľ čo systémové knižnice sú nad ním a používajú služby jadra. Systémové knižnice naviac sprostredkovávajú služby jadra užívateľským programom a poskytujú ďalšie služby (ktoré nutne nemusia byť súčasťou jadra).
Zvyšné vrstvy sú tvorené prevažne programami v obvyklom zmysle: od užívateľských programov ich odlišuje v podstate iba fakt, že sú súčasťou operačného systému. Prvou kategóriou takýchto programov sú systémoví démoni (system daemons) alebo systémové služby (system services). Ide typicky o dlho bežiace programy, ktoré reagujú na udalosti alebo zaisťujú úlohy súvisiace s údržbou systému.
Užívateľské rozhranie je trochu komplikovanejšie, v zmysle, že je tvorené viacerými dielčími komponentami, ktoré zapadajú do ostatných zmienených častí. Táto odrážka zastrešuje časti užívateľského rozhrania, ktoré sú viac-menej štandardnými programami, napríklad interpret príkazov.

Jadro (Kernel)

32

- najnižšia úroveň operačného systému
- beží v privilegovanom režime
- spravuje všetok zvyšný softvér
  - vrátane ďalších komponent OS
- vynucuje izoláciu a bezpečnosť
- poskytuje nízkoúrovňové služby programom

Jadro je najnižšou a pravdepodobne najdôležitejšou časťou operačného systému. Odlišuje sa hlavne tým, že beží v špeciálnom režime procesora (známeho ako privilegovaný, monitorovací alebo tzv. supervisor mode). Hlavnými úlohami jadra sú správa základných hardvérových zdrojov (procesor, pamäť) a konkrétne poskytovanie týchto zdrojov ďalšiemu softvéru bežiacemu na počítači. Ďalší softvér zahŕňa aj zvyšok operačného systému.
Ďalšou kritickou úlohou je vynucovanie izolácie a bezpečnosti. Hardvér typicky poskytuje prostriedky na umožnenie vzájomnej izolácie jednotlivých programov, ale je to softvér (jadro OS), kto je zodpovedný za nastavenie tohto hardvérového vybavenia tak, aby pracovalo správne a efektívne.
Konečne, kernel často poskytuje najnižšiu úroveň rôznych služieb vyšším vrstvám. Tieto sú poskytované väčšinou v podobe systémových volaní, a súvisia najmä (priamo či nepriamo) s prístupom k hardvéru.

Systémové knižnice

33

- tvoria vrstvu nad jadrom OS (kernelom)
- poskytujú služby na vyššej úrovni
  - vnútri využívajú služby jadra
  - jednoduchšie na použitie než rozhranie jadra
- typický príklad: `libc`
  - poskytuje funkcie jazyka C ako `printf`
  - tiež známe ako `msvcrt` u Windows

Jednu úroveň nad kernelom sa nachádzajú systémové knižnice: mimo iné poskytujú rozhranie medzi kernelom a vyššími úrovňami systému. Rozhranie poskytované aplikáciám knižnicou je jednu úroveň abstrakcie nad službami jadra, čím je typicky jednoduchšie na použitie.
Ako iné knižnice, aj systémové knižnice sú prilinkované k programom a v podstate sa stávajú ich súčasťou: kvôli tomuto beží kód z knižnice s rovnakými oprávneniami ako kód aplikácie – pre funkcionalitu, ktorá nie je čisto výpočtová potrebujú systémové knižnice komunikovať s inými časťami operačného systému: kernelom alebo ďalšími privilegovanými komponentami (so systémovými službami, tiež známymi ako démoni).

Systémoví Démoni

34

- programy, ktoré bežia na pozadí
- buď poskytujú služby priamo
  - ale démoni sa líšia od systémových knižníc
  - viac si povieme v ďalších prednáškach
- alebo vykonávajú úlohy súvisiace s údržbou či periodické úlohy
- alebo vykonávajú úlohy, ktoré si vyžiadal kernel

Démoni sú dlho bežiace programy: starajú sa o úlohy, ktoré treba robiť nepretržite alebo periodicky, ale nepotrebujú byť súčasťou jadra. Zahŕňa to veci ako doručovanie internetovej pošty, vzdialený prístup k systému (napr. démon pre zabezpečený prístup k interpretu príkazov – secure shell), synchronizácia systémových hodín (démon pre protokol sieťového času), konfigurácia a prideľovanie sieťových adries (dynamic host control protocol daemon), a tak ďalej, správa front u tlačiarní, služby doménových mien, časti sieťového súborového systému, monitorovanie stavu hardvéru, systémové logovanie a ďalšie.

Užívateľské rozhranie

35

- sprostredkúva interakciu človeka s počítačom
- hlavný shell (interpret príkazov) je typicky súčasťou OS
  - príkazový riadok na UNIX-e alebo DOS
  - grafické rozhrania s pracovnou plochou a oknami
  - ale tiež tlačidlá na vašej mikrovlnke
- stavebné bloky pre aplikačné rozhranie (UI)
  - tlačidlá, panely (tabs), zobrazovanie textu, OpenGL...
  - poskytované systémovými knižnicami a/alebo démonmi

Na väčšine systémov nemôžu aplikačné programy priamo ovládať hardvér – preto je na operačnom systéme, aby zabezpečil rozhranie medzi užívateľom (ktorý pracuje s hardvérom) a aplikačným programom. Užívateľské rozhranie má na starosti užívateľský vstup (v podobe udalostí ako stlačenie klávesy, pohyb myšou, touchpad, interakcia s dotykovým displejom, ap.) a sprostredkovanie výstupu aplikácie užívateľovi (zobrazenie textu, vykreslenie okien na obrazovku, audio výstup, atď.).

Pomocné programy (System Utilities)

36

- malé programy potrebné pre úlohy súvisiace s OS
- napr. konfigurácia systému
  - veci ako editor registru (registry editor) na Windows
  - alebo jednoduché textové editory
- údržba súborového systému, správa démonov, ...
  - programy ako `ls`/`dir` alebo `newfs` či `fdisk`
- tiež väčšie programy, napríklad správca súborov

Nie všetky 'krátko bežiace' (teda nie démoni) programy spadajú pod aplikačný softvér. Existuje množstvo pomocných programov (utilít), ktoré pomáhajú so správou samotného operačného systému, konfiguráciou služieb a hardvéru atď. Tieto pomocné programy obvykle používajú rovnaký typ rozhrania ako aplikačný softvér – či už ide o rozhranie vo forme príkazov alebo grafické.
Nie vždy je rozdiel medzi pribaleným aplikačným softvérom a pomocnými systémovými programami zrejmý: napr. v prípade prehliadača súborov (file explorer) vo Windowse – ide o pomerne veľký a komplikovaný program, ale tiež zastáva ústrednú úlohu v bežnom používaní systému. Dokážeme si ale predstaviť, že niekto by zobral program ako Explorer a portoval ho na iný operačný systém. Možno by to vyžadovalo nemalé úsilie, ale pravdepodobne by tento program nevyčnieval v inom GUI-orientovanom operačnom systéme.
Na druhú stranu, množstvo malých programov má jednoznačný účel a dajú sa výrazne jednoduchšie klasifikovať. Ide napríklad o utilitu `ifconfig` pre sieťovú konfiguráciu, alebo pomocný program `fdisk`, slúžiaci na rozdelenie disku na partície (disk partitioning). Obdobne, programy ako `fsck` (alebo `chkdisk` u Windows) sú pomerne neužitočné mimo operačný systém, s ktorým prišli.

Voliteľné komponenty

37

- pribalený aplikačný softvér
  - webový prehliadač, prehrávače médií (media player), ...
- správa softvéru iných dodávateľov (3rd-party SW management)
- vývojové (programovacie) prostredie
  - napr. prekladač & linker pre jazyk C
  - C hlavičkové súbory ap.
- zdrojový kód

K operačnému systému sú často pribalené programy, ktoré nie sú nevyhnutnou súčasťou operačného systému, ani nie sú žiadnym spôsobom zapojené do jeho bežného chodu. Typickým príkladom je malá kolekcia hier – tradícia, ktorá sa datuje spätne do pôvodného Berkeley UNIX, ak nie ešte ďalej do minulosti, a odvtedy ju môžeme pozorovať u takmer všetkých univerzálnych (general-purpose) operačných systémov: hry ako Solitaire a Míny (Minesweeper), ktoré sú pribalené vo Windowse majú priam ikonický charakter. Samozrejme do tejto kategórie spadá aj ďalší softvér: MS Paint ani Windows Media Player nie sú v žiadnej forme nevyhnutné na používanie Windowsu, takisto ani webový prehliadač.
Na UNIX-e má tradične pribalený softvér trochu iný charakter, čo je zapríčinené rozdielnou cieľovou skupinou (a tiež tým, že pochádza z inej doby): často je spolu s UNIX-ovými systémami distribuovaný prekladač jazyka C, linker a podobne pokročilé editory zdrojového kódu. V niektorých prípadoch bola časť týchto programov dokonca nevyhnutná, keďže užívateľ musel preložiť jadro prispôsobené na svoj počítač. Už nejakú dobu to ale takto nie je, napriek tomu väčšina užívateľov UNIX-u očakáva, že prekladač jazyka C bude štandardne k dispozícii. Spolu s prekladačom jazyka C zvyknú byť prítomné hlavičkové súbory pre

systémové knižnice a ďalšie súbory potrebné na vytvorenie a preklad vlastných programov. Ako som povedal, iná doba.
Nakoniec môžete mať zdrojový kód operačného systému: prísne vzaté nejde o softvérovú komponentu, ale o inú (čitateľnú pre človeka namiesto spustiteľnú počítačom) reprezentáciu rovnakého operačného systému. Môže to byť veľmi užitočné, ak sa chcete naučiť nízkoúrovňové detaily fungovania počítačov a operačných systémov.

Časť 1.2: Rozhrania

Iný spôsob pohľadu na operačný systém je z pozície jeho okolia – aké druhy rozhraní existujú medzi operačným systémom a ostatnými komponentami počítača?

Programovacie rozhranie

39

- kernel poskytuje systémové volania
  - ABI: Application Binary Interface
  - definované formou strojových inštrukcií
- systémové knižnice poskytujú API
  - Application Programming Interface
  - symbolické / vysokoúrovňové rozhrania
  - typicky vo forme funkcií v jazyku C
  - systémové volania sú tiež dostupné vo forme API

Ako najviac zrejmé rozhranie operačného systému (najmä ak ste programátor) človeka napadne jeho API: Application Programming Interface (Rozhranie pre programovanie aplikácií) – ako napovedá jeho názov, ide o funkcionalitu, ktorú môžu získať programy od operačného systému. Väčšina tohto API je poskytovaná prostredníctvom systémových knižníc – balíkov podprogramov vo forme strojového kódu, z ktorých môžu aplikačné programy (a systémoví démoni a systémové pomocné programy – utility) volať funkcie. Tieto podprogramy často ďalej komunikujú s kernelom, využívajúc nízkoúrovňový protokol špecifický pre daný systém: protokol známy ako ABI (Application Binary Interface) daného kernelu. Programátori väčšinou nie sú vystavení detailom tohto ABI.
API je väčšinou popísané vo forme funkcií vo vyššom programovacom jazyku: obvykle C, niekedy C++ alebo Objective C, zriedka iný jazyk. Programátori používajú tieto funkcie rovnakým spôsobom, akým používajú funkcie, ktoré sami naimplementovali, s tým rozdielom, že nemusia dodať ich definície; prekladač ich preloží na volania do strojového kódu odpovedajúceho podprogramu (subroutine) v systémových knižniciach.

Posielanie správ (Message Passing)

40

- API nemajú vždy formu funkcií v jazyku C
- je možné mať rozhrania pre posielanie správ
  - založené na komunikácii medzi procesmi (inter-process c.)
  - možné dokonca aj naprieč sieťami
- systémoví démoni často poskytujú API v tejto forme
  - môže byť tiež obalená C-čkovým API

Funkcie jazyka C (alebo C++ alebo Objective C alebo iných programovacích jazykov) nie sú jediná forma API, ktorá existuje. Často sú prítomné rozhrania pre komunikáciu medzi procesmi (inter-process communication), najčastejšie v nejakej forme posielania správ. Táto podoba rozhrania zvykne byť poskytovaná napr. systémovými démonmi, napr. `syslogd` (zvyčajne používa UNIX-ový doménový soket – UNIX domain socket), alebo poštový démon (obvykle používa TCP soket).

Prenositeľnosť / Portovateľnosť

41

- niektoré úlohy OS vyžadujú úzku spoluprácu hardvéru
  - virtuálna pamäť a nastavenie CPU
  - platformovo závislé ovládače zariadení
- ale mnohé ju nevyžadujú
  - plánovacie algoritmy (scheduling)
  - alokácia pamäte
  - rôzne formy správy systému
- portovanie: zmena programu tak, aby dokázal bežať v novom prostredí
  - pre OS to typicky znamená nový hardvér

Je žiaduce, aby operačné systémy boli schopné bežať na rôznych hardvérových platformách: znižuje to náklady viacerými spôsobmi. Najlepší (a najlacnejší) kód je taký, ktorý nemusíte písať – použitie rovnakého operačného systému na rôznych hardvérových platformách znamená (do určitej miery) presne to. Na druhú stranu, šetrí to prostriedky vývojárom aplikácií, ktorí sa môžu zamerať na jeden operačný systém a obsiahnuť tým množstvo rôznych hardvérových zariadení; a tiež náklady na školenie – užívatelia môžu byť presunutí na novú hardvérovú platformu bez rozsiahleho preškoľovania na nový softvér.
Aj keď je prakticky nemožné napísať operačný systém tak, aby bol hardvérovo nezávislý, veľa jeho častí nemusia zaujímať detaily hardvérovej platformy. Toto sa týka dokonca niektorých centrálnych komponent kernelu (opäť, do určitej miery). Konkrétny plánovač vlákien môže byť často použitý bez zmeny (niekedy aj bez akéhokoľvek dodatočného ladenia) na inej hardvérovej platforme; to isté platí pre alokátory pamäte, kód súborového systému, atď.
Samozrejme, niektoré časti sú úzko späté s daným hardvérom: bootovacia sekvencia (čo zahŕňa veci ako nastavenie CPU a virtuálnej pamäte), ovládače zariadení (ktoré sú späté s konkrétnym zariadením) a tak ďalej.
Nakoniec, keď hovoríme o prenositeľnosti (portovateľnosti / portabilite), portovanie samotného operačného systému nie je jediným problémom: často je žiaduce portovať aplikačné programy, aby bežali na inej softvérovej architektúre (SW stack). Vo všeobecnosti, portabilita je schopnosť programu byť (jednoducho) prispôsobený na nové prostredie.

Hardvérová platforma

42

- CPU inštrukčná sada (ISA)
- zbernice, IO kontroléry
  - PCI, USB, Ethernet, ...
- firmvér, správa napájania (power management)

Príklady

- x86 (ISA) – PC (platforma)
- ARM – Snapdragon, i.MX 6, ...
- m68k – Amiga, Atari, ...

Čo je to hardvérová platforma? Je to voľná množina hardvéru a firmvéru, ktorý je často nachádzaný spolu, a ktorý dosahuje určitú úroveň spätnej (a doprednej) kompatibility v čase. Z pohľadu softvéru samozrejme nezáleží na špecifikách kremíka, iba na protokoloch a rozhraniach, ktoré sú prístupné softvéru.
Asi najznámejšou hardvérovou platformou je PC, datované spätne do IBM 80-tych rokov: pôvodne bolo postavené okolo procesorov Intel 8086, jedného z dvoch grafických adaptérov, 5.25" disketovej mechaniky, a, čo je dôležité, BIOS-u – firmvéru daného stroja. Jednotlivé komponenty boli medzičasom nahradené (väčšina viac než raz), ale šlo o postupný proces, v podstate zabezpečujúci kontinuitu pre softvérovú

architektúru až do dnešných časov. Na rozdiel od ostatných platforiem, PC bolo atypické v jednom aspekte: šlo o otvorenú platformu, v zmysle, že spoločnosti mimo IBM mohli dodať hardvér, ktorý spĺňal požiadavky danej platformy a tým na ňom mohol bežať rovnaký softvér.
Na rozdiel od PC, čo je v podstate jediná dôležitá platforma, ktorá používa procesorové jednotky (CPU) typu x86, ostatné CPU architektúry sa objavujú pri mnohých rôznych a nekompatibilných platformách: z historických príkladov ide napr. o sériu procesorov Motorola 68000, ktoré boli použité v počítačoch Atari a Amiga, a pracovných staniciach (workstation) Sun a NeXT.
Z moderných príkladov môžeme spomenúť CPU architektúru ARM, používanú hlavne v mobilných a iných nízkoenergetických zariadeniach, ktorá sa objavuje na množstve rôznych platforiem. V tejto oblasti má v podstate každý dodávateľ SoC (system on a chip – systém na čipe) svoju vlastnú platformu, s vlastnými protokolmi, firmvérom a potrebnými hardvérovými zariadeniami.

Prenositeľnosť Platforiem & Architektúry

43

- OS typicky podporuje veľa platforiem
  - Android na veľa rôznych typoch ARM SoC
- často aj rôzne CPU inštrukčné sady – ISA
  - dlhá tradícia v UNIX-ových systémoch
  - NetBSD beží na 15 rôznych ISA
  - mnoho z nich zahŕňa 6+ rôznych platforiem
- špecializované systémy sú obvykle menej prenositeľné

Moderné operačné systémy sú obvykle schopné bežať na veľa rôznych platformách, a často na niekoľkých rôznych CPU (inštrukčných sadách). Vezmite si napríklad Android OS, ktorý beží na množstve rôznych platforiem (v podstate každý SoC dodávateľ má vlastnú platformu) a 2 rôznych CPU architektúrach (ARM a x86).
Tradícia bola v podstate spustená UNIX-om, čo bol jeden z prvých operačných systémov napísaný vo 'vysokoúrovňovom' programovacom jazyku – jazyku, ktorý mohol byť preložený na strojový kód pre rôzne CPU. Skoršie operačné systémy boli obvykle napísané v assembly kóde špecifickom pre daný stroj.

Znovupoužitie kódu

44

- znovupoužitie kódu je veľmi logický krok
- väčšina kódu OS je HW-nezávislá
- nebolo to vždy tak
  - prišiel s tým UNIX, ktorý bol napísaný v C
  - bežný vtedajší OS bol napísaný v strojovom kóde
  - portovanie znamenalo v podstate „napísanie znova“

Prenositeľnosť (Portabilita) je špeciálnym prípadom znovupoužitia kódu (code re-use) – kód je napísaný raz a potom použitý niekoľkokrát v rôznych kontextoch, čo môže byť vyvolané zmenou hardvéru, alebo iných častí behového prostredia.

Prenositeľnosť aplikácií

45

- aplikáciám záleží viac na OS než na HW
  - aplikácie sú písané vo vyšších programovacích jazykoch
  - a hojne využívajú systémové knižnice
- stačí naportovať OS na nový/iný HW
  - väčšina aplikácií môže byť jednoducho znovu preložená
- stále veľká prekážka (cf. Itanium)

V princípe sa väčšina aplikácií nebaví s hardvérom priamo, takže im veľmi nezáleží na platforme, a tiež sú napísané v jazykoch, ktoré nie sú viazané na konkrétnu CPU architektúru. Komunikujú však s operačným systémom – a v mnohých prípadoch tvorí táto komunikácia významnú časť kódu aplikácie.
Ak toto vezmeme do úvahy, malo by byť pomerne jednoduché portovať existujúce aplikácie na nový hardvér, keďže aj na ich 'materskej' (native) aj na novej platforme beží rovnaký operačný systém. Je obvykle potrebné urobiť viac, než len opätovný preklad (recompilation), ale aj pre zložitejšie aplikácie to neznamená veľa úsilia. Rozdiely v hardvére platforiem (oproti periférií a veciam ako rozmery obrazovky a rozlíšenie) sú bezvýznamné u Androidových aplikácií a typické UNIX-ové programy budú bežať bez zmien na množstve platforiem po jednoduchom znovupreložení.
Nie všetky ekosystémy sú však takéto – uvážte prvý významný pokus migrovať PC na 64-bitovú architektúru, Itanium, na začiatku rokov 2000. Táto snaha zlyhala, z viacerých dôvodov, ale nedostatok prenositeľnosti MS Windows (a aplikácií cielených na MS Windows ako na ich jedinú podporovanú platformu) hrala významnú úlohu.

Prenositeľnosť aplikácií (2)

46

- rovnaká aplikácia môže často bežať na mnohých OS
- najmä vrámci rodiny POSIX
- ale rovnaká aplikácia môže bežať na Windows, macOS, UNIX, ...
  - Java, Qt (C++)
  - webové aplikácie (HTML, JavaScript)
- veľa systémov poskytuje rovnakú množinu služieb
  - rozdiely sú obvykle v programovacích rozhraniach
  - vysokoúrovňové knižnice a jazyky ich dokážu skryť

Okrem prenositeľnosti aplikačného softvéru na nový hardvér, čo by malo byť v podstate zadarmo (hoci niekedy nie je), aplikácie môžu byť portované na iné operačné systémy. Vrámci rodiny POSIX ide často o formalitu na úrovni podobnej portovaniu na nový hardvér – tieto operačné systémy sú, z pohľadu aplikácie, veľmi podobné. Toto je v skutočnosti hlavný dôvod, prečo POSIX vlastne existuje.
Komplikovanejší proces je portovanie medzi rôznymi operačnými systémami, ktoré nespadajú do rovnakej rodiny, napr. medzi Windowsom a POSIX-om. API sú veľmi odlišné, a na úrovni aplikácií je toto často úplne nepraktické. Ak je žiaduce, aby aplikácie bežali 'natívne' na rôznych operačných systémoch (mimo POSIX), môžu byť napísané použitím platformovo-nezávislého API, ktoré je implementované prenositeľným behovým prostredím (portable runtime), ktoré prekladá svoje vlastné API na volania podporované akýmkoľvek daným hostiteľským operačným systémom. Takéto prenositeľné behové prostredie je napríklad, známe, ako časť programovacieho jazyka Java ('write once, run everywhere' – napíšte raz, spúšťajte všade). Existujú ďalšie takéto behové prostredia, pre iné programovacie jazyky: C++ má Qt, väčšina vysokoúrovňových jazykov (Python, Haskell, JavaScript, ...) má nejaké prostredie vstavané (built in).

Abstrakcia

47

- inštrukčné sady poskytujú abstrakciu nad detailmi CPU
- prekladače abstrahujú nad inštrukčnými sadami
- operačné systémy abstrahujú hardvér
- prenositeľné behové prostredia abstrahujú nad operačnými systémami
- aplikácie sa nachádzajú nad týmito abstrakciami

Kým sme diskutovali o prenositeľnosti, vznikal obrázok 'veže abstrakcií': každá vrstva skrýva detaily predchádzajúcich vrstiev, čím

ich umožňuje ignorovať pri návrhu na vrchole veže (alebo zásobníka (stack), ako sa to často nazýva). Toto umožňuje prenositeľnosť (keďže sú nižšie vrstvy skryté, dajú sa jednoduchšie nahradiť), ale toto zďaleka nie je jediný dôvod prečo takéto veže budujeme.
Pravdepodobne je hlavným dôvodom abstrakcií skrývanie zložitosti – niečo tak jednoduché ako zapísanie 'hello world' do súboru na disku vyžaduje stovky tisíc CPU inštrukcií, ale my to môžeme urobiť jednoduchým krátkym príkazom. To je to, čo nám abstrakcie umožňujú.

Cena abstrakcií

48

- vyššia zložitosť
- nižšia výkonnosť
- presakujúce (leaky) abstrakcie

Výhody abstrakcií

- jednoduchšie písať a portovať softvér
- menej obmedzení na vývoj HW

Abstrakcie, samozrejme, nie sú zadarmo. Robiť veci nepriamo vždy viac stojí: ak by sme zredukovali množstvo abstrakcií, písanie do súborov by zabralo niekoľko tisíc alebo desiatok tisíc inštrukcií, miesto stoviek tisíc. Ale písanie čo i len 2000 inštrukcií na zapísanie súboru je 3 rády nad tým, čo je akceptovateľné pre písanie takejto jednoduchej a všadeprítomnej úlohy ručne. Takže akceptujeme pridanú réžiu.
Viac záludným problémom sú presakujúce – leaky abstrakcie: radi sa tvárime, že každá úroveň abstrakcií úplne utesní vrstvy pod ňou spôsobom, že na ne nevidíme. Ale skoro nikdy to tak nie je: všetky abstrakcie sú do určitej miery presakujúce – odhaľujú detaily vrstiev pod nimi. Ak je povrch magnetického disku poškodený, toto spôsobí problémy cez všetky vrstvy až v samotnej aplikácii, napriek tomu, že rozhranie, ktoré používame – čítanie a zapisovanie bajtov do súboru – nás malo uchrániť od takýchto nízkoúrovňových problémov ako sú defekty na krútiacom sa disku pokrytom feromagnetickým prachom.

Kompromisy abstrakcií

49

- výkonný hardvér umožňuje viac abstrakcií
- embedded a real-time systémy až tak nie
  - OS je menší a menej prenositeľný
  - rovnako to platí pre aplikácie
  - efektívnejšie využitie zdrojov

Je zrejmé, že môže existovať 'príliš málo' abstrakcie (predstavte si nahradenie vášho 3-riadkového Python skriptu ručným prepísaním na 25 tisíc assembly inštrukcií). Ale tiež ich môže byť priveľa – najmä na obmedzenom hardvéri môže byť réžia príliš veľká. Často je najjednoduchšou cestou ako získať efektivitu redukovanie množstva abstrakcií.

Časť 1.3: Klasifikácia

Náš posledný pokus o pochopenie toho, čo je operačný systém, sa bude týkať rôznych typov operačných systémov a rozdielov medzi nimi.

Univerzálne (general-purpose) operačné systémy

51

- použiteľné vo väčšine situácií
- flexibilné ale zložité a veľké
- môžu bežať na serveroch aj klientoch
- ich osekané verzie bežia na smartfónoch
- podporujú množstvo hardvéru

Najdôležitejšiu a najzaujímavejšiu kategóriu tvoria tzv. 'general-purpose' (univerzálne alebo všeobecné) operačné systémy. V tomto kurze sa budeme baviť hlavne o nich. Systémy v tejto kategórii sú pomerne flexibilné (aby mohli pokryť všetko, na čo ľudia zvyčajne počítače používajú), ale z rovnakého dôvodu sú aj pomerne zložité. Často je rovnaký operačný systém schopný bežať na tzv. 'serveroch' (počítačoch, ktoré zväčša ležia v dátových centrách a poskytujú služby vzdialene iným počítačom) aj na počítačoch, ktoré vystupujú ako 'klienti' – také, ktoré komunikujú s užívateľmi priamo.
Podobne, rovnaký operačný systém, možno v zmenšenej verzii, môže bežať na smartfóne alebo na zariadení, ktoré je podobne obmedzené svojou veľkosťou a dostupnou energiou. Všetky súčasné hlavné operačné systémy smartfónov sú tohto typu. Historicky existovalo niekoľko viac špecializovaných operačných systémov pre telefóny, najmä preto, že vtedy bol hardvér telefónov výrazne viac obmedzený, než dnes. Avšak, vieme si predstaviť, že operačný systém ako napríklad Symbian, mohol byť použitý v osobných počítačoch, za predpokladu, že by bola rozšírená jeho hardvérová podpora.

Operačné systémy: príklady

52

- Microsoft Windows
- Apple macOS & iOS
- Google Android
- Linux
- FreeBSD, OpenBSD
- MINIX
- veľa, veľa ďalších

Existuje množstvo operačných systémov, dokonca aj u univerzálnych (general-purpose) operačných systémov. Aj keď samotné spúšťanie OS nie je hlavným dôvodom na obstarávanie si počítača (tým je aplikačný softvér), hraje významnú úlohu pri skúsenosti užívateľa. Samozrejme tiež komunikuje s počítačovým hardvérom a aplikačnými programami, a nie všetky systémy bežia na všetkých počítačoch a nie všetky aplikácie budú bežať na všetkých operačných systémoch.

Operačné systémy na zvláštne účely

53

- embedded zariadenia
  - obmedzený rozpočet
  - malé, pomalé, energeticky-obmedzené
  - zložité alebo nemožné aktualizovať
- real-time systémy
  - musia reagovať na udalosti z reálneho sveta
  - často kritické z hľadiska bezpečnosti (safety-critical)
  - roboty, autonómne autá, vesmírne sondy, ...

Okrem univerzálnych (general-purpose) operačných systémov existujú ďalšie, viac obmedzené systémy. Typickým príkladom sú embedded systémy, ktoré bežia na veľmi malom množstve hardvéru (v porovnaní s modernými univerzálnymi počítačmi), s obmedzeným rozpočtom a výraznými energetickými obmedzeniami. V týchto systémoch nie je prítomné pohodlie, ktoré nám poskytovala nadmerná abstrakcia. Je to výrazné hlavne u real-time systémov, ktoré ešte pridávajú obmedzenia na výpočetný čas. Predvídateľné časovanie je jednou z vecí, ktoré sa veľmi ťažko dosahujú v prítomnosti abstrakcií (inými slovami, precízne časovanie operácií je jedna z vecí, ktoré prenikajú za hranice abstrakcií).

Veľkosť a zložitosť

54

- operačné systémy sú obvykle veľké a zložité
- typicky 100K a viac riadkov kódu
- 10+ miliónov je kľudne možné
- veľa tisíc človeko-rokov práce
- špecializované systémy sp výrazne menšie

Ako sme už spomenuli skôr, univerzálne (general-purpose) operačné systémy sú zvyčajne veľké a zložité. Najmenšie úplné operačné systémy (ak nie sú výhradne učebnými hračkami) začínajú na 100 tisíc riadkoch kódu, ale typicky ide skôr o milióny riadkov. Nie je nezvyčajné, že operačný systém je tvorený viac než 10 miliónmi riadkov kódu. Takéto množstvo kódu zjavne reprezentuje tisícky človeko-rokov práce – napísanie si vlastného operačného systému, sólo, nie je veľmi realistické.
Špecializované systémy sú často výrazne menšie. Obvykle podporujú výrazne menej hardvérových zariadení a poskytujú jednoduchšie a menej rôznorodé služby 'aplikačnému' softvéru.

Opäť Kernel

55

- chyby v kerneli sú veľmi nepríjemné
  - zlyhanie systému, strata dát
  - kritické bezpečnostné chyby
- väčší kernel znamená viac chýb
- externé ovládače vnútri v kerneli?

Pripomeňme si, že kernel beží v privilegovanom CPU režime. Akýkoľvek softvér bežiaci v tomto režime je v podstate všemocný a môže ľahko obísť prístupové obmedzenia alebo bezpečnostnú ochranu. Je to známy fakt, že čím viac kódu máte, tým viac je v ňom chýb. Keďže chyby v kerneli môžu mať ďalekosiahle a katastrofické následky, je nevyhnutné, aby ich tam bolo čo najmenej. Čo je ešte dôležitejšie, ovládače zariadení často potrebujú prístup k hardvéru a najjednoduchší (a niekedy jediný) spôsob, ako to dosiahnuť je, aby bežali v režime kernelu (privilegovanom režime).
Ako tiež možno viete, ovládače zariadení sú často pochybnej kvality: predajcovia hardvéru ich často považujú za nepodstatný dodatok a nevenujú veľkú pozornosť svojim softvérovým tímom. Ak tieto ovládače budú bežať v režime kernelu, nastáva tým vážny problém. Rôzni výrobcovia OS uplatňujú rôzne stratégie, ako zmierniť tento problém.
Z vyššie uvedeného vyplýva, že chceme, aby boli kernely malé a chceme odtiaľ vytlačiť čo najviac ovládačov. Nie je však jednoduché (ani očividne správne) to urobiť. Sú dve hlavné myšlienkové prúdy pokiaľ ide o návrh 'veľkosti' kernelu:

Monolitické kernely

56

- veľa kódu v jadre
- menej abstrakcie, menej izolácie
- rýchlejšie a výkonnejšie

Mikrokernely

- čo najviac vecí presunúť mimo kernel
- viac abstrakcie, viac izolácie
- pomalšie a menej výkonné

Monolitický kernel je starším a v určitom zmysle jednoduchším návrhom. Veľká časť kódu končí v kerneli, čo nie je problém do momentu, kým sa neobjavia chyby. V tomto návrhu je prítomných menej abstrakcií, menej rozhraní a všeobecne menej pohyblivých častí na rovnaké množstvo funkcionality. Z týchto vlastností plynie rýchlejší beh a výhodnejšie využitie zdrojov. Tieto kernely sa nazývajú monolitické, lebo všetko, čo robí bežný kernel je vykonávané jedinou (monolitickou) časťou softvéru.
Opakom monolitických kernelov sú mikrokernely. Samotný kernel v takomto systéme je najmenšou možnou podmnožinou kódu, ktorá musí bežať v privilegovanom režime. Všetko, čo môže byť presunuté do užívateľského režimu (procesora), presunuté je. Tento návrh poskytuje výrazne viac izolácie a vyžaduje viac abstrakcie. Rozhrania v rôznych častiach nízkoúrovňových služieb OS sú komplikovanejšie. Subsystémy sú od seba navzájom dobre izolované a chyby sa až tak jednoducho nepropagujú. Operačné systémy, ktoré používajú tento typ jadra však bežia pomalšie a využívajú zdroje menej efektívne.

Paradox?

57

- real-time & embedded systémy často používajú mikrokernely
- izolácia je vhodná pre spoľahlivosť
- efektivita tiež závisí od tzv. workload, pracovnej záťaže
  - priepustnosť vs. latencia
- real-time nemusí nutne znamenať rýchly

Nakoniec, existuje mierny paradox čo sa týka mikrokernelov: je známe, že sú často použité v embedded (resp. real-time) systémoch – jedných z najvýraznejšie výkonovo-kritických softvérových architektúr, aké existujú. Je to tým, že existuje vec, ktorá je dôležitejšia, než výkonnosť, pokiaľ ide o embedded softvér: spoľahlivosť. A spoľahlivosť je to, v čom mikrokernely excelujú. Naviac, aj u náročných real-time systémov, kde často považujeme výkon za prvoradý, rýchlosť samotná je tak trochu falošným údajom – dôležitá je latencia, a ešte dôležitejšia je horná hranica tejto latencie. Dodať túto hranicu je výrazne jednoduchšie u mikrokernelových systémov, kde je množstvo kódu výrazne menšie a výrazne jednoduchšie sa o ňom dajú vyvodzovať závery.

Review Questions

58

1. What are the roles of an operating system?
2. What are the basic components of an OS?
3. What is an operating system kernel?
4. What are API and ABI?

Časť 2: Systémové knižnice a API

V tejto sekcii sa budeme zaoberať programovacími rozhraniami operačných systémov, najprv všeobecne, bez väzby na konkrétny systém.

Potom sa konkrétne budeme zaoberať rozhraním jazyka C v POSIX-ových systémoch.

Programovacie rozhrania

60

- rozhranie systémových volaní jadra
- → systémové knižnice / API ←
- protokoly pre medzi-procesovú komunikáciu
- pomocné programy z príkazovej riadky (skriptovanie)

Vo väčšine operačných systémov je najnižšie rozhranie, ktoré je prístupné aplikačným programom, rozhranie systémových volaní. Typicky je definované vo forme protokolu na úrovni strojového jazyka (t.j. ABI), ale tiež zvykne byť poskytované vo forme API jazyka C. To je prípad napríklad systémových volaní, ktoré určuje štandard POSIX, ale tiež to platí u systémov Windows NT.

Obsah prednášky

61

1. Programovací jazyk C
2. Systémové knižnice
   - čo je to knižnica?
   - hlavičkové súbory & knižnice
3. Prekladač & Linker
   - objektové súbory, spustiteľné binárky
4. API založené na súboroch (file-based)

Prednášku začneme opakovaním (alebo prípadne predstavením si) jazyka C. Potom sa dostaneme k téme knižníc, všeobecne aj špeciálne systémových knižníc. Pozrieme sa, ako knižnice vstupujú do procesu prekladu a aké ďalšie zložky doň vstupujú. Nakoniec sa bližšie pozrieme na konkrétnu množinu programovacích rozhraní na úrovni súborov.

Poznámka: UNIX a POSIX

62

- tieto termíny budeme väčšinou používať zámenne
- ide o rodinu operačných systémov
  - začiatok v neskorých 60-tych / skorých 70-tych rokoch
- POSIX je špecifikácia / norma
  - dokument popisujúci, čo by mal OS poskytovať
  - vrátane programovacích rozhraní

Budeme sa baviť o POSIX-e, pokiaľ nie je uvedené inak.

Než začneme, je nutné poznamenať, že počas tohto kurzu budeme používať v príkladoch systémy POSIX a UNIX. Ak bude spomenutá konkrétna funkcia alebo rozhranie bez stanovenia nejakých ďalších podmienok, budeme predpokladať, že je vymedzená štandardom POSIX a implementovaná v systémoch na báze UNIX-u.

Časť 2.1: Programovací jazyk C

Jazyk C je jedným z najpoužívanejších programovacích jazykov v implementáciách operačných systémov. Je tiež predmetom PB071 a v tomto bode by ste už mali byť oboznámení s jeho základnou syntaxou. Podobne sa od vás očakáva, že rozumiete konceptu funkcie a ďalších základných stavebných blokov programov. Aj keď nepoznáte syntax špecifickú pre jazyk C, je veľmi podobná akémukoľvek programovaciemu jazyku, ktorý poznáte.

Programovacie jazyky

64

- existuje veľa rôznych jazykov
  - C, C++, Java, C#, ...
  - Python, Perl, Ruby, ...
  - ML, Haskell, Agda, ...
- ale C má vo väčšine OS špeciálne miesto

Rôzne programovacie jazyky boli vytvorené na rôzne účely a existujú na rôznych úrovniach abstrakcie. Väčšina jazykov mimo jazyka C, s ktorými sa stretnete, či už na univerzite alebo v praxi, patrí do tzv. vysokoúrovňových jazykov. Existuje množstvo rodín jazykov, a veľa vysokoúrovňových jazykov vyvinutých z C, ako C++, Java alebo C#. Pre účely tohoto kurzu sa budeme zaoberať hlavne čistým C a POSIX-ovým (Bourne-style) shell-om, ktorý tiež možno považovať za programovací jazyk.

C: Najmenší spoločný základ

65

- až na assembly je C „nevyhnutné minimum“
- môžeme o C uvažovať takmer ako o prenositeľnom assembly
- je veľmi jednoduché volať C-čkové funkcie
- a používať C-čkové dátové štruktúry

knižnice jazyka C môžete použiť v skoro každom jazyku

O jazyku C môžete uvažovať ako o ‘prenositeľnom assembleri’, s nejakými menšími pridanými vlastnosťami, ktorý je poskytovaný vo forme štandardnej knižnice. Okrem tejto knižnice so základnými a veľmi užitočnými podprogramami (subroutines) C poskytuje: abstrakciu od strojového kódu (s užívateľsky prívetivou infixovou syntaxou u operátorov), štruktúrovaným tokom riadenia (control flow) a automatickými lokálnymi premennými, ako svoje hlavné výhody nad assembly.
Konkrétnou vlastnosťou, ktorá sa ukázala ako veľmi užitočná v prvých operačných systémoch, bola abstrakcia nad cieľovým procesorom a jeho inštrukčnou sadou. Táto vlastnosť tiež pomohla upevniť myšlienku, že operačný systém je entita oddelená od hardvéru.
Mimo to je C tiež populárne ako systémový programovací jazyk pretože takmer akýkoľvek program, nezáležiac od toho v akom jazyku je napísaný, dokáže pomerne jednoducho volať C-čkové funkcie a používať dátové štruktúry jazyka C.

Jazyk operačných systémov

66

- veľa (väčšina) kernelov je napísaných v C
- väčšinou sa to vzťahuje aj na systémové knižnice
- niekedy sa to týka takmer celého OS
- operačné systémy, kt. nie sú napísané v C poskytujú C API

C sa v podstate stalo ‘jazykom operačných systémov’: väčšina kernelov a dokonca väčšina zvyšku kódu operačných systémov je napísaná v C. Každý operačný systém (možno na pár výnimiek) poskytuje v nejakej forme štandardnú knižnicu jazyka C a dokáže spúšťať programy napísané v C (a čo je dôležitejšie, poskytovať im základné služby).

Časť 2.2: Systémové knižnice

Už minulý týždeň sme načrtli tému systémových knižníc, v sekcii ‘anatómia’. Teraz je načase pozrieť sa na ne detailnejšie: čo obsahujú, ako sú uložené v súborovom systéme, ako vstupujú do programov. Tiež

budeme stručne hovoriť o tzv. wrapperoch systémových volaní (funkciách, kt. „obaľujú“ systémové volania a sprostredkovávajú nízkoúrovňový prístup k službám jadra – viac túto tému rozoberieme na ďalšej prednáške). Nakoniec sa pozrieme na pár príkladov systémových knižníc, ktoré sa objavujú v populárnych operačných systémoch.

(Systémové) Knižnice

68

- najmä C-čkové funkcie a dátové typy
- rozhrania definované v hlavičkových súboroch
- definície poskytované v knižniciach
  - statické knižnice (archívy): `libc.a`
  - zdieľané (dynamické) knižnice: `libc.so`
- na Windowse: `msvcrt.lib` a `msvcrt.dll`
- existuje ich výrazne viac než len `libc` / `msvcrt`

Keď budeme v tomto kurze hovoriť o knižniciach, budeme konkrétne myslieť C-čkové knižnice. Nie moduly jazykov Python alebo Haskell, ktoré sú dosť odlišné. Typická C-čková knižnica sa v podstate skladá z dvoch častí, jednou sú hlavičkové súbory, ktoré špecifikujú popis rozhrania (API) a druhou je skompilovaný kód knižnice (archívu alebo zdieľanej knižnice).
Rozhranie (ako je definované v hlavičkových súboroch) sa skladá z funkcií (pre ktoré je v hlavičkovom súbore definovaný typ argumentov a typ návratovej hodnoty) a z dátových štruktúr. Telá funkcií (ich implementácie) sú to, čo tvorí kompilovaný kód knižníc. Pre ilustráciu:

Deklarácia: čo ale nie ako

69

```
int sum( int a, int b );
```

Definícia: ako sa daná operácia má vykonať?

```
int sum( int a, int b )
{
    return a + b;
}
```

Prvý príklad na tomto slide je deklarácia: hovorí nám názov funkcie, jej vstupy a jej výstup. Druhý príklad sa volá definícia (niekedy tiež telo) funkcie a obsahuje operácie, ktoré sa majú vykonať, keď je funkcia zavolaná.

Knižničné súbory

70

- v `/usr/lib` na väčšine Unixov
  - môžu byť pomiešané s aplikačnými knižnicami
  - hlavne na systémoch odvodených z Linuxu
  - tiež `/usr/local/lib` pri užívateľských/aplikačných knižniciach
- na Windowse: `C:\Windows\System32`
  - užívateľské knižnice často pribalené k programom

Strojový kód, ktorý tvorí knižnicu (t.j. kód, ktorý bol vygenerovaný z definícií funkcií) žije v súboroch. Tieto súbory sú to, čo zvyčajne nazývame ‘knižnice’ a tieto sa obvykle nachádzajú na špeciálnom mieste v súborovom systéme. Na väčšine UNIX-ových systémov sú umiestnené v `/usr/lib` prípadne `/lib` pre systémové knižnice a `/usr/local/lib` pre užívateľské alebo aplikačné knižnice. Na niektorých systémoch (najmä odvodených z Linuxu) sú užívateľské knižnice pomiešané spolu so systémovými knižnicami a všetky sú uložené v `/usr/lib`.
Na Windowse je situácia podobná v tom, že systémové aj aplikačné knižnice sa nachádzajú na rovnakom mieste. Okrem toho sú u systémov Windows (a macOS) zdieľané knižnice často nainštalované spolu s aplikáciou.

Statické knižnice

71

- uložené v `libfile.a`, alebo `file.lib` (Windows)
- potrebné iba pri preklade (linkovaní) programov
- kód je skopírovaný do spustiteľného súboru
- výsledný spustiteľný súbor sa tiež nazýva statický
  - operačnému systému sa s ním jednoduchšie pracuje
  - dochádza ale k plytvaniu miestom

Statické knižnice sa používajú iba počas prekladu na spustiteľné súbory a nie sú potrebné na normálnu prevádzku systému. Preto ich veľa operačných systémov štandardne neinštaluje – musia byť nainštalované osobitne ako súčasť vývojového balíka. Keď je statická knižnica prilinkovaná k programu, v podstate to obnáša kopírovanie strojového kódu z knižnice do výslednej binárky (spustiteľného súboru).
V tomto prípade po linkovaní už knižnica nie je potrebná, keďže binárka obsahuje všetok kód potrebný pre svoj beh. Pre systémové knižnice to znamená, že kód pochádzajúci z knižnice je prítomný na systéme v mnohých kópiách, raz pre každý program, ktorý knižnicu používa. Plytvanie je trochu zmiernené tým, že linkre kopírujú iba časti knižnice, ktoré program reálne potrebuje, ale stále ide o výraznú duplikáciu.
Takto spôsobená duplikácia neovplyvňuje iba súborový systém, ale aj pamäť (RAM) v bode, keď sú tieto programy načítavané – niekoľko kópií rovnakej funkcie bude načítaných do pamäte keď sú programy spustené.

Zdieľané (Dynamické) knižnice

72

- potrebné pre beh programov
- linkovanie sa deje v čase spustenia (at execution time)
- menej duplikácie kódu
- môže byť aktualizovaná osobitne
- ale: problémy so závislosťami

Druhým prístupom čo sa týka knižníc sú dynamické alebo zdieľané knižnice. V tomto prípade je knižnica potrebná pre vlastný beh programu: linker nekopíruje strojový kód z knižnice do spustiteľného súboru. Namiesto toho si iba poznačí, že knižnica musí byť načítaná spolu s programom, keď je tento spustený.
Toto redukuje duplikáciu kódu, ako na disku tak aj v pamäti. Tiež to znamená, že knižnica môže byť aktualizovaná oddelene od aplikácie. Tým sú aktualizácie často zjednodušené, najmä v prípade, že knižnica je používaná množstvom programov, a napríklad sa zistí, že obsahuje bezpečnostnú chybu. U statických knižníc to znamená, že každý program, ktorý túto knižnicu používa musí byť aktualizovaný. Zdieľaná knižnica môže byť nahradená a opravený kód načítaný popri programoch ako zvyčajne.
Nevýhodou je, že je problematické dodržiavať binárnu kompatibilitu – aby sa zabezpečilo, že programy, ktoré boli vytvorené s nejakou verziou knižnice budú fungovať aj s neskoršou verziou. Ak je to porušené, čo sa často stáva, ľudia narážajú na problémy so závislosťami (na Windowse tiež známe ako DLL peklo (DLL hell)).

Hlavičkové súbory

73

- na UNIXe: `/usr/include`
- obsahujú prototypy funkcií jazyka C
- a definície C-čkových dátových štruktúr
- nevyhnutné na preklad (kompiláciu) C a C++ programov

Rovnako ako u statických knižníc, hlavičkové súbory sú potrebné iba pri preklade programov, nie na ich beh. Hlavičkové súbory sú fragmenty zdrojového kódu jazyka C, a na UNIX-ových systémoch sa tradične nachádzajú v `/usr/include`. Užívateľom nainštalované hlavičkové súbory (tzn. nie tie dodané systémovými knižnicami) žijú v `/usr/local/include` (avšak opäť pripomíname, že na deriváciách Linuxu sú užívateľské a systémové hlavičky často pomiešané v `/usr/include`).

Hlavičkový súbor Ukážka 1 (z `unistd.h`)

74

```
int     execv(char *, char **);
pid_t   fork(void);
int     pipe(int *);
ssize_t read(int, void *, size_t);
```

(a veľa ďalších prototypov)

Toto je výňatok z reálneho systémového hlavičkového súboru; deklaruje niekoľko funkcií, ktoré tvoria POSIX C API.

Hlavičkový súbor Ukážka 2 (zo `sys/time.h`)

75

```
struct timeval
{
    time_t    tv_sec;
    long      tv_usec;
};

/* ... */

int gettimeofday(timeval *, timezone *);
int settimeofday(timeval *, timezone *);
```

Toto je ďalší výňatok z reálnej hlavičky – tentokrát úryvok obsahuje definíciu dátovej štruktúry. Schéma (layout – poradie polí a ich typy, spolu so skrytým zarovnaním (tzv. padding)) takýchto štruktúr je veľmi dôležitá, keďže sa stáva súčasťou ABI. Inými slovami, definícia uvedená vyššie nepopisuje len vysokoúrovňové rozhranie ale tiež ako sú bajty rozložené v pamäti.

POSIX C knižnica

76

- `libc` – behové prostredie (runtime) jazyka C
- obsahuje ISO C funkcie
  - `printf`, `fopen`, `fread`
- a množstvo POSIX funkcií
  - `open`, `read`, `gethostbyname`, ...
  - C wrappery pre systémové volania

Ako sme už spomenuli, je tradíciou UNIX-ových systémov, že `libc` kombinuje základnú knižnicu jazyka C a základnú POSIX-ovú knižnicu. Pre to, čo bude nasledovať, je špecifická podmnožina POSIX-ovej knižnice dosť podstatná, najmä wrappery systémových volaní. Ide o C-čkové funkcie, ktorých jediným cieľom je zavolať ich odpovedajúce systémové volania.

Systémové volania: číslovanie

77

- systémové volania sú uskutočnené na strojovej úrovni
- ktoré systémové volanie sa zavolá je určené číslom
  - napr. `SYS_write` je 4 na OpenBSD
  - čísla sú definované hlavičkou `sys/syscall.h`
  - rôzne pre každý OS

Na úrovni jadra OS (viac budúci týždeň) sú systémové volania reprezentované číslami (ktoré často dostanú symbolické názvy ako `SYS_write`, ale stále ide o malé celočíselné hodnoty a nie o adresy v pamäti ako u bežných funkcií v C). Čísla sú špecifické pre daný kernel. Samozrejme, `libc` musí používať rovnaké číslovanie ako kernel.

Systémové volania: funkcia `syscall`

78

- existuje C-čková funkcia nazvaná `syscall`
  - prototyp: `int syscall( int number, ... )`
- implementuje nízkoúrovňovú sekvenciu systémového volania
- ako argumenty berie číslo sys. volania a jeho parametre
  - niečo na spôsob `printf`
  - prvý parameter rozhodne čo reprezentujú ďalšie parametre
- (viac o tom ako funguje `syscall()` ďalší týždeň)

Väčšina systémových volaní typicky funguje v podstate rovnako: knižnica vezme číslo (systémového volania) a nejaké dodatočné dáta (parametre), uloží ich do pred-pripravenej oblasti (registre, pamäť) a skočí do kernelu. Keďže je táto sekvencia uniformná cez všetky systémové volania, je možné mať jednu funkciu v C, ktorá dokáže vykonať akékoľvek systémové volanie, na základe jeho čísla.
Táto funkcia naozaj existuje a volá sa `syscall`. Takže v skutočnosti je možné realizovať všetky systémové volania pomocou tejto jednej funkcie jazyka C, a nikdy nezavolať pohodlnejšie wrappery, z ktorých má každé jeden účel (pre konkrétne systémové volanie).

Systémové volania: wrappery

79

- použitie `syscall()` priamo je nepohodlné
- `libc` má funkciu pre každé systémové volanie
  - `SYS_write` → `int write( int, char *, size_t )`
  - `SYS_open` → `int open( char *, int )`
  - a tak ďalej
- interne môžu tieto wrappery používať `syscall()`

Aby sme si programovanie značne uľahčili, miesto volania

```
syscall( SYS_write, fd, buffer, size );
```

môžeme použiť funkciu nazvanú `write`, takto:

```
write( fd, buffer size );
```

Nielenže je kratšia na písanie, tiež je bezpečnejšia: prekladač si môže overiť, že sme poslali správny počet a typy argumentov. Funkcia vnútri môže používať ekvivalent volania funkcie `syscall()` – v praxi však obvykle takúto abstrakciu obetujeme (tzn. každý wrapper je kópia prakticky totožného kódu, namiesto aby všetky volali `syscall`). Týmto sa ušetrí niekoľko inštrukcií pri každom systémovom volaní (a keďže sa to deje často, v úhrne sa jedná o nezanedbateľnú úsporu).

Prenositeľnosť / Portabilita

80

- knižnice poskytujú úroveň abstrakcie nad vnútornými časťami OS
- sú zodpovedné za prenositeľnosť aplikácií
  - spolu so štandardizovaným umiestnením v rámci súborového systému
  - a do určitej miery s užívateľskými pomocnými programami
- vysokoúrovňové jazyky sa spoliehajú na systémové knižnice

Dôležitou funkciou knižníc je poskytovať jednotné API vyšším vrstvám systému. Návrhári operačných systémov sa môžu rozhodnúť výrazne sa odkloniť od tradičného protokolu systémových volaní, alebo dokonca od tradičnej sady systémových volaní. Na druhú stranu, aj keď kernel nevyzerá úplne POSIX-ovo, často je stále možné dodať sadu C-čkových funkcií, ktoré sa chovajú tak, ako to špecifikuje POSIX. Bolo to v minulosti urobené viac než raz, najčastejšie nad mikrokernelmi, napr. Microsoft NT (Windows NT, XP a novšie) alebo na Mach (macOS, HURD). Všetky tieto systémy sú schopné podpory pre POSIX-ové programy, bez toho aby boli vybudované okolo UNIX-ového monolitického kernelu.
Samozrejme, API samotné nie je dostatočné na to, aby POSIX-ové programy fungovali správne: existujú určité očakávania týkajúce sa súborového systému (sémantika súborového systému samotného, ale aj ktoré súbory majú existovať a čo majú obsahovať) a ďalších aspektov systému.

NeXTSTEP a Objective C

81

- NeXT OS bol vybudovaný okolo Objective C
- systémové knižnice mali ObjC API
- pokiaľ ide o API, ObjC je veľmi rozdielne od C
  - tiež veľmi rozdielne od C++
  - tradičné OOP črty (ako Smalltalk)
- čiastočne zdedené do macOS
  - Swift je odvodený od Objective C

Nie všetky operačné systémy dodávajú (výhradne) C API. Historicky, jedným zo skorších odčlenení bol operačný systém NeXT, ktorý do veľkej miery používal Objective C. Kým procedurálna časť tohto jazyka je jednoducho C, jeho objektovo-orientovaná časť je založená na Smalltalk-u, s rozšírenou neskorou väzbou (late binding) a dynamickými typmi.

Systémové knižnice: UNIX

82

- matematická knižnica `libm`
  - implementuje matematické funkcie ako `sin` a `exp`
- knižnica pre vlákna `libpthread`
- prístup k terminálu: `libcurses`
- kryptografia: `libcrypto` (OpenSSL)
- C++ štandardná knižnica `libstdc++` alebo `libc++`

Zatiaľ čo `libc` je pomerne ústredná, existuje veľa ďalších knižníc, ktoré sú súčasťou UNIX-ového systému. Väčšinu z vyššie uvedených príkladov nájdete v nejakej forme na väčšine UNIX-ových systémov.

Systémové knižnice: Windows

83

- `msvcrt.dll` – ISO C funkcie
- `kernel32.dll` – základné OS API
- `gdi32.dll` – Graphics Device Interface – Rozhranie pre grafické zariadenia
- `user32.dll` – štandardné GUI elementy

Systémové knižnice vyzerajú na Windowse dosť rozdielne: nie je tu žiadna `libc`: namiesto toho má štandardná knižnica jazyka C svoje vlastné DLL (`msvcrt`, od MicroSoft Visual C RunTime), kým služby operačného systému (tie nízkoúrovňové) žijú v `kernel32.dll`. Zvyšné dve knižnice umožňujú aplikáciám poskytovať grafické užívateľské rozhranie. Všetky tu spomenuté knižnice dodávajú C API, avšak existujú aj C++ a C# rozhrania (ktoré sú čiastočne wrappermi nad vyššie uvedenými knižnicami, ale nie je to podmienka).

Dokumentácia

84

- manuálové stránky na UNIXe
  - skúste napr. `man 2 write` na `aisa.fi.muni.cz`
  - sekcia 2: systémové volania
  - sekcia 3: knižničné funkcie (`man 3 printf`)
- MSDN na Windowse
  - <https://msdn.microsoft.com>
- z týchto zdrojov sa môžete naučiť veľa

Väčšina dodávateľov OS poskytuje rozsiahlu dokumentáciu svojich rozhraní pre programátorov. Na UNIXe je táto typicky súčasťou samotnej inštalácie OS (manuálové stránky, príkaz `man`), zatiaľ čo na Windowse ide o samostatný zdroj (momentálne dostupný online, v minulosti distribuovaný v tlačenej verzii alebo na optickom médiu).

Časť 2.3: Prekladač & Linker

Hoci preklad – kompilácia (a linkovanie) nie sú základnou funkcionalitou operačného systému, je pomerne užitočné porozumieť tomu, ako tieto komponenty pracujú. V prvých systémoch bol naviac prekladač jazyka C považovaný za dôležitý komponent a táto tradícia dodnes pokračuje v množstve moderných UNIX-ových systémov. Rozoberieme si rôzne artefakty súvisiace s prekladom – objektové súbory, knižnice a spustiteľné súbory (binárky), ale tiež proces linkovania objektového kódu a knižníc na vytvorenie spustiteľných súborov. Takisto si zdôrazníme rozdiely medzi statickými a zdieľanými (dynamickými) knižnicami.

Prekladač jazyka C

86

- veľa POSIX-ových systémov je dodávaných s prekladačom jazyka C
- prekladač na vstupe berie zdrojové súbory jazyka C
  - textový súbor s príponou `.c`
- a vytvorí objektový súbor ako svoj výstup
  - binárny súbor obsahujúci strojový kód
  - ale nemôže byť priamo spustený

Prekladače transformujú ľudsky-čitateľné programy do strojovo-spustiteľných programov. Samozrejme, obe tieto formy programu musia byť uložené v pamäti: prvá má väčšinou formu čistého textu – plain text (zvyčajne kódovaného ako UTF-8, alebo na starších systémoch ako ASCII). V tejto podobe bajty uložené v súbore kódujú ľudsky-čitateľné

písmená.
Na strane výstupu je súbor binárny (čo je v skutočnosti iba všeobecný výraz zastrešujúci súbory, ktoré nie sú plain text), a obsahuje strojom čitateľné inštrukcie – primitívne operácie, ktoré CPU dokáže vykonať. Až na to, že výstup prekladača zatiaľ nie je priamo spustiteľný, napriek tomu, že väčšina inštrukcií už je vo svojej finálnej podobe.
Chýbajúca časť sú adresy: čísla, ktoré popisujú pamäťové miesta v rámci programu samotného (môžu ukazovať na inštrukcie alebo dáta umiestnené priamo v programe). V tejto fáze však ani kód, ani dáta nemajú priradené adresy, takže program nemôže byť spustený (najprv musí byť zlinkovaný, viac sa o tom dozvieme za chvíľu).

Objektové súbory

87

- obsahujú natívny strojový (spustiteľný) kód
- spolu so statickými dátami
  - napr. reťazcové (string) literály použité v programe
- môžu byť rozdelené do viacerých sekcií
  - `.text`, `.rodata`, `.data` a tak ďalej
- a metadátami
  - zoznam symbolov (názvov funkcií) a ich adries

Zmyslom objektových súborov je uložiť tento polo-dokončený strojový kód, spolu so statickými dátami (ako reťazcovými literálmi alebo číselnými konštantami), ktoré sa v programe objavujú. Všetko je to rozdelené do sekcií – obvykle jednej sekcie na strojový kód (tiež nazvanej text a vystupujúcej ako `.text` v objektovom súbore), ďalšej pre dáta výhradne na čítanie (read-only, napr. reťazcové literály), nazvanej `.rodata`, ďalšej sekcie pre modifikovateľné (mutable) ale staticky inicializované premenné – `.data`. K tomuto všetkému sú pribalené metadáta, ktoré popisujú obsah súboru (opäť v strojovo-čitateľnej podobe).
Príkladom takýchto metadát je tabuľka symbolov, ktorá dodáva adresy relatívne voči umiestneniu v súbore funkciám, ktoré boli skompilované do objektového súboru. To znamená, že prekladač vezme definíciu funkcie, ktorú sme napísali v jazyku C a vygeneruje strojový kód pre túto funkciu. Sekcia `.text` v objektovom súbore bude pozostávať z viacerých takýchto funkcií, jedna za druhou: tabuľka symbolov nám potom povie, kde každá z funkcií začína.

Formáty objektových súborov

88

- `a.out` – prvý UNIX-ový formát objektových súborov
- COFF – Common Object File Format
  - pridal podporu pre sekcie nad rámec `a.out`
- PE – Portable Executable (MS Windows)
- Mach-O – Mach Microkernel Executable (macOS)
- ELF – Executable and Linkable Format (všetky moderné UNIX-ové systémy)

Existuje niekoľko rôznych fyzických usporiadaní (layout) objektových súborov, pričom každé má mierne odlišnú sémantiku. Výrazne najrozšírenejším formátom používaným v POSIX-ových systémoch je ELF. Ďalšie rozšírené formáty, ktoré sa v súčasnosti používajú, sú PE (používaný v operačných systémoch MS) a Mach-O (používaný v operačných systémoch Apple).

Archívy (Statické knižnice)

89

- statické knižnice v UNIXe sa volajú archívy
- preto majú príponu `.a`
- je to niečo ako `zip` súbor plný objektových súborov
- plus tabuľka symbolov (názvy funkcií)

Archív je najjednoduchší spôsob, ako spolu zabaliť niekoľko objektových súborov. Ako naznačuje jeho názov, je to v podstate kolekcia objektových súborov, uložená ako jeden súbor. Každý objektový súbor si zachováva svoju identitu a jeho obsah sa žiadnym spôsobom nemení, keď je zabalený do archívu.
Jediný rozdiel oproti typickému dátovému archívu (napr. `tar` alebo `zip` archívu) je, že okrem samotných objektových súborov archív obsahuje dodatočnú sekciu s metadátami – tabuľku symbolov, alebo skôr index symbolov. Ak niekto (typicky linker) potrebuje nájsť definíciu konkrétnej funkcie (symbolu), najprv sa môže pozrieť do indexu pre celý archív, aby zistil, ktorý objektový súbor tento symbol poskytuje. Linkovanie je tým efektívnejšie, keďže linker nemusí sekvenčne prechádzať každý objektový súbor v archíve, aby definíciu našiel.

Linker

90

- objektové súbory sú nekompletné
- môžu odkazovať na symboly, ktoré nedefinujú
  - definície môžu byť v knižniciach
  - alebo v iných objektových súboroch
- linker zlepí niekoľko objektových súborov dokopy
  - aby vytvoril jeden spustiteľný súbor
  - alebo prípadne zdieľanú knižnicu

Ako už bolo spomenuté, za spojenie objektových súborov (a knižníc) do spustiteľných súborov je zodpovedný linker. Ide o pomerne zložitý proces, takže si jeho popis rozdelíme na niekoľko slidov. Vstupom linkeru je niekoľko objektových súborov a výstupom je jediný spustiteľný súbor (executable) alebo niekedy jediná zdieľaná knižnica.
Napriek tomu, že k archívom linker pristupuje špeciálne, objektové súbory, ktoré sú dané linkeru priamo, budú vždy súčasťou výsledného spustiteľného súboru. Objektové súbory sprostredkované skrze archívy budú použité, iba ak poskytujú symboly potrebné pre zostavenie výslednej binárky.

Symboly vs Adresy

91

- používame symbolické názvy na volanie funkcií atď.
- ale strojová inštrukcia `call` potrebuje adresu

- spustiteľný súbor bude nakoniec žiť v pamäti
- dáta a inštrukcie musia dostať adresy
- čo robí linker je, že im tieto adresy pridelí

Hlavnými entitami, ktoré vystupujú v linkovaní sú symboly a adresy. V programe sú strojový kód a dáta načítané do pamäte a, ako vieme, každé pamäťové miesto musí mať adresu. Program vo svojej preloženej (skompilovanej) forme môže používať adresy, aby sa odkazoval na svoje časti. Napríklad na zavolanie podprogramu dodáme jeho začiatočnú adresu špeciálnej inštrukcii `call`, ktorá povie CPU, aby začal vykonávať kód od tejto adresy.
Keď však píšu programy ľudia, nepriraďujú adresy dátam, funkciám alebo jednotlivým inštrukciám. Namiesto toho, ak potrebujeme, aby program odkazoval na svoju časť, dáme týmto častiam názov: tieto názvy sú známe ako symboly. Je zdieľanou zodpovednosťou prekladača a linkeru priradiť adresy jednotlivým symbolom, a to tak, aby nedochádzalo u objektov uložených v pamäti ku konfliktom (aby sa neprekrývali).
Ak sa nad tým zamyslíte, bolo by veľmi zložité robiť to ručne: zvyčajne nevieme, aký dlhý bude strojový kód pre danú funkciu, a museli by sme si tipnúť a potom pridať medzery, pre prípad, že potrebujeme do funkcie nejaký kód pridať, a tak ďalej. Je to veľmi nepríjemné a dokonca aj programátori v assembly sa vyhýbajú ručnému prideľovaniu adries.

Vskutku, jednou z hlavných úloh assembleru je preložiť symbolické adresy na číselné. Ale to som odbočil od témy.

Dohľadanie a priradenie symbolov

92

- linker spracúva naraz jeden objektový súbor
- udržiava si tabuľku symbolov
  - mapovanie symbolov (názvov) na adresy
  - dynamicky aktualizované počas spracovania objektov
- relokácie sú typicky spracované všetky naraz až na konci
- vyriešenie symbolov = nájdenie ich adresy

Linker funguje spôsobom, že si udržiava 'nekompletnú binárku' a postupne do tohto priebežného súboru pripája jednotlivé objektové súbory. Stratégia pre priradenie výsledných adries je jednoduchá: existuje jedna výsledná sekcia `.text`, jedna výsledná sekcia `.data`, atď. Keď je vstupný súbor spracovávaný, jeho vlastná sekcia `.text` je jednoducho prilepená za už spracovaný `.text`. Rovnaký proces je zopakovaný pre každú sekciu.
Tabuľky symbolov vstupných objektových súborov sú obdobne zlučované po jednej a adresy sú upravené počas pridávania symbolov. Okrem definícií symbolov obsahujú objektové súbory použitia symbolov – tieto sú známe ako relokácie a sú uložené v tabuľke relokácií (relocation table). Relokácie obsahujú adresu inštrukcie, ktorá má byť upravená (patched) a symbol, ktorého adresu tam treba dosadiť. Podobne ako u sekcií samotných a u tabuľky symbolov, tabuľka relokácií je budovaná postupne.
Relokácie sú tiež spracované linkerom: zvyčajne to znamená zapísanie finálnej adresy konkrétneho symbolu do dovtedy nekompletnej inštrukcie alebo do premennej v dátovej sekcii. Toto sa zvyčajne deje až keď je výsledná tabuľka symbolov kompletná.
Tabuľka relokácií a tabuľka symbolov sú často na konci zahodené (ale môžu byť v niektorých prípadoch vo výslednom súbore zachované – tabuľka symbolov častejšie ako tabuľka relokácií).

Spustiteľný súbor

93

- hotový obraz (image) programu, ktorý má byť spustený
- obvykle v rovnakom formáte ako objektové súbory
- ale už kompletný, s vyriešenými symbolmi
  - ale: môže používať zdieľané knižnice
  - v tom prípade niektoré symboly zostávajú nepriradené

Výstupom linkeru je obvykle spustiteľný súbor (tzv. executable). Je to súbor založený na rovnakom formáte ako objektové súbory daného operačného systému, ale je v určitom zmysle kompletný. V statických spustiteľných súboroch (takých, ktoré nepoužívajú zdieľané knižnice) sú už všetky referencie a relokácie vyriešené a program môže byť načítaný do pamäte a priamo spustený CPU, bez akýchkoľvek ďalších nastavení.
Stojí za to zmieniť, že adresy, ktoré spustiteľný súbor používa keď referuje na nejaké svoje časti sú virtuálne adresy (tiež je to prípad zdieľaných knižníc nižšie). Na tieto sa bližšie pozrieme v neskoršej prednáške, teraz nám stačí povedať, že to znamená, že rôzne programy na rovnakom operačnom systéme môžu používať prekrývajúce sa adresy pre svoje inštrukcie a dáta. Nevytvára to problém, lebo virtuálne adresy sú súkromné pre každý proces, a tým aj pre každú kópiu každého spusteného programu.

Zdieľané knižnice

94

- každá zdieľaná knižnica potrebuje byť v pamäti iba raz
- zdieľané knižnice používajú symbolické názvy (ako objektové súbory)
- v OS je „mini linker" na vyriešenie týchto názvov
  - väčšinou známy ako runtime / behový linker
  - vyriešenie = nájdenie adries
- zdieľané knižnice môžu používať ďalšie zdieľané knižnice
  - môžu formovať DAG (Directed Acyclic Graph) – orientovaný acyklický graf

Nevýhoda statických knižníc je, že musia byť načítané osobitne (často v mierne odlišných verziách) spolu s každým programom, ktorý ich používa: dokonca, keďže ich linker vložil do programu, sú jeho neoddeliteľnou súčasťou.
Ako sme už skôr spomenuli, nie je to veľmi efektívne. Namiesto toho môžeme uložiť kód knižnice do osobitných súborov (pripomínajúcich spustiteľné súbory), ktoré budú načítané do adresného priestoru programov, ktoré ich potrebujú. Samozrejme, relokácie v hlavnom programe, ktoré odkazujú na symboly zo zdieľaných knižníc (a opačne) a tiež relokácie v zdieľaných knižniciach, ktoré odkazujú do ďalších zdieľaných knižníc, musia byť dohľadané (resolved). Väčšinou sa to deje buď keď je program prvýkrát načítaný do pamäte alebo tzv. lenivým spôsobom, keď je relokácia prvýkrát použitá.
Každopádne musí existovať program, ktorý tieto relokácie dohľadá: ide o runtime linker (behový) – napohľad je podobný normálnemu linkeru, ktorý pracuje v čase prekladu, ale v skutočnosti je dosť odlišný.

Adresy

95

- keď spustíte program, je načítaný do pamäte
- časti programu odkazujú na iné časti programu
  - to znamená, že potrebujú vedieť kde bude načítaný
  - toto má na starosti linker
- zdieľané knižnice používajú pozične nezávislý kód – PIC
  - funguje nezávisle od začiatočnej adresy kde je načítaný
  - nepôjdeme do detailov ako je to implementované

Spomenuli sme, že spustiteľné súbory a knižnice používajú virtuálne adresy na odkazovanie na svoje vlastné časti. To však nepomáha so zdieľanými knižnicami toľko ako so spustiteľnými súbormi. Problém je, že chceme načítať rovnakú knižnicu spolu s niekoľkými programami: ale keď sú adresy použité v knižnici fixné, znamená to, že knižnica musí byť načítaná na rovnakej začiatočnej adrese do každého programu, ktorý túto knižnicu používa. To sa rýchlo stane nepraktickým keď pridáme do systému viac knižníc – žiadne dve knižnice by sa nemohli prekrývať a žiadna by sa nemohla prekrývať s akýmkoľvek spustiteľným súborom.
Čo robíme v praxi je, že prekladáme knižnice tak, aby nepoužívali absolútne adresy keď sa odkazujú na svoje časti. Často to pridáva réžiu za behu, ale umožňuje to načítať knižnicu v ľubovoľnom rozsahu adries, ktoré má aktuálny proces k dispozícii. Výrazne to uľahčuje prácu behovému linkeru.

Prekladač, Linker &c.

96

- prekladač jazyka C sa obvykle volá `cc`
- linker je známy ako `ld`
- správca archívov (statických knižníc) je `ar`
- runtime (behový) linker sa často nazýva `ld.so`

Na mnohých UNIX-och je prekladač a linker k dispozícii ako súčasť samotného systému. Názvy príkazov sú štandardizované.

Časť 2.4: API založené na súboroch (File-Based)

Na POSIX-ových systémoch je API pre používanie súborového systému veľmi dôležité, keďže tiež sprostredkúva prístup ku veľa ďalším prostriedkom, ktoré vystupujú ako 'abstraktné' (špeciálne) súbory v systéme.

Všetko je súbor

98

- súčasť návrhovej filozofie UNIXu
- adresáre sú súbory
- zariadenia sú súbory
- rúry sú súbory
- sieťové spojenia sú (takmer) súbory

Súbor je abstrakcia: je to objekt, z ktorého môžeme čítať bajty a do ktorého môžeme zapisovať bajty (nie všetky súbory nám dovolia robiť oboje). V bežných súboroch môžeme čítať a zapisovať na akomkoľvek offsete, a keď niečo zapíšeme, môžeme neskôr rovnakú vec prečítať (pokiaľ nebola medzičasom prepísaná).
Adresáre fungujú podobne: môžeme z nich čítať bajty, aby sme zistili, ktoré súbory sú prítomné v danom adresári a ako ich nájsť v súborovom systéme. Môžme vytvoriť nové záznamy zapisovaním do adresára. Toto nie je úplne spôsob, akým to obvykle funguje, ale nie je ťažké predstaviť si, že by mohlo.
Dosť veľa zariadení (periférií) sa takto chová: rôzne typy pevných diskov (proste veľká zbierka bajtov), tlačiarne (zapíšte bajty pre ich tlač), skenery (zapíšte bajty pre poslanie príkazov, prečítajte bajty pre dáta obrazu), audio zariadenia (prečítajte bajty z mikrofónov, zapisujte bajty do reproduktorov), a tak ďalej.
Rúry (pipes) sú na tom rovnako: jeden program bajty zapisuje, ďalší ich číta. A sieťové spojenia sú viac-menej iba rúry, ktoré fungujú cez sieť.

Prečo je všetko súbor

99

- znovu-použitie (re-use) obsiahleho API súborového systému
- využitie existujúcich nástrojov príkazového riadku založených na súboroch
- chyby sú zlé → jednoduchosť je dobrá
- chcete tlačiť? `cat file.txt > /dev/ulpt0`
  - (realita je trochu zložitejšia)

Keďže už máme API, ktoré pracuje s abstraktnými súbormi (keďže aj tak potrebujeme pracovať s reálnymi súbormi), je rozumné opýtať sa, či nemôžeme použiť toto existujúce API na prácu s ďalšími objektmi, ktoré vyzerajú ako súbory. Dáva to zmysel nielen na úrovni funkcií jazyka C, ale aj na úrovni programov na príkazovom riadku. Všeobecne, opakované použitie existujúcich mechanizmov robí veci flexibilnejšími, a často tiež jednoduchšími. Samozrejme s tým súvisia určité prekážky (zariadenia často potrebujú podporovať operácie, ktoré sa ťažko mapujú na čítanie alebo zapisovanie bajtov, sokety sú tiež problematické).

Čo je to súborový systém?

100

- zbierka súborov a adresárov
- zvyčajne žije na jednom blokovom zariadení
  - ale tiež môže byť virtuálny
- adresáre a súbory tvoria strom
  - adresáre sú vnútorné uzly
  - súbory sú listové uzly

Napriek tomu, že máme slušnú predstavu, čo je súbor, čo tak súborový systém? Súborový systém je kolekcia súborov a adresárov, typicky uložená na jednom blokovom zariadení. Adresáre a súbory tvoria strom (aspoň do doby, kým do toho nevstúpia symbolické odkazy, kedy sa veci začnú kaziť). Bežné súbory (regular files) vždy reprezentujú listové uzly v tomto strome.

Cesty súborov (File Paths)

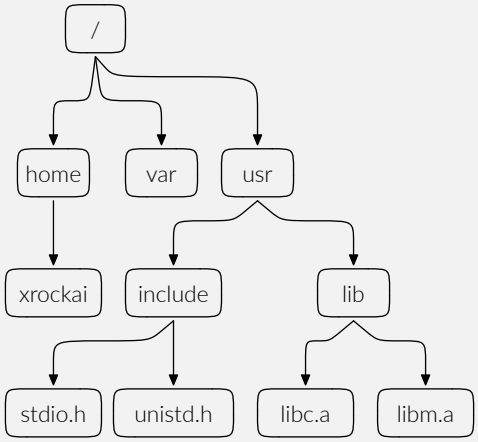
101

- súborové systémy používajú cesty pre ukazovanie na súbory
- reťazec s / ako adresárovým oddeľovačom
  - oddeľovač je \ na Windowse
- úvodný / reprezentuje koreň súborového systému (root)
- napr. `/usr/include`

Cesty sú spôsob, akým odkazujeme na súbory a adresáre v strome. Najvyšší (top-level / root) adresár sa volá /. Každý adresárový záznam (directory entry) nesie meno (a odkaz na vlastný súbor alebo adresár, ktorý reprezentuje) – toto meno môže byť použité v ceste pre odkazovanie sa na danú entitu. Takže ak máme cestu `/usr/include`, začíname v koreňovom adresári (vedúce lomítko), potom v tomto adresári hľadáme entitu, ktorá sa volá `usr` a keď ju nájdeme, skontrolujeme, že opäť ide o adresár. Ak to súhlasí, potom sa pozrieme opäť na jeho priamych následníkov a hľadáme entitu s názvom `include`.

Hierarchia súborov

102



Toto je príklad stromu súborového systému. Môžete si vyskúšať vyhľadávať rôzne cesty v strome, použitím algoritmu popísaného vyššie.

Úloha súborov a súborových systémov

103

- veľmi dôležitá v Plan9
- centrálna na väčšine UNIX-ových systémov
  - cf. Linux pseudo-filesystémy
  - /proc poskytuje informácie o všetkých procesoch
  - /sys sprostredkúva informácie o kerneli a zariadeniach
- pomerne redukované na Windowse
- potlačené na Androide (a ešte viac na iOS)

Rôzne operačné systémy kladú rôzny dôraz na súborový systém. My si ako základnú mieru určíme spôsob, akým k súborovému systému pristupuje POSIX – v tomto prípade je súborový systém pomerne ústredný, nad rámec bežných súborov a adresárov sa v súborovom systéme vyskytuje množstvo špeciálnych súborov, ktoré poskytujú prístup k rôznym ďalším zariadeniam a funkciám operačného systému. Existuje však aj veľa služieb a API, ktoré nie sú založené na súborovom systéme, vrátane napr. správy procesov, správy pamäte atď. V mnohých UNIX-ových systémoch je spoliehanie sa na súborové API prehnané: napr. správa procesov sa robí prostredníctvom virtuálneho súborového systému /proc (na veľa rôznych systémoch), alebo hľadanie (device discovery) a konfigurácia zariadení skrze /sys (Linux). Ďalšou úrovňou nad tým je Plan9, kde v podstate všetko, čo šlo pretvoriť na súborový systém aj bolo. Ďalší experimentálny systém, GNU/Hurd, má podobné ambície.
Ak ideme od POSIXu opačným smerom, máme natívne API Windowsu, ktoré zdôrazňujú súborový systém výrazne menej, než je typické v POSIXe. Väčšina objektov má dedikované API, dokonca aj keď sú pomerne podobné súborom. Súborový systém je však stále výrazne prítomný v API aj v užívateľskom rozhraní. Obe sú ďalej potlačené v 'redukovaných' operačných systémoch ako Android a iOS (hoci oba sú interne kompatibilné s POSIX-om, 'normálnym' aplikáciám nie je dovolené pristupovať k POSIX API, alebo súborovému systému, a obvykle je tiež skrytý pred užívateľmi).

API súborového systému

104

- otvoríte súbor (použitím systémového volania `open()`)
- môžete čítať (`read()`) a zapisovať (`write()`) dáta
- zatvárate (`close()`) súbor, keď ste s ním skončili
- môžete premenovať (`rename()`) a rušiť (`unlink()`) súbory
- môžete použiť `mkdir()` na vytvorenie adresárov

Ako teda vyzerá API súborového systému (filesystem API) na POSIX-ových systémoch? Pre prácu so súborom ho musíte najprv obvykle otvoriť (open): dodáte cestu a nejaké príznaky (flags), kde poviete operačnému systému, čo so súborom plánujete robiť. O tom viac za chvíľu. Keď máte súbor otvorený, môžete z neho čítať (read) a zapisovať do neho (write) dáta. Keď skončíte, použijete close, aby ste uvoľnili príslušné zdroje. Pre prácu s adresármi ich obvykle nepotrebujete otvárať (aj keď môžete). Môžete premenovať súbory (toto je operácia adresára) použitím rename, odstrániť ich z hierarchie súborového systému použitím unlink (toto vymaže príslušný záznam v adresári) a môžete vytvárať nové adresáre použitím mkdir.

Popisovače súborov

105

- kernel si udržiava tabuľku otvorených súborov
- popisovač súboru (file descriptor) je index do tejto tabuľky
- všetko robíte cez popisovače súborov
- nie-Unixové systémy majú podobné koncepty
  - napr. file handle na Windowse

Pamätáte sa na open? Keď chceme pracovať so súborom, potrebujeme spôsob, ako identifikovať tento súbor, a cesty v tomto nie sú úplne vyhovujúce: niekto by mohol premenovať súbor, s ktorým sme pracovali a zrazu by zmizol, alebo ešte horšie, súbor by mohol byť prepísaný iným súborom, alebo dokonca adresárom. Navyše, hľadať súbor prostredníctvom jeho cesty je pomerne drahá operácia: OS musí prečítať každý adresár spomenutý v jeho ceste a vyhľadávať v ňom. Hoci je táto informácia často kešovaná v RAM, stále zaberá vzácny čas.
Keď otvoríme súbor, dostaneme naspäť popisovač súboru (deskriptor) – ide o malé celé číslo, a použitím tohto čísla ako indexu do tabuľky dokáže kernel vyhľadať metadáta, ktoré potrebuje (pre vykonanie čítaní a zápisov) v konštantnom čase. Deskriptor je tiež priamo asociovaný so súborom, takže keď je súbor premiestnený alebo dokonca odstránený (je zrušený jeho odkaz) zo súborového stromu, deskriptor stále odkazuje na rovnaký súbor.
Väčšina nie-POSIX-ových API súborových systémov má podobnú myšlienku (niekedy open nevracia číslo ale iný dátový typ, napr. ukazateľ, a niekedy sa táto hodnota volá handle... ale koncept zostáva viac-menej rovnaký).

Bežné súbory (Regular files)

106

- tieto obsahujú sekvenčné dáta (bajty)
- môžu mať vnútornú štruktúru, ale OS je to jedno
- k súborom sú pripojené metadáta
  - napr. kedy boli naposledy upravované
  - kto môže a kto nemôže pristupovať k súboru
- čítate (`read()`) a zapisujete do (`write()`) súborov

Bežný súbor je presne to, čím sa javí. Je to sekvencia bajtov, uložená na energeticky nezávislom zariadení a má so sebou asociované metadáta, ktoré umožňujú vyhľadať všetky jeho dáta v samotných sektoroch na disku. Jednotlivými bajtmi sa operačný systém nezaťažuje. Keď sú dáta zo súboru čítané, operačný systém nahliadne do metadát, aby zistil na ktorých sektoroch disku má dáta hľadať. Keď sú dáta prepísané, deje sa to isté, až na to, že tieto sektory sú prepísané novými dátami. Keď sú dáta pridané na koniec, operačný systém si vyhľadá voľné miesto na disku, potom upraví metadáta súboru, aby ukazovali na (teraz už obsadené) sektory a zapíše tam dáta. Každý súbor má pri sebe uložené zároveň ďalšie metadáta, napríklad komu súbor patrí alebo kedy bol naposledy upravovaný.

Adresáre

107

- zoznam súborov a ďalších adresárov
  - vnútorné uzly stromu súborového systému
  - adresáre dávajú názvy súborom
- môžu byť otvorené rovnako ako súbory
  - ale `read()` a `write()` nie sú dovolené
  - súbory sa vytvárajú pomocou `open()` alebo `creat()`
  - adresáre pomocou `mkdir()`
  - výpis adresára cez `opendir()` a `readdir()`

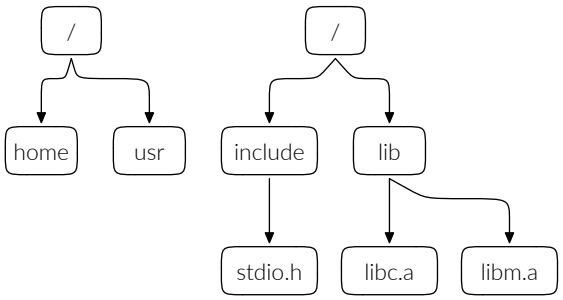
Adresár je (potencionálne) vnútorný uzol v hierarchii súborov: ich úlohou je dať názvy súborom, čím ich sprístupňujú pomocou ciest. Rovnako ako bežné súbory, adresáre sú nezávislé objekty, ale namiesto surových bajtov obsahujú štruktúrované dáta: konkrétne, adresár mapuje názvy súborov na ďalšie súbory (tieto môžu byť bežné súbory, ďalšie adresáre, alebo niektorý zo špeciálnych typov súborov, o ktorých budeme hovoriť za chvíľu).
V princípe by bolo možné implementovať `read` a `write` pre adresáre, ale bolo by to problematické: ak by tieto funkcie manipulovali s reálnou reprezentáciou adresára na disku, užívateľské programy by mohli ľahko poškodiť záznamy v adresároch. To je nežiaduce: adresáre sú namiesto toho za bežných okolností používané prostredníctvom ciest: keď dáme cestu k súboru funkcii `open`, operačný systém bude automaticky prechádzať adresármi ako bude potrebovať.
Samozrejme, užívateľské programy niekedy potrebujú iterovať cez všetky záznamy v adresári, napr. vypísať všetky súbory v danom adresári. Na tento účel POSIX poskytuje funkciu `opendir`, spolu s `readdir`, `seekdir`, `closedir` a tak ďalej. Tieto funkcie poskytujú vysokoúrovňové API pre interakciu s adresármi. Toto API je však výhradne na čítanie: adresárové záznamy sú vytvorené vždy keď sú súbory vytvorené odpovedajúcou cestou, napr. použitím `mkdir` alebo `open` s príznakom `O_CREAT`.

Mounty

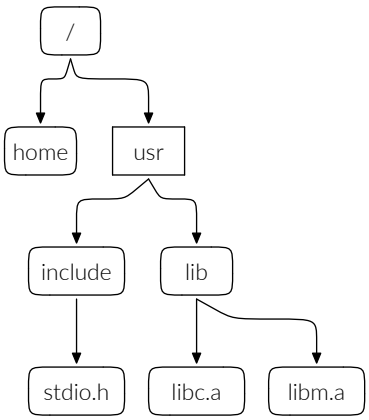
108

- UNIX spája všetky súborové systémy do jedinej hierarchie
- koreň jedného filesystému sa stane adresárom v ďalšom
  - toto sa nazýva mount point
- Windows miesto toho používa písmená diskov (`C:`, `D:` ap.)

Jeden počítač (a teda jeden operačný systém) môže mať k dispozícii viac než jeden pevný disk. V tomto prípade je zvykom, že každé takéto zariadenie má vlastný súborový systém: vyvstáva otázka, ako prezentovať takýchto niekoľko súborových systémov užívateľovi. Stratégia UNIXu je prezentovať všetky súborové systémy v rámci jedného stromového adresára: jeden zo súborových systémov sa zvolí za koreň (root) filesystému: v tomto (a iba v tomto) súborovom systéme je koreňový adresár `/` rovnaký ako systémový koreňový adresár. Všetky ostatné súborové systémy sú spojené s existujúcimi adresármi iných systémov vo svojom koreňovom adresári. Uvážme dva súborové systémy:



Keď teraz pripojíme (mount) druhý súborový systém na adresár `/usr` prvého, dostaneme nasledovnú zjednotenú hierarchiu:



Zvyčajne sú súborové systémy pripojené (mounted) na prázdne adresáre: ak `/usr` nebol prázdny v ľavom (koreňovom) súborovom systéme, jeho obsah by bol skrytý tým, že sme druhý pripojili.
Druhou stratégiou je prezentovať niekoľko súborových systémov použitím viacerých samostatných stromov. Toto je stratégia implementovaná rodinou operačných systémov MS Windows: každému súborovému systému je priradené jedno písmeno, a každý je svojím vlastným samostatným stromom.

Rúry (Pipes)

109

- rúry sú jednoduché komunikačné zariadenie
- jeden program môže zapisovať (`write()`) dáta do rúry
- ďalší program môže čítať (`read()`) tieto rovnaké dáta
- každý koniec rúry dostane popisovač súborov
- rúra môže tiež žiť v súborovom systéme (pomenovaná rúra)

Rúry sú trochu ako súbory, v zmysle, že je možné do nich zapisovať dáta (bajty) a čítať z nich dáta. Vo väčšine prípadov je zapisujúci program odlišný od programu, ktorý dáta číta. Na rozdiel od bežného súboru dáta nie sú nikde permanentne uložené: zmiznú z rúry hneď, ako sú prečítané.
Samozrejme je s rúrou asociovaný buffer, ale je uložený iba v RAM. To umožňuje zapisujúcemu procesu zapisovať dáta aj keď ich práve druhá strana aktívne v rovnakom čase nečíta: OS bude udržiavať zapísané dáta v bufferi, kým druhá strana nebude pripravená si ich prečítať.
Typicky je rúra anonymné zariadenie, prístupné iba cez súborové popisovače. Keď sú tieto zatvorené, zariadenie je zničené. Existuje však aj iný variant rúry, nazvaný pomenovaná rúra, ktorý má názov v rámci súborového systému. Toto neznamená, že dáta sú niekde uložené – pomenovaná rúra funguje presne tak isto ako anonymná rúra, rozdiel je v tom, že môže byť poslaná do `open` pomocou svojej cesty.

Zariadenia

110

- blokové a znakové (character) zariadenia sú (špeciálne) súbory
- k blokovým zariadeniam je pristupované po blokoch – jeden za druhým
  - typické blokové zariadenie je disk
  - vrátane USB veľkokapacitnej pamäte, flash pamäte, atď
  - na blokovom zariadení môžete vytvoriť súborový systém
- znakové zariadenia sú viac ako normálne súbory
  - terminály, kazety, sériové porty, audio zariadenia

Veľa periférnych zariadení vyzerá ako sekvencia bajtov, prípadne ako sekvencia blokov. Typické blokové zariadenie je adresovateľné: užívateľ sa môže posunúť na určité miesto na zariadení a prečítať kus dát

(celočíselný počet blokov). Kým niekto nezapíše na konkrétne miesto na zariadení, čítanie z rovnakej adresy niekoľkokrát vráti zakaždým rovnaké dáta.
Na druhú stranu, znakové zariadenia sa často chovajú ako rúry, v zmysle, že keď program zapíše nejaké bajty do zariadenia, čítanie zariadenia nám nevráti rovnaké bajty. Namiesto toho znakové zariadenia obvykle sprostredkovávajú komunikáciu s periférami, ktoré prijímajú bajty (ktoré sú zapísané do zariadenia) a/alebo poskytujú nejaký výstup (čo je to, čo program dostane, keď zo zariadenia číta). Vezmite si tlačiareň: zapisovanie bajtov do znakového zariadenia tlačiarne spôsobí, že sa tieto bajty vytlačia (po tom, čo ich prípadne tlačiareň spracuje).
Ďalším príkladom by bolo, že potom, čo skener dostal inštrukcie na skenovanie dokumentu, pixely zachytené jeho optickým senzorom môžu byť, v nejakej podobe, získané čítaním z jeho znakového zariadenia. Takže sa v podstate tieto typy znakových zariadení chovajú ako rúra, ale namiesto ďalšieho programu je na druhej strane hardvérové zariadenie (alebo skôr jeho firmvér).

Sokety

111

- API soketov pochádza z raného BSD Unixu
- soket reprezentuje (možné) sieťové spojenie
- sokety sú komplikovanejšie ako normálne súbory
  - vytvorenie spojenia je zložité
  - správy sa strácajú výrazne častejšie, než dáta súborov
- pri otvorení soketu dostanete popisovač súboru (deskriptor)
- môžete čítať (`read()`) a zapisovať do (`write()`) soketov

Sokety sú, v istom zmysle, zovšeobecnenie rúr. V podstate existujú 3 typy soketov:

1. počúvajúci (listening) soket, ktorý umožňuje veľa klientom pripojiť sa k jednému serveru – prísne vzaté, tieto sokety neprenášajú dáta, namiesto toho umožňujú procesom uzavrieť spojenia,
2. pripojený soket, z ktorého jeden je vytvorený pre každé spojenie, a ktorý sa správa v podstate ako obojstranná rúra (pričom štandardné rúry sú jednosmerné),
3. datagramový soket, ktorý môže byť použitý na posielanie dát bez ustanovenia spojenia, použitím špeciálneho send/receive (pošli/prijmi) API.

Zatiaľ čo tretí typ je mierne špeciálny a nechová sa ako rúra, prvé dva sú obvykle použité spolu pre komunikáciu z jedného do mnohých bodov: server počúva na adrese a akýkoľvek klient, ktorý má túto adresu, môže ustanoviť spojenie so serverom. Toto sa líši od rúr, ktoré väčšinou musia byť vopred dohodnuté (teda programy o sebe už musia vedieť).

Typy soketov

112

- sokety môžu byť internetové alebo unixové doménové (unix domain)
  - internetové sokety sa pripájajú k ďalším počítačom
  - Unixové sokety žijú v súborovom systéme
- sokety môžu byť prúdové (stream) alebo datagramové
  - prúdové sokety sú ako súbory
  - môžete do nich zapisovať súvislý tok dát
  - datagramové sokety môžu posielať individuálne správy

Existujú dva základné typy adries: internetové sokety, ktoré sa používajú na komunikáciu medzi rôznymi strojmi (použitím TCP/IP), a unixové doménové sokety, ktoré sa používajú na lokálnu komunikáciu. Unixový soket funguje ako pomenovaná rúra: má cestu v súborovom systéme, ktorú môžu klienti použiť na ustanovenie spojenia so serverom.

Review Questions

113

5. What is a shared (dynamic) library?
6. What does a linker do?
7. What is a symbol in an object file?
8. What is a file descriptor?

Časť 3: Jadro (Kernel)

Táto prednáška je o jadre, najnižšej vrstve operačného systému. Skladá sa z 5 častí:

Obsah prednášky

115

1. privilegovaný režim
2. bootovanie
3. architektúra jadra
4. systémové volania
5. služby poskytované kernelom

Najprv sa pozrieme na režimy procesora a ako sú prepojené s vrstvami operačného systému. Potom sa presunieme na proces bootovania, pretože určitým spôsobom znázorňuje vzťah medzi jadrom a ostatnými komponentami operačného systému, a tiež medzi firmvérom a jadrom. Bližšie sa pozrieme na architektúru jadra: veci, ktoré sme už naznačili v predchádzajúcich prednáškach, a tiež si predstavíme exokernely a unikernely, nad rámec architektúr, ktoré už poznáme (mikro a monolitické kernely).
Štvrtá časť bude zameraná na systémové volania a ich binárne rozhranie – tzn. ako sú reálne systémové volania implementované na strojovej úrovni. Toto úzko súvisí s prvou časťou prednášky o režimoch procesora a nadväzuje na znalosti, ktoré sme získali minulý týždeň o tom, ako systémové volania vyzerajú na úrovni jazyka C.
Nakoniec sa pozrieme na to, aké služby poskytujú jadrá zvyšku operačného systému, aké sú ich povinnosti a tiež sa bližšie pozrieme na to, ako fungujú mikrokernely a hybridné operačné systémy.

Pripomienka: Vrstvy softvéru

116

- → kernel ←
- systémové knižnice
- systémové služby / démoni
- pomocné programy (utility)
- aplikačný softvér

Časť 3.1: Privilegovaný režim

CPU režimy

118

- CPU poskytujú privilegovaný (supervisor) a užívateľský režim
- takto to je u všetkých moderných general-purpose (univerzálnych) CPU
  - nie nutne u mikrokontrolérov
- x86 poskytuje 4 rôzne privilegované režimy (tzv. kruhy – rings)
  - väčšina systémov používa iba ring 0 a ring 3
  - Xen paravirtualizácia používa ring 1 pre guest (hosťujúce) kernely

Existuje mnoho funkcií, ktoré môžu vykonávať iba programy bežiace v privilegovanom režime. To umožňuje kernelom vynucovať hranice medzi užívateľskými programami. Niekedy sú prítomné medzistupne čo sa týka úrovní oprávnení, ktoré umožňujú jemnejšie vrstvenie operačného systému. Napríklad ovládače môžu bežať na menej privilegovanej úrovni než 'jadro' kernelu, čím môžeme zabezpečiť vrstvu ochrany kernelu pred jeho vlastnými ovládačmi zariadení. Možno si pamätáte, že ovládače zariadení sú najproblematickejšou časťou kernelu.
Mimo ovládačov zariadení môžu byť systémy s viacerými vrstvami privilégií procesorov použité na niektorých virtualizačných systémoch. Viac sa o tomto dozvieme v závere semestra.

Privilegovaný režim

119

- veľa operácií je v užívateľskom móde obmedzených
  - takto sú spúšťané užívateľské programy
  - a väčšina operačného systému
- softvér bežiaci v privilegovanom režime môže robiť ~čokoľvek
  - čo je najdôležitejšie, môže programovať MMU
  - kernel beží v tomto režime

Kernel sa spúšťa v privilegovanom režime procesora. V tomto režime je softvéru umožnené urobiť všetko, čo je urobiť možné. Konkrétne, môže (pre)programovať jednotku pre správu pamäti – memory management unit (MMU, pozri ďalší slide). Keďže MMU je spôsob, akým je implementované oddelenie (separation) programov, kód bežiaci v privilegovanom režime môže meniť pamäť akémukoľvek programu bežiacemu na počítači. To vysvetľuje prečo chceme zredukovať množstvo kódu bežiace v privilegovanom (supervisor) režime na minimum.
Pri spôsobe, akým funguje väčšina operačných systémov, je kernel jediná časť softvéru, ktorá má dovolené bežať v privilegovanom režime. Kód v systémových knižniciach, démonoch, atď., spolu s aplikačným softvérom, je obmedzený na užívateľský režim procesora. V tomto režime nemôže byť MMU programovaná a softvér môže robiť iba to, čo mu MMU umožňuje na základe inštrukcií, ktoré dostala od kernelu.

Jednotka pre správu pamäti (MMU)

120

- ide o subsystém procesora
- stará sa o preklad adries
  - užívateľský softvér používa virtuálne adresy
  - MMU ich prekladá na fyzické adresy
- mapovania môžu byť spravované kernelom OS

Poďme sa bližšie pozrieť na MMU. Jej primárnou úlohou je preklad adries. Adresy, na ktoré programy odkazujú sú virtuálne – nekorešpondujú s fixnými fyzickými miestami na pamäťových čipoch. Vždy, keď sa pozriete na, povedzme, ukazateľ v C-čkovom kóde, číselná hodnota tohto ukazateľa je adresa v nejakom virtuálnom adresnom priestore. Úlohou MMU je preložiť túto virtuálnu adresu na fyzickú – ktorá má fixný vzťah voči nejakému fyzickému kondenzátoru alebo inému elektronickému zariadeniu, ktoré si pamätá informáciu.
Spôsob mapovania adries je programovateľný: kernel môže MMU povedať, ako preklad prebieha tým, že mu dodá prekladové tabuľky (translation tables). Ako fungujú tabuľky stránok (page tables) si povieme za krátku chvíľu; čo je teraz dôležité je, že je úlohou kernelu skonštruovať ich a poslať ich MMU.

Stránkovanie

121

- fyzická pamäť je rozdelená do rámcov (frames)
- virtuálna pamäť je rozdelená na stránky (pages)
- stránky a rámce majú rovnakú veľkosť (obvykle 4KiB)
- rámce sú miesta, stránky sú obsah
- tabuľky stránok mapujú stránky na rámce

Než sa dostaneme k virtuálnym adresám, pozrime sa na ďalšie významné použitie mechanizmu prekladu adries, ktorým je stránkovanie (paging). Urobíme to preto, lebo stránkovanie možno lepšie ilustruje ako funguje MMU. V tomto prípade rozdelíme fyzickú pamäť (fyzický adresný priestor) na rámce (frames), ktoré predstavujú oblasti pamäte: miesta, kde môžeme dať dáta a odkiaľ ich neskôr môžeme vyzdvihnúť. Predstavte si to ako poličky v knihovničke.
Virtuálny adresný priestor je potom rozdelený na stránky (pages): reálne kusy dát nejakej fixnej veľkosti. Stránky fyzicky neexistujú, iba reprezentujú nejaké časti, ktoré program potrebuje uložiť. Môžete o stránke premýšľať ako o veľmi veľkom celom čísle (integer). Alebo môžete o stránkach premýšľať ako o zbierke kníh, ktorá sa vojde na jednu policu.
Tabuľka stránok je potom katalóg, alebo nejaká forma adresára (vo význame knihy adries). Programy priradia mená stránkam – knihám – ale hardvér dokáže nájsť iba poličky. Úlohou MMU je vziať meno knihy a nájsť fyzickú policu, kde je kniha uložená. Zjavne má operačný systém dovolené knihy ľubovoľne presúvať, za predpokladu, že udrží tabuľku stránok – katalóg – aktuálny. Zvyšný softvér nespozná rozdiel.

Swapovanie stránok

122

- RAM zvykla byť vzácny zdroj
- stránkovanie umožňuje OS presunúť stránky mimo RAM
  - stránka (obsah) môže byť zapísaná na disk
  - a rámec môže byť použitý na ďalšiu stránku
- nie až tak podstatné pri súčasnom hardvéri
- užitočné pre pamäťovo mapované súbory (viac v ďalšej prednáške)

Ak nám dochádza miesto na polici, možno chceme presunúť nejaké knihy do skladu. Potom môžeme použiť uvoľnenú policu na nejaké ďalšie knihy. Hardvér však dokáže získavať informácie iba z políc, preto keď si program vypýta knihu, ktorá je v danom momente na sklade, operačný systém musí zariadiť veci tak, aby kniha bola presunutá zo skladu na policu, než môže program pokračovať.
Tento proces sa nazýva swapovanie: keď OS dochádza pamäť, vysťahuje stránky z RAM na disk alebo iné veľkokapacitné (ale pomalé) médium. Znovu ich nastránkuje, až keď budú potrebné. V súčasných počítačoch nie je pamäť tak vzácna a tento prípad použitia nie je až taký dôležitý. Umožňuje to však ďalší elegantný trik: namiesto otvárania súboru a

čítania ho použitím systémových volaní open a read môžeme použiť tzv. pamäťovo mapované súbory. To prakticky sprístupňuje obsah súboru ako kus pamäte, ktorý môže byť čítaný a dokonca doň môže byť zapisované a zmeny sú posielané nazad do súborového systému. Na toto sa pozrieme bližšie budúci týždeň.

Načrtnutie: Procesy

123

- proces je primárne definovaný svojím adresným priestorom
  - adresným priestorom chápeme validné virtuálne adresy
- toto je implementované pomocou MMU
- pri zmene procesu je načítaná iná tabuľka stránok
  - to sa nazýva context switch – prepnutie kontextu
- tabuľka stránok určuje, čo proces vidí

Procesmi sa budeme zaoberať neskôr v tomto kurze, teraz chcem len rýchlo uviesť tento koncept, aby sme dokázali oceniť, aká dôležitá je MMU pre moderný operačný systém. Každý proces má svoj vlastný adresný priestor, ktorý popisuje, ktoré adresy sú validné pre daný proces. Ak opomenieme ďalšie obmedzenia, proces môže zapisovať do akejkoľvek zo svojich validných adries a potom nazad prečítať uloženú hodnotu z danej adresy.
Fakt, že adresný priestor procesu je abstraktný a nie je naviazaný na žiadne konkrétne fyzické rozloženie pamäte, je dosť podstatný. Ďalšie dôležité pozorovanie je, že adresný priestor nemusí byť súvislý, a že v adresnom priestore nemusí byť viditeľná všetka fyzická pamäť.

Pamäťové mapy (Memory Maps)

124

- iný pohľad na rovnaké princípy
- OS mapuje fyzickú pamäť do procesu
- viacero procesov môže mať namapovanú rovnakú časť RAM
  - toto sa nazýva zdieľaná pamäť
- často je časť RAM namapovaná iba na jediný proces

Na rovnakú vec sa môžeme pozrieť aj z iného uhla pohľadu. Fyzická pamäť je zdroj, a operačný systém môže 'vydať' časť fyzickej pamäte procesu. Toto sa deje namapovaním tohto kusu pamäte do adresného priestoru procesu. Neexistuje nič, čo by v princípe bránilo operačnému systému namapovať rovnaký fyzický kus RAM do niekoľkých procesov. V tomto prípade sú dáta uložené iba raz, ale ktorýkoľvek z procesov ich môže čítať prostredníctvom adresy vo svojom virtuálnom adresnom priestore (prípadne na inej adrese v každom procese).
Pre 'pracovnú' pamäť – ktorá je čítaná aj zapisovaná programom – je najčastejší prípad, že každá oblasť fyzickej pamäte je namapovaná iba do jedného procesu. Inštrukcie sú zdieľané výrazne častejšie: napríklad zdieľaná knižnica je často namapovaná do niekoľkých rôznych procesov. Vlastný spustiteľný súbor môže podobne byť tiež namapovaný do niekoľkých procesov, ak spúšťajú rovnaký program.

Stránkové tabuľky

125

- MMU je naprogramovaná pomocou prekladových tabuliek
  - tieto tabuľky sú uložené v RAM
  - zvyčajne sa volajú tabuľky stránok (page tables)
- a sú plne v správe kernelu
- kernel môže požiadať MMU aby vymenila tabuľku stránok
  - takto je riešená izolácia procesov

Vlastná implementácia mechanizmu virtuálnej pamäte je známa ako stránkové tabuľky (page tables): ide o prekladové tabuľky (translation tables), ktoré hovoria MMU, ktoré virtuálne adresy sa mapujú na ktoré fyzické adresy. Tabuľky stránok sú uložené v pamäti, rovnako ako akékoľvek iné dáta, a môžu byť vytvorené a zmenené kernelom. Kernel si obvykle udržiava oddelenú sadu tabuliek stránok pre každý proces, a keď dôjde k prepnutiu kontextu, požiada MMU aby nahradila aktívnu tabuľku stránok novou (tou, ktorá patrí procesu, ktorý bude aktivovaný). Obvykle je toto uskutočnené uložením fyzickej adresy prvej úrovne novej tabuľky stránok (v terminológii x86 tzv. adresár stránok – page directory) do špeciálneho registra.
Často je zapisovateľná fyzická pamäť odkazovaná prvou sadou tabuliek stránok neprístupná z druhej sady a naopak. Aj keď sa náhodou prekrývajú, spoločných častí bude pomerne málo (procesy si môžu vyžiadať zdieľanú pamäť na vzájomnú komunikáciu). Takže dáta, ktoré predchádzajúci proces zapísal do svojej pamäte, budú úplne neviditeľné pre nový proces.

Ochrana kernelu

126

- pamäť kernelu je obvykle namapovaná do všetkých procesov
  - na veľa CPU to zvyšuje výkon
  - (aspoň do momentu než prišiel meltdown)
- stránky kernelu majú nastavený špeciálny príznak 'supervisor'
  - kód spúšťaný v užívateľskom móde sa ich nemôže dotýkať
  - inak by užívateľský kód mohol zasahovať do pamäte kernelu

Výmena stránkových tabuliek je zvyčajne pomerne drahá operácia a chceme sa jej čo najviac vyhnúť. Obzvlášť sa jej chceme vyhnúť na ceste systémového volania (systémové volania si zrejme pamätáte z minulého týždňa a ešte o nich dnes budeme hovoriť detailnejšie). Preto je častým trikom namapovať kernel do každého procesu, ale zneprístupniť túto pamäť pre užívateľský kód. Nanešťastie sa vyskytli nejaké CPU chyby, ktoré to robia menej bezpečným, než by sme chceli.

Časť 3.2: Bootovanie

Bootovací proces je sekvencia krokov, ktorá začína s vypnutým počítačom a končí, keď je počítač pripravený interagovať s užívateľom (prostredníctvom operačného systému).

Štartovanie OS

128

- pri zapnutí je systém v predvolenom – default stave
  - hlavne z dôvodu, že RAM je energeticky závislá
- celá platforma musí byť inicializovaná
  - ide predovšetkým o CPU
  - a konzolový hardvér (klávesnica, monitor, ...)
  - potom zvyšok zariadení

Počítače sa dajú vypínať a zapínať (zjavne). Keď sú vypnuté, energia už nie je dostupná a dynamická RAM, bez aktívneho obnovovania, rýchlo zabudne informácie, ktoré udržiavala. Takže keď počítač zapneme, v RAM nič nie je, CPU je v nejakom default stave a niečo podobné platí prakticky pre všetky čiastkové zariadenia v počítači. Mimo obsah trvalých úložísk je počítač v stave v akom opustil fabriku. Počítač v tomto stave, aby sme si to povedali na rovinu, nie je veľmi užitočný.

Proces bootovania

129

- proces začína vstavanou inicializáciou hardvéru
- keď je pripravený, hardvér odovzdá riadenie firmvéru
  - na 16 a 32 bitových systémoch to bol BIOS
  - nahradený EFI na súčasných amd64 platformách
- firmvér potom načíta bootloader
- bootloader načíta kernel

Do hardvérovej časti sekvencie sa nebudeme púšťať. Prepne sa spínač, hardvér sa naštartuje a urobí si svoje. V nejakom bode to prevezme firmvér a urobí ďalšie veci. Hardvér a firmvér sa nakoniec dostanú do stavu, kedy môžu začať načítať operačný systém. Obvykle existuje časť softvéru, ktorú firmvér načíta z trvalej pamäte, nazvaná bootloader. Tento bootloader je viac-menej súčasť operačného systému: jeho úlohou je nájsť a načítať kernel (z trvalého úložiska, zvyčajne použitím služieb firmvéru na identifikáciu tohto úložiska a načítanie dát z neho). Môže a nemusí rozumieť súborovým systémom a podobným vysokoúrovňovým záležitostiam. V najjednoduchšom prípade má bootloader zoznam diskových blokov, v ktorých je kernel uložený, a tieto si vyžiada od firmvéru. V moderných systémoch sú ako firmvér tak bootloader pomerne sofistikované, a rozumejú zložitým, vysokoúrovňovým veciam (vrátane napr. šifrovaných diskov).

Proces bootovania (pokračovanie)

130

- kernel potom inicializuje ovládače zariadení
- a koreňový súborový systém
- potom odovzdá riadenie procesu `init`
- v tomto bode prevezme zodpovednosť užívateľský priestor

Konečne sa dostávame do známych oblastí. Bootloader načítal kernel do RAM a skočil na pred-nastavenú adresu v obraze kernelu (kernel image). Inštrukcie uložené na tejto adrese naštartujú sekvenciu inicializácie kernelu (kernel initialization sequence). Jej prvá časť je zvyčajne ešte stále pomerne nízkoúrovňová: dostane CPU a nejaké ďalšie periférie (konzoly, časovače, atď.) do stavu, aby ich mohol operačný systém používať. Potom odovzdá kontrolu C-čkovému kódu, ktorý nastaví základné dátové štruktúry používané kernelom. Potom kernel začne inicializovať jednotlivé periférne zariadenia – táto úloha je vykonávaná konkrétnymi ovládačmi zariadení. Keď sú periférie inicializované, kernel sa môže začať pozerať po koreňovom súborovom systéme (root filesystem) – obvykle je uložený na jednom z pripojených energeticky nezávislých úložných zariadení (ktoré v tomto bode bude funkčné a prístupné kernelu prostredníctvom svojich ovládačov zariadení).
Po pripojení koreňového súborového systému dokáže kernel nastaviť prázdny proces, načítať program `init` do tohto procesu a prenechať mu riadenie. V tomto bode sa prestáva kernel chovať ako sekvenčný program s funkciou `main` a ustúpi do úzadia: odteraz sú všetky akcie riadené procesmi z užívateľského priestoru (alebo hardvérovými prerušeniami, ale o tých sa budeme baviť výrazne neskôr).

Inicializácia užívateľského režimu

131

- `init` pripojí zvyšok súborového systému
- proces `init` spustí užívateľské systémové služby
- potom spustí aplikačné služby
- a napokon proces prihlasovania (`login`)

Ani zďaleka sme neskončili. Teraz proces `init` potrebuje nájsť všetky ostatné súborové systémy a pripojiť ich (mount), spustiť množstvo systémových služieb a prípadne nejaké aplikačné služby (démonov, ktorí nie sú súčasťou operačného systému – veci ako webové servery).
Keď sú všetky nevyhnutné systémy pripravené, `init` začne proces prihlasovania (`login`), ktorý prezentuje užívateľovi známu prihlasovaciu obrazovku a žiada ho, aby zadal svoje meno a heslo. V tomto bode je proces bootovania ukončený, ale pozrieme sa v rýchlosti na ešte jeden krok.

Po prihlásení

132

- `login` proces zaháji tzv. user session (užívateľskú reláciu)
- načíta moduly pracovnej plochy (desktop) a aplikačný softvér
- zanechá užívateľa v (textovom alebo grafickom) shell-i
- teraz môžete začať používať počítač

Keď sa užívateľ prihlási, začne ďalšia inicializačná sekvencia: systém potrebuje nastaviť užívateľovi tzv. session (reláciu). Toto opäť zahŕňa nejaké kroky, ale nakoniec je konečne možné interagovať s počítačom.

CPU Init

133

- závisí na architektúre ako aj na platforme
- na x86 CPU štartuje v 16-bitovom režime
- na pôvodných systémoch zostali BIOS & bootloader v tomto režime
- kernel sa potom prepne do chráneného režimu počas svojho bootovania

Vráťme sa na začiatok a doplňme nejaké dodatočné detaily. Na úvod, aký je stav CPU počas bootovania, a prečo musí čokoľvek robiť operačný systém? Súvisí to so spätnou kompatibilitou: CPU zvyčajne štartuje v čo najkompatibilnejšom režime – v prípade 32b x86 procesorov je toto v 16b režime s deaktivovanou MMU. Keďže celá platforma zachováva spätnú kompatibilitu, firmvér udržiava CPU v tomto režime a je úlohou buď bootloadera alebo kernelu samotného to napraviť. Nie je to tak vždy (moderné 64b x86 procesory stále štartujú v 16b režime, ale firmvér ich prestaví do tzv. long mode – dlhého režimu, teda 64 bitového – než kontrolu prevezme bootloader).

Bootloader

134

- historicky limitovaný na desiatky kilobajtov kódu
- bootloader lokalizuje kernel na disku
  - môže umožniť užívateľovi zvoliť si rôzne kernely
  - obmedzené porozumenie súborovým systémom
- potom načíta obraz kernelu do RAM
- a odovzdá riadenie kernelu

Bootloader je krátky, platformovo-závislý program, ktorý načítava kernel z energeticky nezávislej (stálej) pamäte (väčšinou zo súborového systému na disku) a odovzdá riadenie kernelu. Bootloader môže urobiť nejakú veľmi základnú inicializáciu hardvéru, ale väčšina inicializácie je robená priamo samotným kernelom v neskoršom štádiu.

Moderné bootovanie na x86

135

- bootloader v súčasnosti beží v chránenom režime
  - alebo dokonca v dlhom režime (long mode) na 64-bitových CPU
- firmvér rozumie FAT súborovému systému
  - dokáže odtiaľ načítať súbory do pamäte
  - toto značne uľahčuje proces bootovania

Proces bootovania bol v poslednej (cca) dekáde značne zjednodušený na x86 počítačoch. Do štandardizovaného rozhrania firmvéru boli pridané API na podstatne vyššej úrovni, čím sa kód zodpovedný za bootovanie výrazne zjednodušil.

Bootovanie ARM

136

- na ARM systémoch neexistuje jednotné firmvérové rozhranie
- U-boot je najbližšie ako sa dá dostať k zjednoteniu
- bootloader potrebuje nízkoúrovňovú znalosť hardvéru
- to robí písanie bootloaderov pre ARM pomerne pracným
- súčasný U-boot môže používať EFI protocol z PC

Na rozdiel od x86 sveta, ekosystém ARM je výrazne menej štandardizovaný a každý systém na čipe (system on a chip – SoC) potrebuje mierne odlišný bootovací proces. Toto je extrémne nepraktické, keďže existujú desiatky SoC modelov od rôznych výrobcov, a nové stále pravidelne vychádzajú. Našťastie, U-boot sa stal de-facto štandardom, a hoci U-boot samotný stále potrebuje byť prispôsobený na každý ďalší SoC alebo dokonca každú dosku, operačný systém je v súčasnosti väčšinou uchránený pred touto zložitosťou.

Časť 3.3: Architektúra kernelu

V tejto sekcii sa pozrieme na rôzne architektúry (návrhy) kernelov: hlavný rozdiel, o ktorom budeme hovoriť, je ktoré služby a komponenty sú súčasťou vlastného kernelu, a ktoré sú už mimo kernelu.

Typy architektúr

138

- monolitické kernely (Linux, *BSD)
- mikrokernely (Mach, L4, QNX, NT, ...)
- hybridné kernely (macOS)
- hypervízory typu 1 (Xen)
- exokernely, rump kernely

Už skôr v priebehu predmetu sme spomenuli dva základné typy kernelov. Tieto typy reprezentujú extrémy hlavných smerov návrhu kernelov: mikrokernely sú najmenším (najviac uzavretým) hlavným smerom, zatiaľ čo monolitické kernely sú najväčšie (najviac inkluzívne/otvorené). Systémy s hybridnými kernelmi sú prirodzeným kompromisom medzi týmito dvomi extrémnymi návrhmi: majú 2 komponenty, mikrokernel a takzvaný superserver, čo je v podstate vykuchaný monolitický kernel – tzn. funkcionalita pokrytá mikrokernelom je vyňatá.
Okrem 'mainstream' návrhov kernelov existuje niekoľko viac exotických možností. Napríklad o hypervízoroch typu 1 (tzv. bare metal – bežiacich priamo na železe) môžeme uvažovať ako o špeciálnom type kernelu operačného systému, kde aplikácie sú jednoducho virtuálne stroje – t.j. 'normálne' operačné systémy (viac o tomto neskôr). Potom sú tu exokernelové operačné systémy, ktoré drasticky zredukovali služby poskytované aplikáciám a unikernely, ktoré sú v podstate knižnice pre spúšťanie celých aplikácií v režime kernelu.

Mikrokernel

139

- stará sa o ochranu pamäte
- (hardvérové) prerušenia
- plánovanie úloh / procesov (scheduling)
- posielanie správ (message passing)

- všetko ostatné je osobitne

Mikrokernel sa stará iba o nevyhnutné služby – také, ktoré sa nedajú rozumne robiť mimo kernel (teda mimo privilegovaný režim CPU). Toto samozrejme zahŕňa programovanie MMU (t.j. správu adresných priestorov a ochranu pamäte), spracovávanie prerušení (tieto prepnú CPU do privilegovaného režimu, takže prinajmenšom začiatočná obslužná rutina prerušenia musí byť súčasťou kernelu), prepínanie vláken a procesov (a typicky aj plánovanie) a nakoniec nejaký mechanizmus pre vzájomnú komunikáciu medzi procesmi (typicky posielanie správ). S týmito komponentami v kerneli môže byť všetko ostatné realizované mimo vlastný kernel (hoci ovládače zariadení potrebujú nejaké dodatočné nízkoúrovňové služby od kernelu, ktoré tu neboli vymenované, ako DMA programovanie a delegovanie hardvérových prerušení).

Monolitické kernely

140

- všetko, čo robí mikrokernel
- plus ovládače zariadení
- súborové systémy, správa zväzkov (volumes) disku
- sieťové vrstvy (network stack)
- šifrovanie dát, ...

Monolitický kernel potrebuje obsahovať všetko, čo robí mikrokernel (hoci niektoré časti majú mierne odlišnú podobu, aspoň obvykle: medziprocesová komunikácia je prítomná, ale môže byť iného typu, integrácia ovládačov vyzerá inak). Na druhú stranu má množstvo dodatočných povinností: veľa ovládačov zariadení (také, ktoré potrebujú prerušenia alebo DMA, alebo majú kritické požiadavky na výkon) je integrovaných do kernelu, rovnako ako súborové systémy a správa zväzkov (volume management) na disku. Úplný TCP/IP "zásobník" (stack) – sústava protokolov – je takmer istý. Niektoré ďalšie časti môžu byť súčasťou kernelu, napríklad kryptografické služby (správa kľúčov, šifrovanie disku, atď.), filtrovanie paketov, kuchynský drez[1] atď. Samozrejme, všetok tento kód beží v privilegovanom režime, a ako taký má úplnú kontrolu nad OS a nad počítačom ako celkom.

Mikrokernel (Opakovanie)

141

- potrebujeme viac, než nám poskytuje mikrokernel
- v "skutočnom" mikrokernelovom OS je prítomných veľa modulov
- každý ovládač zariadenia beží v osobitnom procese
- to isté platí pre súborové systémy a sieťové služby
- tieto moduly / procesy sa volajú servery

Vyvstáva otázka, kto je zodpovedný za všetky služby vymenované na predchádzajúcom slide (tie, ktoré sú súčasťou monolitického kernelu, ale chýbajú v mikrokerneli). V 'skutočnom' mikrokernelovom OS sú tieto služby poskytované osobitne, každá samostatným procesom (v tomto kontexte tiež známe ako servery).

[1] pozn. prekladu: o veciach sa ironicky hovorí, že keď obsahujú všetko, obsahujú aj kuchynský drez

Hybridné kernely

142

- založené na mikrokerneli
- a vykuchanom monolitickom kerneli

- monolitický kernel je veľký server
  - stará sa o veci, ktoré nezabezpečuje mikrokernel
  - jednoduchšie na implementáciu než skutočný mikrokernel OS
  - čo sa týka výkonu je niekde uprostred

V hybridnom kerneli je väčšina služieb poskytovaná jedným veľkým serverom, ktorý je do istej miery izolovaný od hardvéru. Často je tento server založený na kerneli monolitického OS, s odstránenými najnižšími vrstvami, ktoré sú nahradené volaniami do mikrokernelu.
Hybridné kernely sú lacnejšie na navrhnutie a teoreticky dosahujú lepší výkon než 'pravé' (viac-serverové) mikrokernelové systémy.

Mikro vs Mono

143

- mikrokernely sú robustnejšie
- monolitické kernely sú efektívnejšie
  - menej prepínania kontextu
- je diskutabilné, čo je jednoduchšie na implementáciu
  - výhľadovo vyhráva monolitické jadro
- hybridné kernely sú kompromis

Hlavnou výhodou mikrokernelov je ich robustnosť voči softvérovým chybám. Keďže kernel samotný je malý, šanca, že obsahuje chybu vo vlastnom kerneli je výrazne nižšia v porovnaní s relatívne obrovským množstvom kódu monolitického kernelu. Následky chýb mimo kernelu (v serveroch) sú výrazne menšie, keďže sú izolované od zvyšku systému a, hoci poskytujú dôležité služby, systém sa často dokáže spamätať z poruchy reštartovaním chybného serveru.
Na druhú stranu, monolitické kernely poskytujú lepší výkon, najmä vďaka zredukovanému prepínaniu kontextu, ktoré je stále pomerne drahé, aj na moderných procesoroch schopných virtualizácie. Pri tom ako monolitické kernely adoptujú technológie ako izolácia stránkovacej tabuľky kernelu, kvôli zvýšeniu bezpečnosti, sa rozdiel vo výkone zmenšuje.
Čo sa týka implementácie, monolitické kernely majú dve výhody: v mnohých prípadoch môže byť kód písaný v priamom, synchrónnom štýle, a rôzne časti kernelu môžu zdieľať dátové štruktúry bez ďalšieho úsilia. Na rozdiel od toho, pravý viac-serverový systém často musí používať asynchrónnu komunikáciu (posielanie správ) na dosiahnutie rovnakých výsledkov, čím sa kód stáva zložitejším na napísanie a pochopenie. Z dlhodobého hľadiska môže lepšia modularita a izolácia komponentov prevážiť krátkodobé zisky v efektivite programovania vďaka priamemu programovaciemu štýlu.

Exokernely

144

- menšie než mikrokernel
- výrazne menej abstrakcií
  - aplikácie majú k dispozícii iba blokové úložisko
  - sieťové služby sú značne zredukované
- existujú iba vedecké systémy

Operačné systémy založené na mikrokerneloch stále poskytujú svojím aplikáciám plnú sadu služieb, vrátane súborových systémov, sieťových vrstiev, atď. Rozdiel je iba v tom, kde je táto funkcionalita implementovaná, buď vo vlastnom kerneli, alebo v serveri na užívateľskej úrovni.

Pri exokerneloch toto už nie je pravda: služby poskytované operačným systémom sú značne osekané. Výsledný systém je niečo medzi paravirtualizovaným počítačom (o tomto koncepte si viac povieme v závere kurzu) a 'štandardným' operačným systémom. Na rozdiel od virtuálnych strojov (a unikernelov) je stále k dispozícii izolácia aplikácií založená na procesoch, ktorá zohráva dôležitú úlohu. V súčasnosti neexistujú žiadne komerčné systémy tohto typu.

Hypervízory typu 1

145

- tiež známe ako bare metal (bežiace priamo na železe) alebo natívne hypervízory
- pripomínajú mikrokernelové operačné systémy
  - alebo exokernely, v závislosti od uhla pohľadu
- "aplikácie" pre hypervízor sú operačné systémy
  - hypervízory môžu používať hrubšie abstrakcie než OS
  - celé úložné zariadenia miesto súborového systému

Bare metal hypervízor je podobný OS na báze exokernelu alebo mikrokernelu (v závislosti na konkrétnom hypervízore a na našom uhle pohľadu). Typicky hypervízor poskytuje rozhrania a zdroje, ktoré sú tradične implementované v hardvéri: blokové zariadenia, sieťové rozhrania, virtuálne CPU, vrátane virtuálnej MMU, ktorá umožňuje 'aplikáciám' (t.j. hosťovským operačným systémom) využívať stránkovanie.

Unikernely

146

- kernely pre spúšťanie jedinej aplikácie
  - na reálnom hardvéri to nedáva veľmi zmysel
  - ale môže to byť veľmi užitočné na hypervízore
- pribaľte aplikácie ako virtuálne stroje
  - bez réžie univerzálneho (general-purpose) OS

Unikernely patria do inej vetvy (v porovnaní s exokernelmi) minimalistického návrhu operačných systémov. V tomto prípade paralelné spracovávanie úloh na úrovni procesov a izolácia adresného priestoru nie sú súčasťou kernelu: kernel existuje na podporu jedinej aplikácie, tým že jej dodáva (podmnožinu) tradičných OS abstrakcií ako sieťové vrstvy, hierarchický súborový systém, atď. Keď je aplikácia zabalená s kompatibilným unikernelom, výsledok môže byť priamo spustený na hypervízore (alebo v exokerneli).

Exo vs Uni

147

- exokernel vie spúšťať viacero aplikácií
  - obsahuje izoláciu na úrovni procesov
  - ale abstrakcie sú veľmi základné
- unikernel vie spúšťať iba jednu aplikáciu
  - poskytuje viac-menej štandardné služby
  - napr. štandardný hierarchický súborový systém
  - sieťové vrstvy / API založené na soketoch

Časť 3.4: Systémové volania

V zbytku prednášky sa zameriame na monolitické kernely, keďže pokrokovejšie návrhy nepoužívajú tradičný mechanizmus systémových volaní. V týchto systémoch je väčšina 'systémových volaní' implementovaná pomocou posielania správ a iba služby poskytované priamo mikrokernelom používajú mechanizmus, ktorý sa podobá systémovým volaniam ako sú popísané v tejto sekcii.

Opakovanie: Ochrana kernelu

149

- kernel je spúšťaný v privilegovanom režime CPU
- pamäť kernelu je chránená od užívateľského kódu

Ale: Kernelové služby

- užívateľský kód si musí od kernelu vypýtať služby
- ako prepneme CPU do privilegovaného režimu?
- nemôžeme to robiť ľubovoľne (bezpečnosť)

Hlavným účelom rozhrania systémových volaní je umožniť bezpečný prevod kontroly medzi užívateľskou aplikáciou a kernelom. Spomeňte si, že každý je spúšťaný s inou úrovňou oprávnení (na úrovni CPU). Schopný mechanizmus systémových volaní musí umožniť aplikácii prepnúť CPU do privilegovaného režimu (aby CPU mohlo vykonať kód patriaci kernelu), ale spôsobom, aby neumožnil aplikácii spustiť jej vlastný kód v tomto režime.

Systémové volania

150

- odovzdajú riadenie kernelovej rutine
- sprostredkujú argumenty kernelu
- obdržia návratovú hodnotu z kernelu
- všetko to musí byť urobené bezpečne

Chceme, aby sa systémové volania chovali viac-menej ako štandardné podprogramy (napr. tie poskytované systémovými knižnicami): to znamená, že chceme byť schopní podprogramu odovzdať argumenty a obdržať jeho návratovú hodnotu. Podobne ako keď prevádzame control flow (tok riadenia), potrebujeme aby odovzdávanie argumentov bolo bezpečné: užívateľská strana volania nesmie byť schopná čítať ani modifikovať pamäť kernelu.

Prenos riadenia do kernelu (Trapping into the Kernel)

151

- existuje niekoľko možných mechanizmov
- detaily sú veľmi závislé na architektúre
- všeobecne, kernel nastaví fixnú vstupnú adresu
  - inštrukcia prepne CPU do privilegovaného režimu
  - a zároveň skočí na túto adresu

Ochrana pred spustením ľubovoľného kódu aplikáciou je zabezpečená naviazaním zvýšenia privilégií (t.j. prepnutie do privilegovaného CPU režimu) na súčasný prevod výpočtu na fixnú adresu, ktorú aplikácia nemôže zmeniť. Konkrétny mechanizmus je výrazne závislý na architektúre, ale popísaný princíp je univerzálny.

Príklad prepnutia (Trap): x86

152

- na týchto CPU existuje inštrukcia `int`
- nazýva sa softvérové prerušenie – interrupt
  - prerušenia sú obvykle záležitosťou hardvéru
  - obsluha prerušení (handlers) bežia v privilegovanom režime
- tiež je synchrónna
- handler je nastavený v `IDT` (interrupt descriptor table – tabuľka deskriptorov prerušení)

Na tradičných (32 bitových) x86 CPU je preferovaná metóda implementácie prenosu kontroly systémových volaní (syscall trap) pomocou softvérových prerušení. V tomto prípade aplikácia použije inštrukciu `int`, ktorá spôsobí, že CPU vykoná proces podobný hardvérovému prerušeniu. Dva dôležité aspekty sú:

1. CPU sa prepne do privilegovaného režimu, aby spustilo obsluhu prerušenia,
2. prečíta adresu, na ktorú skočiť, z tabuľky obslúh prerušení; jedná sa o dátovú štruktúru uloženú v RAM, na adrese dodanej v špeciálnom registri.

Kernel nastaví tabuľku obslúh prerušení (interrupt handler table) takým spôsobom, aby ju užívateľský kód nemohol meniť (pomocou štandardnej ochrany pamäte založenej na MMU). Register, v ktorom je uložená jeho adresa nemôže byť zmenený mimo privilegovaný režim.

Softvérové prerušenia

153

- sú dostupné na množstve CPU
- všeobecne nie sú veľmi efektívne pre systémové volania
- nadbytočné presmerovanie – indirection
  - adresa obsluhy (handleru) je získaná z pamäte
  - musí byť uloženého veľa CPU stavu

Podobný mechanizmus je prítomný na mnohých ďalších architektúrach procesorov. Existujú však nevýhody, čo sa týka použitia tohto prístupu pre systémové volania, hlavnou je slabý výkon. Keďže tento mechanizmus sa "vezie" na hardvérovom variante prerušení, CPU obvykle uloží výrazne viac výpočetného stavu, než je nutné. Ďalšou nepríjemnosťou je, že existuje viacero vstupných bodov, ktoré tým pádom musia byť uložené v RAM (namiesto registra), čo spôsobuje dodatočné spomalenie, keď CPU potrebuje čítať tabuľku prerušení. Konečne, argumenty musia byť odovzdávané skrze pamäť, keďže registre sú prerušením resetované, čo ešte viac navyšuje latenciu.

Na okraj: SW prerušenia na PC

154

- sú použité dokonca aj v reálnom režime
  - pôvodný 16-bitový režim 80x86 CPU
  - BIOS (firmvér) rutiny prostredníctvom `int 0x10` & `0x13`
  - MS-DOS API cez `int 0x21`
- a na starších CPU v 32-bitovom chránenom režime (protected)
  - Windows NT používa `int 0x2e`
  - Linux používa `int 0x80`

Na všadeprítomnej architektúre x86 boli softvérové prerušenia preferovaným mechanizmom poskytovania služieb aplikačným programom, až do konca 32-bitovej éry x86. Čo je zaujímavé, x86 CPU od 80386 dodávajú mechanizmus, ktorý mal priamo implementovať služby operačného systému (teda systémové volania), ale bol pomerne zložitý a programátori operačných systémov ho do veľkej miery ignorovali.

Príklad Trap: amd64 / x86_64

155

- inštrukcie `sysenter` a `syscall`
  - a odpovedajúce `sysexit` / `sysret`
- vstupný bod je uložený v tzv. machine state register –

strojový stavový register

156

- existuje iba jeden vstupný bod
  - na rozdiel od softvérových prerušení
- o dosť rýchlejšie než prerušenia

Keď x86 prešla na 64-bitový adresný priestor, veľa nových inštrukcií si našlo cestu do inštrukčnej sady. Medzi nimi bola aj jednoduchá inštrukcia pre zvýšenie oprávnení s jedným vstupným bodom. Tento mechanizmus sa vyhýba väčšine réžie spojenej so softvérovými prerušeniami: výpočetný stav (computation state) je spravovaný softvérovo, čo umožňuje prekladačom uložiť a znovu načítať malý počet registrov naprieč systémovým volaním (namiesto toho, aby CPU automaticky ukladal do pamäte celý svoj stav).

Ktoré systémové volanie?

157

- často existuje veľa systémových volaní
  - viac než 300 na 64-bitovom Linuxe
  - asi 400 na 32-bitovom Windows NT
- ale existuje iba niekoľko málo prerušení
  - a iba jedna adresa `sysenter`

Obvykle existuje iba jeden vstupný bod (adresa), zdieľaná všetkými systémovými volaniami. Kernel však potrebuje byť schopný rozlíšiť, ktorú službu si aplikačný program vyžiadal.

Opakovanie: Číslovanie systémových volaní

158

- každé systémové volanie má priradené číslo
- dostupné ako `SYS_write` ap. na POSIX-ových systémoch
- pre "univerzálne" `int syscall( int sys, ... )`
- toto číslo sa posiela pomocou CPU registra

Toto sa dosahuje tým, že sa jednoducho pošle číslo systémového volania ako argument v konkrétnom CPU registri. Kernel sa potom môže rozhodnúť, na základe tohto čísla, ktorú rutinu vykonať v mene programu.

Sekvencia systémového volania

159

- najprv `libc` pripraví argumenty systémového volania
- a dá číslo syscallu do správneho registra
- potom sa CPU prepne do privilegovaného režimu
- tiež to predá kontrolu obsluhe systémového volania – handleru

Prvé štádium systémového volania beží v užívateľskom režime a je typicky implementované v `libc`.

Obsluha Systémového volania – Handler

160

- handler si najprv vyzdvihne číslo syscallu
- a rozhodne sa kde pokračovať
- môžete si to predstaviť ako obrovský príkaz `switch`

```
switch ( sysnum )
{
   case SYS_write: return syscall_write();
   case SYS_read: return syscall_read();
   /* veľa ďalších */
}
```

Po prepnutí do privilegovaného režimu musí kernel porozumieť argumentom, ktoré dodal užívateľský program, a čo je najdôležitejšie, musí sa rozhodnúť, ktoré systémové volanie bolo vyžiadané. Táto časť kódu v kerneli môže vyzerať napríklad ako príkaz `switch` uvedený vyššie.

Argumenty systémového volania

161

- každé systémové volanie má rôzne argumenty
- ako sa posielajú kernelu je závislé na CPU
- na 32-bitových `x86` je väčšina posielaná cez pamäť
- na `amd64` Linuxe sa všetky argumenty dávajú do registrov
  - dostupných 6 registrov pre argumenty

Keďže rôzne systémové volania očakávajú rôzne argumenty, spracovanie argumentov sa deje až po vybraní systémového volania prostredníctvom jeho čísla. V moderných systémoch sú argumenty posielané cez CPU registre, ale toto nebolo možné u protokolov založených na softvérových prerušeniach (namiesto toho boli argumenty posielané prostredníctvom pamäte, zvyčajne na vrchole zásobníka užívateľského priestoru).

Časť 3.5: Služby kernelu

Konečne si pripomenieme služby poskytované monolitickými kernelmi, a pozrieme sa, ako sú realizované v mikrokernelových operačných systémoch.

Čo robí kernel?

163

- správa pamäte & procesov
- plánovanie úloh (vláken) – scheduling
- ovládače zariadení
  - SSD, GPU, USB, bluetooth, HID, audio, ...
- súborové systémy
- sieťové služby

Prvé dva body tvoria kľúčové úlohy kernelu: tieto sú zriedka 'posunuté' externým službám. Zvyšné služby sú ústrednou časťou operačného systému, ale nie nutne kernelu. Ale je ťažké predstaviť si moderný univerzálny (general-purpose) operačný systém, ktorý by nejakú z týchto častí vynechal. V tradičných (monolitických) návrhoch sú všetky súčasťou kernelu.

Ďalšie služby

164

- medzi-procesová komunikácia
- časovače a udržiavanie si času
- trasovanie procesov, profiling
- bezpečnosť, sandboxing
- kryptografia

Monolitický kernel môže poskytovať množstvo ďalších služieb, s rôznou dôležitosťou. Nie všetky systémy poskytujú všetky služby, a ich implementácie môžu vyzerať dosť rozdielne na rôznych operačných systémoch. Z tohto (nekompletného) zoznamu je IPC (inter-process communication – medzi-procesová komunikácia) jediná položka, ktorá je univerzálne prítomná, v nejakej forme, v mikrokerneloch. Okrem toho, zatiaľ čo IPC mechanizmy sú pomerne bežné v monolitických kerneloch, sú dôležitejšie u mikrokernelov.

Opakovanie: Mikrokernelové systémy

165

- samotný kernel je veľmi malý
- je sprevádzaný servermi
- v "pravých" mikrokernelových systémoch je prítomných veľa serverov
  - každé zariadenie, súborový systém, atď. je oddelené
- v hybridných systémoch je server jeden alebo niekoľko
  - "superserver", ktorý pripomína monolitický kernel

Spomeňte si, že mikrokernel je malý: dodáva iba služby, ktoré sa nedajú rozumne implementovať mimo kernel. Samozrejme, operačný systém ako celok stále potrebuje implementovať tieto služby. Sú dostupné dve základné stratégie:

1. jediný program, bežiaci v jednom procese, implementuje všetku chýbajúcu funkcionalitu: tento program sa nazýva superserver a jeho interná architektúra dosť pripomína architektúru monolitického kernelu,
2. každá služba je poskytovaná osobitným, špecializovaným programom, bežiacim vo svojom vlastnom procese (a teda, adresnom priestore) – toto je charakteristické pre takzvané 'pravé' (true) mikrokernelové systémy.

Samozrejme tieto dva základné návrhy so sebou nesú rôzne kompromisy. Hybridný systém (t.j. systém so superserverom) je spočiatku jednoduchší na návrh a implementáciu (napríklad, ovládače trvalého úložiska, bloková vrstva a súborový systém môžu všetky zdieľať rovnaký adresný priestor, čím sa implementácia zjednodušuje) a je často výrazne rýchlejší, keďže komunikácia medzi komponentmi nezahŕňa prepínanie kontextu. Na druhú stranu, pravý mikrokernelový systém so službami a ovládačmi prísne oddelenými do samostatných procesov je robustnejší, a teoreticky tiež jednoduchšie škálovateľný na veľké SMP systémy.

Služby kernelu

166

- obvykle je nám jedno ktorý server čo poskytuje
  - každý systém je iný
  - systém sa chová akoby bol monolitický
- služby sú používané skrze systémové knižnice
  - abstrahujú veľa detailov
  - napr. či je služba systémové volanie alebo IPC volanie

Z pohľadu užívateľského priestoru by nemalo záležať na špecifikách architektúry jadra. Aplikácie v každom prípade používajú systémové knižnice pre komunikáciu s kernelom: je úlohou týchto knižníc implementovať protokol pre nájdenie príslušných serverov a pre interakciu s nimi.

Ovládače v užívateľskom priestore u monolitických systémov

167

- nie všetky ovládače zariadení sú súčasťou kernelu
- príklad: ovládače tlačiarne
- tiež niektoré USB zariadenia (ale nie USB zbernica)
- časť GPU/grafického zásobníka
  - správa pamäte a výstupu (output) v kerneli
  - väčšina OpenGL v užívateľskom priestore

Zatiaľ čo ovládače v užívateľskom priestore nie sú úplne ideálne, ale sú očakávané, v mikrokernelových systémoch, existujú prípady, kedy ovládače v operačných systémoch založených na monolitických kerneloch majú významnú užívateľskú zložku. Najčastejší príklad sú pravdepodobne ovládače tlačiarní: nízkoúrovňová komunikácia s tlačiarňou (na úrovni USB) je sprostredkovaná kernelom, ale pre veľa tlačiarní predstavuje veľkú časť funkcionality ovládača spracovanie dokumentov. V niektorých prípadoch to zahŕňa prevod formátu (napr. PCL tlačiarne), u ďalších je vstupný dokument rastrovaný ovládačom na hlavnom CPU: namiesto posielania textu a informácií o rozložení dokumentu tlačiarni posiela ovládač dáta o pixeloch, alebo dokonca prúd príkazov pre hlavu tlačiarne.
Situácia u GPU je trochu analogická: nízkoúrovňový prístup k hardvéru je poskytovaný kernelom, ale opäť, veľká časť ovládača je dedikovaná na manipuláciu s dátami: staranie sa o trojuholníkovú sieť, textúry, osvetlenie, atď. Naviac, moderné GPU sú všetky *programovateľné*: prekladač shadera je tiež súčasťou ovládača, prekladajúc vysokoúrovňové shader programy do prúdov inštrukcií, ktoré môžu byť spustené CPU.
Ovládačmi zariadení sa budeme bližšie zaoberať v 8. prednáške.

Review Questions

168

9. What CPU modes are there and how are they used?
10. What is the memory management unit?
11. What is a microkernel?
12. What is a system call?

Časť 4: Súborové systémy

Súborové systémy sú neoddeliteľnou súčasťou univerzálnych (general-purpose) operačných systémov. Okrem ukladania užívateľských dát, sídlia v súborovom systéme aj programy a ďalšie komponenty, ktoré tvoria operačný systém. V tejto prednáške sa pozrieme na vnútorné fungovanie typickej POSIX-kompatibilnej vrstvy súborového systému.

Obsah prednášky

170

1. Základy súborových systémov
2. Bloková vrstva
3. Prepínač virtuálneho súborového systému
4. UNIX-ový súborový systém
5. Pokročilé funkcie

Najprv si zopakujeme nejaké základné koncepty, ktoré sme si uviedli v predchádzajúcich prednáškach. Potom sa pozrieme na blokovú vrstvu, ktorá sa nachádza pod súborovým systémom a poskytuje jednoduchšiu abstrakciu ukladania dát. Následne sa pozrieme na VFS, systém, ktorý kernely typicky používajú na umožnenie viacerých implementácií štruktúr na disku, pri udržaní jednotného rozhrania systémových volaní zvonku.
Potom sa pozrieme na sémantiku a na organizáciu typického UNIX-ového súborového systému na disku. Nakoniec si v rýchlosti prejdeme niektoré pokročilé funkcie prítomné v moderných súborových systémoch (a pod nimi, na blokovej vrstve).

Časť 4.1: Základy súborových systémov

V tejto sekcii sa budeme zaoberať základnými zložkami súborového systému, tak ako im rozumie POSIX – niektorá látka bude opakovaním konceptov, ktoré sme už videli.

Čo je to súborový systém?

172

- zbierka súborov a adresárov
- (z veľkej časti) hierarchické
- obvykle prístupné užívateľovi
- obvykle perzistentné (medzi reštartovaním)
- správca súborov, príkazový riadok, atď.

Prvým problémom je definícia súborového systému ako celku. Najabstraktnejší pohľad je, že súborový systém je kolekcia súborov a adresárov, kde súbory sú v podstate sekvencie bajtov a adresáre dávajú mená súborom a ďalším adresárom. Keďže adresáre môžu obsahovať ďalšie adresáre, celá štruktúra tvorí strom (až na pár výnimiek).
Súborové systémy sú obvykle viditeľné pre užívateľov: užívateľ môže priamo pristupovať k hierarchii adresárov a prehliadať si obsah adresárov aj súborov, použitím nástrojov, ktoré mu poskytuje operačný systém. Ďalšou typickou vlastnosťou súborového systému je, že je perzistentný: reštartovanie počítača nevymaže ani nezmení dáta uložené v súborovom systéme. (Samozrejme, obe tieto vlastnosti majú výnimky: súborové systémy môžu byť uložené na virtuálnych zariadeniach, ktoré používajú na ukladanie pamäť RAM, tieto systémy nebudú perzistentné; podobne, niektoré operačné systémy neumožňujú užívateľovi priamo pristupovať k súborovému systému).

Čo je to (bežný) súbor - regular file?

173

- sekvencia bajtov
- a nejaké základné metadáta
  - vlastník, skupina, časová známka
- OS obsah súboru nezaujíma
  - text, obrázky, video, zdrojový kód sú všetky rovnaké
  - spustiteľné súbory sú trochu špeciálne

Keďže sme si definovali súborový systém ako kolekciu súborov a adresárov, budeme tiež potrebovať zadefinovať súbor. Opäť môžeme použiť pomerne abstraktnú definíciu, ktorá funguje: súbor je objekt, ktorý sa skladá z postupnosti bajtov (dáta) a nejakých dodatočných metadát - informácií o súbore. Obsah (tzn. postupnosť bajtov) obvykle nie je interpretovaný operačným systémom (až na špeciálne prípady, ako spustiteľné súbory). Metadáta obsahujú veci ako identifikátor vlastníka (owner) súboru, nejaké časové známky (timestamp; napr. čas poslednej úpravy) a prístupové práva. Uvedomte si prosím, že metadáta súboru nezahŕňajú jeho názov: názov súboru nie je vlastnosťou súboru samotného na POSIX-ových systémoch.

Čo je to adresár?

174

- zoznam mapovaní názov → súbor
- ak chcete, asociatívny kontajner
  - sémanticky, typy hodnôt nie sú homogénne
  - syntakticky, sú to proste i-uzly (i-nodes)
- jeden adresár = jeden komponent cesty (path)
  - `/usr/local/bin`

Posledná časť definície súborového systému, ktorú sme si ešte nevysvetlili, je adresár. Adresáre sú mapy (asociatívne polia, slovníky), kde kľúče sú názvy / mená a hodnoty sú súbory a ďalšie adresáre. V skutočnosti sú súbory aj adresáre v súborovom systéme reprezentované rovnakou dátovou štruktúrou: i-uzlom (i-node).
Tiež si spomíname, že adresáre tvoria strom, a že používame cesty (paths) na identifikovanie jednotlivých súborov a adresárov v súborovom systéme. Každý komponent (sekcia oddelená / na každej strane) takejto cesty je použitý ako kľúč v jednom adresári.

Čo je i-uzol (i-node)?

175

- anonymný objekt podobný súboru
- môže ísť o bežný súbor
  - alebo adresár
  - alebo špeciálny súbor
  - alebo symbolický odkaz

i-uzol je spôsob, akým sú súbory a adresáre (a každý ďalší objekt v súborovom systéme) reprezentované v POSIX-ovom súborovom systéme. i-uzol si ukladá referencie na bloky dát (o tom viac neskôr) a metadáta súboru.

Súbory sú anonymné

176

- toto platí v UNIXe
  - nie všetky súborové systémy takto fungujú
- tento prístup má výhody a nevýhody
  - napr. otvorené súbory môžu byť odpojené (unlinked)
- mená sú priradené cez directory entries - adresárové záznamy

Ako sme už spomenuli, súbory nemajú mená - t.j. i-uzol nemá pole, v ktorom by bol uložený názov objektu. Namiesto toho je názov daný súboru prostredníctvom adresárového záznamu, ktorý vytvorí väzbu medzi reťazcom (čokoľvek, čo neobsahuje znaky /) a i-uzlom.
To okrem iného znamená, že je možné „odpojiť“ (unlink) súbory (odstrániť ich adresárové záznamy) bez vymazania ich dát: toto sa napríklad deje, keď je súbor práve otvorený (a niekto sa ho pokúsi „zmazať“). Keď posledný proces zavrie svoj posledný popisovač súboru (file descriptor) spätý s daným i-uzlom, dáta sú konečne vymazané z disku.

Čo ďalšie je postupnosť bajtov? 177

- znaky prichádzajúce z klávesnice
- bajty uložené na magnetickej páske
- audio dáta prichádzajúce z mikrofónu
- pixely prichádzajúce z webkamery
- dáta prichádzajúce na TCP spojení

Naznačili sme si, že postupnosť bajtov (byte sequence) je široko uplatniteľná abstrakcia. Existuje veľa vecí, ktoré (s trochou zveličovania) možno považovať za postupnosti bajtov. Väčšina z nich nie je perzistentná ako súbory, ale to nie je u programovacích rozhraní problém.

Zapisovanie postupností bajtov 178

- posielanie dát tlačiarni
- spätné prehrávanie zvuku
- zapisovanie textu do terminálu (emulátor)
- posielanie dát cez TCP prúd

Okrem čítania bajtov podporujú súbory (a veľa ďalších vecí) zapisovanie bajtov, t.j. ukladanie alebo iné spracovanie sekvencie bajtov, ktorá je poslaná (zapísaná) do daného objektu.

Špeciálne súbory 179

- veľa vecí vyzerá trochu ako súbory
- poďme to zneužiť a zjednotiť ich so súbormi
- spomeňte si na časť 2 u API: "všetko je súbor"
  - API je rovnaké pre špeciálne a bežné súbory
  - nie však jeho implementácia

Všeobecnosť tejto abstrakcie nám umožňuje (a je to žiaduce) reprezentovať všetky tieto objekty jednotne, použitím API súborového systému. Okrem využívania rovnakého API sa však veľa z týchto objektov doslova objavuje v súborovom systéme ako i-uzly so špeciálnymi vlastnosťami. Samozrejme, keď je takýto súbor otvorený, čítanie a zapisovanie do popisovača súboru nie je vykonávané rovnakými obslužnými rutinami kernelu, ako tými, ktoré sa starajú o bežné súbory. Viac si o tomto povieme v neskoršej sekcii, pri VFS.

Typy súborových systémov 180

- fat16, fat32, vfat, exfat (DOS, flash médiá)
- ISO 9660 (CD-ROMs)
- UDF (DVD-ROM)
- NTFS (Windows NT)
- HFS+ (macOS)
- ext2, ext3, ext4 (Linux)
- ufs, ffs (BSD)

Prirodzene, existuje veľa rôznych implementácií abstraktnej myšlienky súborového systému. Hoci ich veľa má identickú alebo podobnú sémantiku (vo väčšine prípadov popísaných v tejto prednáške a kodifikovaných POSIX-om), nie je to vždy tak: napríklad súborové systémy v rodine `FAT` nemajú koncept i-uzlu a názvy súborov sú vstavanou vlastnosťou súboru.
Napriek tomu, že sémantika je často podobná, diskový formát na pozadí sa môže značne rozchádzať: zatiaľ čo rodina `ext` a rodina `ufs` používajú pomerne tradičný prístup, veľa moderných súborových systémov, ako ZFS, btrfs alebo hammer je vnútri postavených na B-stromoch, rozšíriteľnom hashovaní (extendible hashing) alebo iných škálovateľných dátových štruktúrach, ktoré zvládajú výrazne väčší objem s oveľa viac súbormi ako tradičné, relatívne naivné prístupy. Súčasne moderné súborové systémy poskytujú lepšiu odolnosť voči poškodeniu svojich dátových štruktúr v prípade chýb alebo neočakávaného výpadku energie.

Viac-užívateľské (Multi-User) systémy 181

- vlastníctvo súborov
- práva súborov
- diskové kvóty

Viac-užívateľské systémy prinášajú dodatočnú sadu výziev, čo sa týka implementácie súborového systému. Väčšinou je žiaduce, aby súbory každého užívateľa neboli prístupné žiadnemu inému užívateľovi, pokiaľ sa majiteľ súboru nerozhodne konkrétny súbor zdieľať. Podobne, chceme byť schopní obmedziť, koľko miesta môže každý užívateľ zabrať, čo sa väčšinou rieši diskovými kvótami.

Vlastníctvo & Práva 182

- predpokladáme tzv. voliteľný - discretionary model
- ten, kto vytvoril súbor, je jeho majiteľ (owner)
- vlastníctvo môže byť prevedené
- majiteľ sa rozhoduje o právach (permissions)
  - v podstate čítanie, zápis, spúšťanie

Kvôli vyššie uvedeným dôvodom musí systém byť schopný pamätať si:

1. vlastníctvo v rámci súborového systému, tj. pamätať si, ktorý súbor patrí ktorému užívateľovi,
2. práva súboru, tj. čo majiteľ daného súboru považuje za vhodné použitie svojho súboru – či môžu ostatní užívatelia (a ktorí užívatelia) čítať obsah súboru, alebo dokonca zapisovať nové dáta alebo nahradiť existujúce dáta v súbore.

V modeli voliteľného prístupu (discretionary access model) sa môže majiteľ slobodne rozhodnúť o prístupových právach. Existujú iné modely, kde je táto možnosť obmedzená (väčšinou na zvýšenie bezpečnosti).

Diskové kvóty 183

- disky sú veľké ale nie nekonečné
- keď sa súborový systém zaplní, dejú sa zlé veci
  - denial of service
  - programy môžu zlyhať a dokonca poškodiť dáta
- kvóty obmedzujú množstvo priestoru pre užívateľa

Okrem limitovania, čo sa môže robiť so súbormi, musí systém byť schopný spravovať voľné miesto v súborovom systéme: keďže sú súborové systémy uložené na nejakom fyzickom médiu, množstvo dát, ktoré môžu byť uložené, je obmedzené. Diskové kvóty sú mechanizmus, ktorým operačný systém zabezpečuje férovú alokáciu priestoru medzi užívateľmi (alebo aspoň bráni jednému užívateľovi obsadiť všetko miesto v súborovom systéme).

Časť 4.2: Bloková vrstva

Bloková vrstva (operačného systému) sa stará o nízkoúrovňový prístup k perzistentným úložným zariadeniam, ako sú pevné disky, SSD (solid-state drive - mechanika s nepohyblivým médiom), a ďalšie podobné zariadenia. Samotný súborový systém tým pádom nemusí rozumieť detailom konkrétneho prítomného zariadenia alebo protokolu, prostredníctvom ktorého komunikácia s daným zariadením prebieha.

Namiesto toho používa jednotné API, ktoré umožňuje súborovému systému ukladať a získavať bloky dát. Alebo, inými slovami, blokové zariadenie je abstrakcia, ktorá zjednodušuje implementáciu súborových systémov (keďže táto nemusí riešiť toľko detailov) a robí ju univerzálnejšou (nezávisí na špecifikách daného úložného zariadenia).

Zariadenia podobné disku

185

- pevné disky poskytujú prístup na úrovni blokov
- čítanie a zapisovanie dát v 512-bajtových kusoch (tzv. chunks)
  - alebo tiež 4K na veľkých moderných diskoch
- veľké číslované pole blokov

Blokové zariadenie (tj. zariadenie podobné disku) má nasledujúce základné operácie: dokáže čítať alebo zapisovať blok, čo je nejaký kus dát (chunk) fixnej veľkosti. Konkrétna veľkosť závisí na type zariadenia, ale obvykle má blok veľkosť buď 512 bajtov (na starších zariadeniach) alebo 4096 bajtov na modernejšom hardvéri. Bloky sú očíslované a celé zariadenie v podstate 'vyzerá' ako veľké pole takýchto blokov. Čítanie a zapisovanie blokov znamená, v tomto kontexte, prenos dát uložených v tomto bloku do alebo z hlavnej pamäte (RAM).

Na okraj: Schémy adresovania disku

186

- CHS: Cylinder, Hlava, Sektor
  - štruktúrované adresovanie používané vo (veľmi) starých diskoch
  - poskytuje informácie o relatívnych časoch prístupu
  - neužitočné pri cylindroch rôznej dĺžky
  - 10:4:6 CHS = 1024 cylindrov, 16 hláv, 63 sektorov
- LBA: Logical Block Addressing - Logické adresovanie blokov
  - lineárny, neštruktúrovaný adresný priestor
  - začal ako 22, potom 28, ... teraz 48 bitový

Staré rotačné disky používali schému adresovania, ktorá odzrkadľovala ich fyzickú geometriu. Adresa sa skladala z 3 čísel: cylinder (vzdialenosť od stredu platne disku), hlava (ktorá platňa, alebo skôr ktorá strana ktorej platne obsahuje daný sektor) a číslo sektora (uhol, pod ktorým je sektor uložený). Toto umožnilo ovládaču a operačnému systému odhadnúť latenciu operácie: čítanie sektorov z rovnakého cylindra bolo zvyčajne rýchle, čítanie sektorov z rôznych cylindrov vyžadovalo drahý pohyb hlavy (známy ako 'seeking' - vyhľadávanie).
Od schémy CHS bolo upustené, keď sa diskové kapacity zväčšili a začal dochádzať adresný priestor. Jeho náhrada, známa ako LBA alebo Logické adresovanie blokov (Logical Block Addressing), používa neštruktúrovaný plochý adresný priestor (rovnako ako hlavná pamäť, ale jednotkou adresovania je blok, nie bajt).

Prístup na úrovni blokov

187

- ovládače diskov umožňujú iba lineárne adresovanie
- minimálna veľkosť čítania/zápisu je jeden blok (sektor)
- veľa sektorov môže byť zapísaných 'naraz'
  - sekvenčný prístup je rýchlejší ako náhodný
  - maximálna priepustnosť (throughput) vs IOPS

Vyššie vrstvy operačného systému (počnúc blokovou vrstvou a implementáciou súborového systému) nezaujíma rozhranie medzi ovládačom disku a diskom samotným a vždy používajú lineárne adresovanie. Na blokovej vrstve sú presuny často re-organizované, aby sa maximalizovali sekvenčné zápisy (o tom si povieme za chvíľu), pretože väčšina diskových jednotiek dokáže čítať alebo zapisovať súvislú sekvenciu blokov výrazne rýchlejšie, než ich dokážu čítať jeden po druhom.
Režim sekvenčného prístupu je dôležitý, keď sa pýtame na priepustnosť (throughput) systému (koľko bajtov je zariadenie schopné dodávať alebo ukladať za sekundu), zatiaľ čo počet blokov, ktoré dokáže čítať alebo zapisovať, keď sú náhodne distribuované po celom zariadení, je známy ako IOPS – počet vstupných/výstupných (input/output) operácií za sekundu.

Na okraj: Doba/Čas prístupu

188

- blokové zariadenia sú pomalé (v porovnaní s RAM)
  - RAM je pomalá (v porovnaní s CPU)
- nemôžeme sa chovať k diskom ako ku nadstavbe RAM
  - ani ku najrýchlejšiemu modernému flash úložisku
  - latencia: HDD 3–12 ms, SSD 0.1 ms, RAM 70 ns

Celý návrh úložnej vrstvy, počnúc hardvérovými zbernicami, až k súborovým systémom a dokonca aplikačným softvérom, najčastejšie databázovým systémom, je výrazne ovplyvnený pomalosťou blokových zariadení. Aj najnovšie disky založené na flash pamätiach sú niekoľkonásobne pomalšie než RAM (ktorá je niekoľkonásobne pomalšia než CPU keše, ktoré sú stále pomalšie než výpočetné jednotky v CPU).

Keš prístupu k blokom

189

- kešovanie (caching) sa používa na skrytie latencie
  - rovnaký princíp medzi CPU a RAM
- súbory, ku ktorým bolo nedávno pristupované, sú držané v RAM
  - existuje veľa stratégií správy keší
- implementované plne v OS
  - veľa zariadení si implementuje vlastné kešovanie
  - ale množstvo rýchlej pamäte je väčšinou obmedzené

Už viete, že CPU sa spolieha na keš pamäte na skrytie latencie RAM: dáta, ktoré boli nedávno použité, alebo také, o ktorých systém predpokladá, že ich bude čoskoro potrebovať, sú uložené v, alebo prednačítané do, výrazne rýchlejšej (ale výrazne menšej) keše. To znamená, že u dát, ktoré sa používajú najčastejšie, CPU nemusí čakať celý čas latencie RAM.
Rovnaký princíp je použitý u operačných systémov na zakrytie latencie blokových zariadení, ale v tomto prípade je 'rýchla' a 'malá' pamäť hlavná pamäť (RAM), zatiaľ čo veľká a pomalá pamäť je blokové zariadenie. Na rozdiel od CPU keše, táto takzvaná disková keš je implementovaná plne softvérovo.
Zvyčajne je však prítomná ďalšia vrstva (alebo dve) kešovania, ktorá je vykonávaná hardvérom alebo firmvérom disku a ktorá používa dedikované hardvérové buffery (iný typ vyrovnávacej pamäte; zvyčajne implementované ako dynamické RAM alebo dokonca flash úložisko) na strane zariadenia na zbernici. Toto sa líši od keše na úrovni OS a odteraz to budeme ignorovať.

Zapisovacie buffery (Write Buffers)

190

- zapisovací ekvivalent blokovej keše
- dáta sú udržiavané v RAM než môžu byť spracované
- musí byť synchronizované s kešovaním
  - iní užívatelia môžu práve súbor čítať

Je jeden veľký rozdiel medzi CPU kešou a kešou blokového zariadenia,

ktorá je spravovaná OS: medzi kešou (RAM) a hlavným úložiskom (disk) je nezhoda v životnosti dát. Keď sa systém vypne, obsah RAM je stratený a spolu s ním všetky zmeny, ktoré nestihli byť replikované z keše do perzistentného úložného zariadenia. Toto vytvára veľa zložitosti, keďže aplikačný softvér (a užívatelia) očakávajú, že dáta zapísané na disk tam zostanú. Je to obzvlášť dôležité (a problematické) u systémov, ktoré musia byť odolné voči neočakávaným výpadkom energie, ale aj výpadky operačného systému môžu mať rovnaký efekt, aj keď je systém pripojený k spoľahlivému zdroju energie.
Preto veľa operačných systémov využíva tzv. split cache (rozdelená keš): dáta, ktoré majú byť zapísané sú uložené v zapisovacích bufferoch a „vyliate" (flushed) na disk, ako to dovoľuje priepustnosť a ďalšie zdroje. Samozrejme, dáta uložené v zapisovacích bufferoch musia byť replikované do čítacej keše (alebo s ňou iným spôsobom zdieľané), keďže ďalšie procesy môžu čítať rovnaké bloky (väčšinou prostredníctvom súborového systému) a mali by vidieť nový obsah.

I/O Plánovač (Výťah - Elevator)

191

- čítania a zápisy sú vyžiadané užívateľmi
- usporiadanie prístupov je kľúčové na mechanickom disku
  - nie až tak dôležité u SSD
  - ale sekvenčný prístup je stále výrazne preferovaný
- požiadavky sú dané do fronty (spomeňte si, že disky sú pomalé)
  - ale nie sú spracované v poradí FIFO

Už sme spomenuli, že sekvenčný prístup je výrazne rýchlejší ako náhodný prístup a že latencia rotačných diskov závisí na fyzickej vzdialenosti medzi umiestnením jednotlivých kusov dát. Keďže na IO subsystémoch dochádza k častému paralelizmu (rôzne procesy čítajú a zapisujú do rôznych súborov naraz), často sa vyplatí preusporiadať požiadavky na IO operácie.
Požadovaný efekt je, že úplne náhodná sekvencia IO operácií prichádzajúca z niekoľkých procesov, povedzme 16 požiadaviek, kde každá požiadavka je na inej fyzickej pozícii, než predchádzajúca, je preusporiadaná do 4 skupín po 4 operáciách, ktoré sú fyzicky blízko (alebo dokonca sekvenčné). Toto preusporiadanie môže ľahko zlepšiť celkovú priepustnosť systému (v tejto vzorovej situácii) 3–4×, keďže jedna dávka 4 sekvenčných operácií nezaberie skoro žiaden čas naviac, než latencia plynúca z náhodného prístupu spôsobená prvou operáciou v tejto sekvencii.
Preusporiadanie sa robí výrazne lepšie s buffrovanými zápismi: v tomto prípade má bloková vrstva voľnosť v prehadzovaní zapisovacej fronty (pri dodržaní obmedzení na usporiadanie daných súborovým systémom alebo prípadne aplikáciou). Čítanie je však zložitejšie: OS môže nepochybne odhadovať („špekulovať"), ktoré operácie čítania budú požadované ako ďalšie. V systémoch s pravým asynchrónnym IO (kde aplikácia nie je prebiehajúcim IO blokovaná), môže byť čítacia fronta dlhšia, ale stále bude poskytovať menej voľnosti, než zapisovacia fronta.

RAID

192

- pevné disky sú tiež nespoľahlivé
  - zálohy pomáhajú, ale obnovenie trvá dlho
- RAID = Redundant Array of Inexpensive Disks
  - súbežná replikácia rovnakých dát cez niekoľko diskov
  - veľa rôznych konfigurácií
- systém zostáva online napriek zlyhaniam disku

Hoci je rýchlosť perzistentného úložiska veľkým problémom, jeho spoľahlivosť (resp. jej nedostatok) je tiež dosť dôležitá. Zatiaľ čo problém s rýchlosťou sa rieši kešovaním, spoľahlivosť sa obvykle zlepšuje cez redundanciu: v prípade, že komponent zlyhá, ďalšie komponenty ho nahradia. V prípade úložiska to znamená, že dáta musia byť replikované cez niekoľko zariadení, spôsobom, že keď jedno zlyhá, ostatné stále dokážu dať dokopy úplnú kópiu dát.
RAID (Redundant Array of Independent/Inexpensive Disks - Nadbytočné pole nezávislých/lacných diskov) je nízkoúrovňovou realizáciou tohoto princípu. RAID môže byť implementovaný hardvérovo alebo softvérovo, pričom druhá možnosť je v súčasných systémoch bežnejšia. Softvérový RAID je súčasťou blokovej vrstvy operačného systému, a väčšinou býva prezentovaný vyšším vrstvám ako jedno virtuálne zariadenie. Čítanie a zapisovanie do tohto virtuálneho zariadenia spôsobí, že RAID subsystém blokovej vrstvy rozmiestni dáta cez niekoľko fyzických zariadení. Väčšina konfigurácií RAID potom dokáže pokračovať bez straty dát (a bez prestávky), keď jedno z fyzických zariadení zlyhá.

Výkon RAID

193

- RAID ovplyvňuje výkonnosť blokovej vrstvy
- často zlepšuje priepustnosť čítania
  - dáta sú skombinované z viacerých kanálov
- výkonnosť zápisu je viac rôznorodá
  - môže vyžadovať pomerne veľa výpočtov
  - viac dát musí byť zapísaných na zabezpečenie redundancie

V plne funkčnom poli RAID je výkonnosť čítania obvykle výrazne posilnená: dáta sú paralelne zozbierané z viacerých zariadení, kde každé prispieva časť do celkovej priepustnosti.
Zapisovanie je menej zrejmé: záleží na konfigurácii RAID, zápisy môžu byť rýchlejšie alebo pomalšie, než pri jednom disku. Zapisovanie do softvérového RAID poľa tiež spotrebúva dodatočné CPU cykly, keďže operačný systém musí rozdeliť dáta (a často aj spočítať kontrolné súčty - checksum).

Šifrovanie na úrovni blokov

194

- symetrické & zachovávajúce dĺžku
- šifrovací kľúč je odvodený od hesla
- tiež známe ako "full disk encryption" - šifrovanie celého disku
- spôsobuje mierne zhoršenie výkonu
- veľmi dôležité pre bezpečnosť / súkromie

Ďalšou funkciou, ktorá je bežne prítomná v moderných operačných systémoch je šifrovanie dát na úrovni blokov. To chráni systém pred offline útokmi: napr. ak je počítač ukradnutý, dáta zostávajú neprístupné bez šifrovacieho kľúča (ktorý sa obvykle odvodzuje z hesla).
Šifrovanie disku používa symetrickú kryptografiu (zvyčajne hardvérovo-zrýchlený - HW accelerated - AES) a u väčšiny implementácií šifrovanie zachováva dĺžku: jeden blok dát produkuje jeden blok kódovaných dát, ktorý je potom priamo uložený v odpovedajúcom fyzickom bloku perzistentného úložného zariadenia. Pre väčšinu softvéru je tento typ šifrovania plne transparentný. Dokonca ani implementácia súborového systému v operačnom systéme si nemusí uvedomovať, že je uložená na šifrovanom zariadení.

Ukladanie dát v blokoch

195

- rozdeľovanie dát na kusy fixnej dĺžky je neprirodzené
- neexistuje systém oprávnení pre jednotlivé bloky
  - na rozdiel od virtuálnej (stránkovanej) pamäte
  - bolo by to veľmi nepohodlné pre užívateľov
- procesy nie sú perzistentné, ale blokové úložiská áno

Sme zvyknutí pracovať s ľubovoľne veľkými súbormi, ale tento predpoklad sa nedá rozumne namapovať na abstrakciu blokových zariadení, kde sú dáta uložené v pomerne veľkých kusoch. Bloková vrstva navyše neponúka žiaden systém oprávnení alebo možnosť zdieľania či multiplexovania: sú tam iba bloky, a proces môže čítať alebo zapisovať do ktoréhokoľvek z nich, alebo do žiadneho.
Zatiaľ čo hlavná pamäť je tiež len pomerne hlúpe pole bajtov, organizované do stránok (ktoré do veľkej miery pripomínajú bloky), obsah tejto hlavnej pamäte je výrazne viac dynamický a výrazne menej dôležitý pre užívateľa. Sčasti je rozdiel spôsobený perzistenciou, ale inak ide v podstate o historickú náhodu.

Súborový systém ako zdieľanie zdrojov

196

- väčšinou iba 1 alebo niekoľko diskov na počítač
- veľa programov chce ukladať perzistentné dáta
- súborový systém vyhradí pre dáta miesto
  - ktoré bloky patria ktorému súboru
- rôzne programy môžu zapisovať do rôznych súborov
  - nevzniká riziko použitia rovnakého bloku

Musí však existovať nejaký mechanizmus pre zdieľanie perzistentného úložiska medzi rôznymi procesmi (a rôznymi užívateľmi). Týmto mechanizmom je, samozrejme, súborový systém. Okrem iného spravuje alokáciu voľného miesta pre súbory, a zabezpečuje perzistentné identity pre tieto súbory.

Súborový systém ako abstrakcia

197

- umožňuje dátam byť organizované do súborov
- umožňuje užívateľovi spravovať a prehliadať dáta
- súbory majú ľubovoľnú & dynamickú veľkosť
  - bloky sú transparentne alokované & recyklované
- štruktúrované dáta namiesto plochého poľa blokov

Blokové úložisko nie je veľmi vyhovujúce a v skoro všetkých prípadoch potrebuje dodatočné abstrakcie. Môžeme si vyvodiť analógiu s hlavnou pamäťou: ploché pole bajtov, ktoré CPU sprístupňuje, nie je to, s čím priamo pracujú programátori: namiesto toho je táto pamäť spravovaná alokátorom pamäte, ako `malloc`, ktorý ju rozdelí na logické objekty rôznej veľkosti, ktoré môžu byť vytvorené a zrušené podľa potreby.
Súborový systém zohráva rovnakú úlohu u perzistentného úložiska. Keďže je však súborový systém zdieľaný medzi mnohými procesmi, a dokonca priamo prístupný užívateľom, musí byť usporiadaný menej dynamicky a intuitívnejšie. Namiesto ukladania číselných ukazateľov na ľubovoľných miestach v súbore (čo je prípad dátových štruktúr uložených v pamäti), objekty v súborovom systéme sú prísne oddelené do objektov, ktoré nesú ukazatele (štruktúru), tj. adresáre, a objekty nesúce dáta, teda bežné súbory. Keďže je každému ukazateľu v adresári priradené meno, tento organizačný princíp má za následok adresárovú hierarchiu, ako sme si ukázali na prednáške 2.

Časť 4.3: Prepínač virtuálneho súborového systému

Typický kernel musí byť schopný pojať pomerne veľké množstvo rôznych implementácií súborových systémov: natívny súborový systém operačného systému je samozrejmosťou, ale určite minimálne ešte súborové systémy, ktoré sú typicky používané na prenositeľných médiách, napríklad USB kľúče (`FAT` alebo `exFAT`) alebo optické médiá (ISO 9660, UDF). Tiež je často žiaduce, aby mohli byť pripojené (mounted) ne-natívne súborové systémy, napríklad pristupovanie k `NTFS` zväzkom na Unixových systémoch. Konečne, existujú súborové systémy, ktoré sú portované na konkrétny operačný systém kvôli svojim žiaducim vlastnostiam, napríklad vysoký výkon, spoľahlivosť alebo škálovateľnosť, a nie iba z dôvodov kompatibility (napr. SGI `xfs` portovaná z IRIX na Linux, alebo Sun/Oracle `ZFS` portovaný zo Solaris na FreeBSD).
Veľa operačných systémov tiež využíva rôzne virtuálne súborové systémy (napríklad `proc` na mnohých UNIX-ových systémoch, alebo `sys` na Linuxe), ktoré tiež prezentujú jednotné rozhranie s ostatnými súborovými systémami.

Vrstva virtuálneho súborového systému

199

- veľa rôznych súborových systémov
- OS sa chce chovať ku všetkým rovnako
- VFS poskytuje interné API v rámci kernelu
- systémové volania súborového systému sú pripojené na VFS

Keďže všetky rôzne súborové systémy poskytujú prakticky rovnakú sémantiku, a potrebujú byť dostupné pre vonkajší svet prostredníctvom jedného jednotného API, kernel by sa k nim tiež ideálne mal správať rovnako. Toto je úlohou prepínača virtuálneho súborového systému alebo vrstvy virtuálneho súborového systému.

VFS z hľadiska OOP

200

- VFS poskytuje abstraktnú triedu, `filesystem` - súborový systém
- každá implementácia súborového systému je odvodená z `filesystem`
  - napr. `class iso9660 : public filesystem`
- každý skutočný súborový systém dostane inštanciu
  - `/home`, `/usr`, `/mnt/usbflash` každý jednu
  - kernel používa abstraktné rozhranie na komunikáciu s nimi

Ak ste sa stretli s objektovo-orientovaným programovaním, táto paradigma by vám mala byť celkom povedomá: máme rozhranie, ktoré abstraktne popisuje súborový systém, z pohľadu kernelu. Každý konkrétny súborový systém implementuje toto abstraktné rozhranie, ale z pohľadu API nezáleží na tom, o ktorý súborový systém sa jedná, keďže všetky poskytujú jednotné API. Na rozdiel od blokovej vrstvy, táto abstrakcia typicky nie je viditeľná mimo kernel. Napriek tomu ide o veľmi dôležitú abstrakciu.

Trieda filesystem

201

```
struct handle { /* ... */ };
struct filesystem
{
    virtual int open( const char *path ) = 0;
    virtual int read( handle file, ... ) = 0;
    /* ... */
}
```

Ak uvážime rozhranie VFS ako C++ triedu, takto nejak by zhruba vyzerala. Konkrétne, handle je možno zložitá dátová štruktúra: abstrakcia popisovača (deskriptora) súboru existuje medzi kernelom a užívateľským priestorom. V rámci kernelu je výrazne menej užitočná, keďže sa, okrem iného, viaže na proces. Keď potrebuje implementácia súborového systému pracovať so súborom, čo potrebuje je referencia na konkrétny i-uzol, ktorý reprezentuje tento súbor (alebo skôr jeho verziu v pamäti), a prípadne iterátor do súboru: ideálne taký, ktorý je efektívnejší ako obyčajný ofset, tj. dátovú štruktúru, ktorá dokáže, bez prechádzania zoznamov blokov v i-uzle, označiť konkrétny blok (alebo bloky), ktoré potrebujú byť načítané z disku.

Operácie špecifické pre súborové systémy

202

- open: nájdi súbor pre jeho sprístupnenie
- read, write – zjavné
- seek: posuň ukazateľ pre čítanie/zapisovanie
- sync: zapíš dáta na disk
- mmap: pamäťovo-mapované IO
- select: notifikácia, že IO je pripravené

VFS operácie čiastočne odzrkadľujú užívateľské API pre prístup k súborom, s vyššie vymenovanými výnimkami. Zoznam vyššie je skrátená verzia VFS rozhrania ako je implementované v kerneli Linuxu. Keďže VFS je interné kernelu, nie je žiadnym spôsobom štandardizované, a rôzne kernely pristupujú k tomuto problému rôznym spôsobom.

Štandardné IO

203

- obvyklý spôsob použitia súborov
- otvorenie súboru
  - operácie pre čítanie a zapisovanie bajtov
- dáta musia byť buffrované v užívateľskom priestore
  - a potom skopírované do/z priestoru kernelu
- nie veľmi efektívne

Štandardné API pre vstup a výstup je založené na operáciách čítania (read) a zapisovania (write). Nanešťastie, tieto nie sú vždy efektívne, keďže dáta musia byť skopírované medzi kešou súborového systému a (súkromnou) pamäťou procesu, ktorý si vyžiadal operáciu. Efektivita v prípade požiadaviek, ktoré obsahujú veľa operácií čítania môže byť značne zvýšená použitím pamäťovo-mapovaného IO. Toto platí rovnako v prípade, že program potrebuje náhodný prístup k obsahu súboru (tj. seek z miesta na miesto) aj v prípade, že proste chce spracovať jeden bajt za druhým.

Pamäťovo-mapované IO

204

- používa virtuálnu pamäť (pozri minulú prednášku)
- správa sa k súboru ako keby to bol swapovací priestor
- súbor je namapovaný do pamäte procesu
  - page fault - výpadok stránky - indikuje, že dáta musia byť načítané
  - tzv. dirty (špinavé) stránky spôsobujú zápis
- dostupné ako systémové volanie mmap

Pamäťovo mapované IO používa subsystém virtuálnej pamäte na zredukovanie množstva kopírovania a prepínania kontextu, ktoré je potrebné u IO operácií. U tejto schémy sú dáta súborov (aspoň na začiatku) zdieľané medzi kešou blokového prístupu (block access cache) a procesom.
Modifikácia dát sa dá riešiť dvoma spôsobmi:

1. modifikácie sú súkromné, napr. keď proces potrebuje iba prispôsobiť dáta ako súčasť spracovania, ale nechce zapísať zmeny do pôvodného súboru,
2. zmeny sú zdieľané, t.j. proces plánuje meniť dáta a následne zapísať tieto zmenené dáta naspäť do súboru.

V prvom prípade sa mapovanie robí spôsobom copy-on-write: stránky z keše sú označené ako read-only (výhradne na čítanie) pre proces a žiadosti o zápis spôsobia, že operačný systém vytvorí novú kópiu dát do fyzickej pamäte. Budúce čítania a zápisy sú potom presmerované na túto novú kópiu. V druhom prípade sú stránky tzv. write-through - modifikácie ovplyvňujú keš a následne aj súbor na disku (keď sú stránky označené ako špinavé zapísané na disk).

Synchronizácia dát

205

- pripomeňme si, že disk je veľmi pomalý
- čakanie, až sa každý zápis dostane na disk, je neefektívne
- ale keď sú dáta držané iba v RAM, čo keď vypadne elektrina?
  - operácia sync zabezpečuje, že dáta sa dostanú na disk
  - často sa používa v implementáciách databázových systémov

Operácia zápisu do súboru je zvyčajne asynchrónna voči fyzickému médiu. To znamená, že systémové volanie write vráti riadenie programu dlho predtým, než sú dáta trvalo zapísané. V prípade poruchy systému alebo výpadku prúdu teda aplikácia nemusí byť schopná získať dáta, ktoré zapísala. Ak je dôležité, aby sa zápis dát dokončil, než bude výpočet pokračovať, operačné systémy na tento účel poskytujú systémové volanie sync (prípadne nejakú jeho obmenu, ako fsync či fdatasync), ktoré zabezpečuje, že všetky nedokončené zápisy sú poslané na zariadenie.

Operácie nezávislé na súborovom systéme

206

- manipulácia so spustiteľnými súbormi
- ovládanie fcntl
- špeciálne súbory
- správa popisovačov súborov (file descriptors)
- zámky súborov

Vrámci kernelu je časť funkcionality súborových systémov nezávislá na konkrétnej implementácii súborového systému, a je namiesto toho implementovaná v iných moduloch jadra. Jedným z príkladov je kód, ktorý sa stará o spustiteľné súbory: predpokladajme, že súborový sys-

tém implementuje pamäťové mapovanie súborov (ktoré je samotné čiastočne implementované na blokovej vrstve a v subsystéme virtuálnej pamäte). Keď je program (uložený ako bežný súbor na nejakom súborovom systéme) spustený, modul operačného systému, ktorý je za to zodpovedný, si vyžiada pamäťové mapovanie súboru skrze VFS, ale zvyšok kódu zaoberajúci sa systémovým volaním `exec` nepotrebuje žiadne znalosti o súborovom systéme.

Spustiteľné súbory

207

- pamäťovo mapované (ako `mmap`)
- môžu byť nastránkované lenivo
- spustiteľné súbory musia byť za behu nemenné
- ale stále môžu byť odstránené (unlinked) z adresára

Spustiteľné súbory majú v niekoľkých ohľadoch špeciálne správanie. Jedným z nich je, že sú často „nastránkované“ lenivo: tento koncept dôkladnejšie preskúmame na 5. prednáške. Ide však o kompromis: znamená to, že keď daný spustiteľný súbor „beží“ – to znamená, keď existuje proces, ktorý práve vykonáva program uložený v danom spustiteľnom súbore – operačný systém musí zakázať akékoľvek zapisovanie do súboru. Inak by obraz programu, ktorý môže čiastočne sídliť v pamäti a čiastočne na disku, mohol byť poškodený, keďže sa stránky už načítané v pamäti kombinujú so stránkami lenivo načítanými z disku.
Keďže však názov súboru nie je vlastnosťou súboru samotného, nie je problém spustené binárky odpojiť (unlink) zo súborového systému. Keď skončí posledný proces, ktorý používa tento spustiteľný súbor, počítadlo referencií pre daný i-uzol spadne na nulu a i-uzol bude recyklovaný (spolu s dátovými blokmi, ktoré používal).

Zamykanie súborov

208

- viacero programov zapisujúcich do rovnakého súboru je zlé
  - operácie budú prichádzať náhodne
  - výsledný súbor bude jeden veľký bordel
- tento problém riešia zámky súborov
  - niekoľko API: `fcntl` vs `flock`
  - rozdiely v sieťových súborových systémoch

Kernel poskytuje mechanizmus na zamykanie súborov, čím umožňuje viacerým procesom bezpečne čítať a zapisovať do zdieľaného súboru (t.j. bez poškodenia súboru kvôli konfliktom v paralelných volaniach `write`). V POSIXe historicky existujú dva samostatné mechanizmy na zamykanie súborov: systémové volanie `flock`, ktoré zamkne celý súbor naraz, a volanie `fcntl`, ktorá dokáže zamknúť rozsah bajtov v rámci súboru.
Tieto API na zamykanie so sebou nanešťastie nesú určité problémy, čím sa dajú ľahko zneužiť, týkajúce sa systémového volania `close` aj sieťových súborových systémov. Nebudeme zachádzať do detailov, ale ak plánujete používať zamykanie súborov, je dobrý nápad si o tom najprv niečo naštudovať.

Systémové volanie `fcntl`

209

- väčšinou operácie súvisiace s deskriptormi
  - synchrónny vs asynchrónny prístup
  - blokujúce vs neblokujúce
  - close on exec: viac v neskoršej prednáške
- jedno z viacerých zamykacích API

Toto je ďalšia časť rozhrania súborového systému, ktorá typicky nemusí interagovať s VFS alebo s konkrétnou implementáciou súborového systému. Okrem sprístupňovania API na zamykanie súborov systémové volanie `fcntl` hlavne interaguje s tabuľkou deskriptorov a rôznymi príznakmi (flags) deskriptorov: môže napríklad byť použité na sprístupnenie synchrónneho IO, to znamená, že každé volanie `write` sa vráti až keď boli dáta odoslané na úložné zariadenie.
Podobne, `fcntl` môže byť použité na nastavenie blokujúceho (predvolené) alebo neblokujúceho módu na súborových deskriptoroch: v neblokujúcom móde nebude systémové volanie v prípade, že je buffer na zápis plný, čakať, až sa miesto uvoľní: namiesto toho oznámi programu, že operáciu nemožno vykonať a že sa má pokúsiť ju zopakovať neskôr. Podobne sa program chová v prípade, že dáta nie sú k dispozícii (napr. u rúry alebo soketu), operácia `read` sa hneď vráti a indikuje, že dáta nie sú dostupné, namiesto aby čakala, kým dáta dostupné budú (ako by to robila v blokujúcom móde).
Príznak close-on-exec inštruuje kernel, aby vykonal na danom deskriptore operáciu `close` v prípade, že sa vykoná systémové volanie `exec`, kým je deskriptor stále otvorený.

Špeciálne súbory

210

- súbory zariadení (device nodes), rúry (pipes), sokety, ...
- iba metadáta špeciálnych súborov žijú na disku
  - zahŕňa to ich prístupové práva & vlastníctvo
  - typ a vlastnosti špeciálneho súboru
- ide iba o iný druh i-uzla
- `open`, `read`, `write`, atď. obchádzajú súborový systém

Špeciálne súbory, ktoré sme si spomínali v 2. prednáške sú tiež prevažne nezávislé na konkrétnom súborovom systéme. Postup čítania a zápisu do takýchto súborov nie je spätý so žiadnym súborovým systémom.
Varianty týchto objektov podobných súborom, ktoré sú pripojené do hierarchie súborového systému (pomenované rúry, UNIXové doménové sokety, zariadenia) sú pre účely súborového systému proste špeciálnym typom i-uzla, a súborový systém si potrebuje uložiť iba ich metadáta. Keď je takýto súbor otvorený, súborový systém si iba vytiahne metadáta a prenechá riadenie inej časti kernelu.

Pripájacie body (Mount Points)

211

- spomeňte si, že existuje iba jeden adresárový strom
- ale je niekoľko diskov a súborových systémov
- súborové systémy môžu byť spojené v adresároch
- koreň jedného sa stáva podadresárom druhého

Konečne, pripájacie body (mount points) sú ďalšou funkciou na VFS vrstve, ktorá nesiaha do jednotlivých implementácií súborových systémov. Dalo by sa tvrdiť, že hlavný dôvod, prečo VFS existuje je, aby viacero rôznych súborových systémov na disku mohlo byť hladko integrovaných do jedného stromu. Systémové volania, ktoré zabezpečujú pripájanie (a odpájanie) sú `mount` a `unmount`, v tomto poradí.

Časť 4.4: UNIX-ový súborový systém

V tejto sekcii si popíšeme, relatívne nízkoúrovňovo, ako sú tradičné UNIX-ové systémy zorganizované, vrátane toho, ako sú veci uložené na disku. Tiež si povieme o niektorých problémoch, ktoré prináša organizácia dát na médiu, kde je nesekvenčný prístup k dátam prísne penalizovaný, a ktoré má pomerne neflexibilné operácie čítania a zápisu (t.j. problémy, ktoré vyvstávajú z toho, že zariadenie poskytuje iba operácie po celých blokoch).

Superblok

213

- drží základné / toplevel informácie o súborovom systéme
- umiestnenie tabuliek i-uzlov
- umiestnenie bitmáp i-uzlov a voľného miesta
- veľkosť bloku, veľkosť súborového systému

Existuje značné množstvo metadát, ktoré si musí súborový systém ukladať na disku. Väčšinou je nepraktické alokovať oblasti s fixnými adresami pre všetky tieto metadáta, takže umiestnenie týchto blokov metadát musí byť uložené niekde priamo v samotnom súborovom systéme. Samozrejme, aspoň časť metadát musí mať fixnú, známu adresu, aby sme mohli zaviesť dátové štruktúry umiestnené v pamäti počas pripájania (`mounting`). Táto fixná časť metadát je známa ako superblok. Keďže je pomerne kľúčový, a zriedka sa mení, väčšinou býva uložený na niekoľkých fixných lokáciách naprieč diskom, pre prípad, že by došlo k poškodeniu hlavnej kópie.

I-Uzly

214

- pripomeňme, že i-uzol je anonymný súbor
  - alebo adresár, alebo špeciálny súbor
- i-uzly majú iba čísla
- adresáre viažu názvy na i-uzly

Doteraz sme sa k i-uzlom správali ako ku abstraktnému konceptu: jednoducho reprezentujú objekt v súborovom systéme, či už je to bežný súbor, adresár, alebo nejaký typ špeciálneho súboru. V tradičných UNIX-ových súborových systémoch je i-uzol tiež reálna, fyzická dátová štruktúra uložená na disku. Keďže je veľkosť i-uzla fixná, môžu byť uložené vo veľkých poliach a indexované pomocou čísel: tieto indexy tvoria základ číslovacieho systému i-uzlov.
Keďže treba do tabuľky i-uzlov často pristupovať, a latencia náhodného prístupu na tradičnom disku závisí od fyzickej vzdialenosti rôznych kusov dát, pole i-uzlov býva často rozdelené na niekoľko kusov, kde každý je uložený na inej fyzickej pozícii. Systém sa potom snaží udržiavať referenciu na i-uzol a dáta, na ktoré i-uzol odkazuje, blízko pri sebe.

Alokácia i-uzlov

215

- často fixný počet i-uzlov
- i-uzly sú buď použité alebo voľné
- voľné i-uzly môžu byť uložené v bitmape
- alternatívy: B-stromy

V prípade, že sú i-uzly uložené v poli, toto pole má obvykle fixnú veľkosť, čo znamená, že niektoré z položiek sú nepoužité. Keďže vytváranie (a mazanie) súborov sú pomerne časté operácie, systém musí byť schopný rýchlo nájsť nepoužitý i-uzol, a tiež rýchlo označiť pôvodne používaný i-uzol za nepoužitý. Prehľadávanie celej tabuľky i-uzlov na nájdenie nejakého nepoužitého by bolo veľmi pomalé, keďže sa ich do jedného sektora na disku vojde iba niekoľko.
Častou metódou na urýchlenie tohto procesu sú bitmapy: okrem vlastných i-uzlov si súborový systém ukladá pole bitov, jeden pre každý i-uzol. Hoci takto musí systém ešte stále lineárne prejsť celé bitmapové pole, aby našiel voľný i-uzol, bude to robiť rýchlosťou 512 až 4096 i-uzlov na sektor.
Uvážte nasledujúci príklad (veľkosť sektora je 512 bajtov): súborový systém je organizovaný do skupín, každá má cca 200MiB dát, a každá má k dispozícii pole 26000 i-uzlov. Toto vo výsledku dáva asi 8KiB dát na jeden i-uzol, čo je priemerná očakávaná veľkosť súboru na tomto súborovom systéme (ak by to bolo menej, súborovému systému môžu dôjsť i-uzly než mu dôjde priestor pre dáta). Každý i-uzol má 128 bajtov, teda na jednom sektore sú uložené 4 i-uzly a celkové pole i-uzlov zaberá 6500 sektorov (alebo 3.25MiB). Na druhú stranu, bitmapa týchto i-uzlov zaberá 51 sektorov (alebo asi 25KiB).

Obsah I-uzlov

216

- konkrétny obsah i-uzla závisí od jeho typu
- i-uzly bežných súborov obsahujú zoznam dátových blokov
  - priame aj nepriame (cez dátový blok)
- symbolické odkazy obsahujú cieľovú cestu
- špeciálne zariadenia popisujú aké zariadenie reprezentujú

Najdôležitejšia časť i-uzla sú pravdepodobne ukazatele na dáta, ktoré obsahuje: t.j. adresy dátových blokov, ktoré obsahujú konkrétne dáta (či už bajty bežného súboru, alebo dátovú štruktúru, ktorá mapuje názvy na čísla i-uzlov, v prípade adresárov).

Priraďovanie dát i-uzlom

217

- niekoľko priamych adries blokov v i-uzle
  - napr. 10 refs, 4K blokov, max. 40 kilobajtov
- nepriame dátové bloky
  - blok plný adries na ďalšie bloky
  - jeden nepriamy blok môže držať cca 2 MiB dát
- rozsahy / extents: súvislý rozsah blokov

V tradičnom prístupe je každý dátový blok zaznamenaný osobitne. Jedna z častých optimalizácií vychádza z pozorovania, že väčšina súborov je uložená v malom počte súvislých rozsahov blokov. Tieto rozsahy sa volajú extents a veľa moderných súborových systémov ukladá namiesto jednoduchého poľa adries blokov zoznam rozsahov (t.j. okrem adresy bloku si udržiavajú číslo, ktoré hovorí súborovému systému koľko súvislých blokov je za ním zabratých). V podstate ide o formu tzv. run-length encoding, typ kompresie súvislých hodnôt. Jednou zjavnou nevýhodou je, že nájdenie adresy bloku, ktorý obsahuje nejaký daný ofset je lineárna operácia (namiesto konštantnej), ale v dátovej štruktúre, ktorá je zvyčajne výrazne menšia (väčšina súborov má iba jedno-ciferné číslo rozsahu, v porovnaní s možno stovkami jednotlivých dátových blokov).

Fragmentácia

218

- vnútorná – nie všetky bloky sú plne využité
  - súbory majú rôznu veľkosť, bloky sú fixné
  - 4100 bajtový súbor potrebuje 2 bloky po 4 KiB
  - toto spôsobuje mrhanie miestom na disku
- vonkajšia – voľné miesto nie je súvislé
  - stáva sa keď sa veľa súborov chce zväčšiť naraz
  - znamená to, že nové súbory sú tiež fragmentované

Vždy keď sú štruktúrované dáta uložené v neštruktúrovanom poli bajtov, musí dochádzať ku kompromisom. Jeden súvisí s efektivitou ukladania: ukladanie súborov nahusto často spôsobí, že veľa operácií bude pomalších, a že potrebné metadáta budú komplikovanejšie.
Jedným z veľmi známych problémov tohto typu je fragmentácia, ktorá má dva základné typy. Vnútorná / interná fragmentácia je spôsobená zarovnaním: je výrazne efektívnejšie začať každý súbor na hranici bloku, a teda alokovať celý počet blokov pre každý súbor. Ale keďže súbory môžu mať ľubovoľnú veľkosť, často sa na konci súboru nachádza nevyužité miesto. Toto miesto je výhradne „režijné“ – nie sú v ňom

uložené žiadne užitočné dáta. Robiť to týmto spôsobom však urýchľuje prístup k súborom. Inými slovami, na konci väčšiny súborov je malý fragment diskového miesta, ktorý nemôže byť využitý (lebo je menší ako minimálna veľkosť, ktorá môže byť alokovaná pre súbor, t.j. jeden blok).

Problémy vonkajšej fragmentácie

219

- výkon: nedá sa použiť rýchle sekvenčné IO
  - programy často čítajú súbory sekvenčne
  - fragmentácia → náhodné IO na zariadení
- veľkosť metadát: nedajú sa použiť dlhé rozsahy - extents

Vonkajšia alebo externá fragmentácia, na druhú stranu, nemrhá miestom priamo. V tomto prípade, ako sú súbory vytvárané a mazané, sa voľné miesto distribuuje naprieč celým súborovým systémom, namiesto toho, aby bolo sústredené v jednom súvislom kuse.
Keď sa potom vytvárajú nové súbory, je viac práce nájsť pre ne miesto, keďže musí byť 'pozliepané' z veľa menších fragmentov. Metadáta narastajú, pretože sa priemerná dĺžka rozsahu zmenšuje a súbory potrebujú viac rozsahov (extents).
Konečne, a čo je možno najdôležitejšie, keď je voľné miesto fragmentované, tak budú neskôr fragmentované aj súbory, ktoré ho využijú. Prístup k týmto súborom potom bude menej efektívny, lebo vždy keď sa narazí na nespojitosť v dátach, systém je vystavený dodatočnej latencii, kvôli nesekvenčnému prístupu k disku.

Adresáre

220

- používajú dátové bloky (ako bežné súbory)
- ale bloky udržiavajú mapovania názov → i-uzol
- moderné súborové systémy používajú hash-e alebo stromy
- formát adresárových dát je závislý na súborovom systéme

Podobne ako u bežných súborov, adresáre používajú dátové bloky na ukladanie svojho obsahu. Avšak, zatiaľ čo obsah bežných súborov je úplne ľubovoľný (t.j. definovaný užívateľom alebo aplikáciou), adresáre sú interpretované samotným operačným systémom. Najjednoduchší formát, ktorý sa dá použiť na ukladanie adresárov je jednoducho zreťaziť všetky dvojice (názov zakončený nulou, číslo i-uzla) jednu za druhou. Samozrejme, toto je veľmi neefektívny spôsob. Väčšina súborových systémov používa sofistikovanejšie dátové štruktúry (hašovacia tabuľka na disku, alebo vyvážený strom).

Vyhľadávanie súboru podľa názvu

221

- často potrebujeme nájsť súbor podľa cesty
- každý komponent znamená vyhľadávanie v adresári
- adresáre môžu obsahovať tisícky záznamov

Jedna z najčastejších operácií v súborovom systéme je vyhľadávanie súboru podľa názvu (file name lookup). Táto operácia je vykonávaná veľakrát pre každú cestu, ktorú potrebujeme tzv. rozlíšiť (resolve) - raz pre každý komponent cesty. Je preto veľmi dôležité, aby sa to dalo robiť rýchlo.
Hoci je dôležité, aby formát na disku umožňoval rýchle vyhľadávanie, aj najrýchlejšia štruktúra na disku bude príliš pomalá: takmer všetky operačné systémy používajú keš v pamäti na zrýchlenie 'častých' vyhľadávaní (t.j. udržiavajú si keš s adresármi alebo záznamami adresárov, ktoré boli nedávno použité pri vyhľadávaní).

Adresáre starého typu

222

- neusporiadaný sekvenčný zoznam záznamov
- nové záznamy sú proste pridané na koniec
- pri mazaní (unlinking) môžu vzniknúť diery
- vyhľadávanie vo veľkých adresároch je veľmi neefektívne

Ako sme spomenuli už skôr, najjednoduchší (ale veľmi neefektívny) spôsob ukladania adresárov je jednoducho si udržiavať lineárny zoznam záznamov v adresári (t.j. dvojice, ktoré mapujú názov na číslo i-uzla). Spôsobuje to veľa problémov a žiaden seriózny súčasný súborový systém tento prístup nepoužíva.

Adresáre založené na hašovaní

223

- v priemere potrebujú čítať iba jeden blok
- často najefektívnejšia možnosť
- rozšíriteľné - extendible hašovanie
  - adresáre sa môžu postupom času zväčšovať
  - postupná alokácia ďalších blokov

Častou alternatívou je použitie hašovacích tabuliek, u ktorých je zvyčajne vyhľadávanie podľa názvu veľmi rýchle: očakávaný výsledok je, že na základe hašu názvu bude prvý sektor, ktorý je načítaný z disku, obsahovať požadovanú dvojicu názov/i-uzol, aj u pomerne veľkých adresárov.
Samozrejme to má aj nevýhody: prechádzanie adresára v, povedzme, abecednom poradí, spôsobí náhodný prístup u IO operácií na disku, keďže adresár je usporiadaný podľa hašu, ktorý je v podstate náhodný. Navyše môžu nastať patologické prípady s veľmi nežiaducim chovaním, v prípade, že sa všetky (alebo väčšina) záznamov v adresári hašujú do rovnakého vedierka.

Adresáre založené na stromoch

224

- samo-vyvažovacie vyhľadávacie stromy
- optimalizované pre prístup po blokoch
- B stromy, B+ stromy, B* stromy
- logaritmický počet čítaní
  - toto je v najhoršom prípade, na rozdiel od hašovania

Ďalšou pomerne bežnou stratégiou je použitie vyvažovacích stromov, ktoré majú mierne horšiu priemernú časovú zložitosť, ale výrazne lepšiu garantovanú zložitosť v najhoršom prípade, v porovnaní s hašovacími tabuľkami. Záznamy sú navyše uložené v usporiadanom poradí, čo zvyšuje efektivitu určitých typov prístupov.

Tvrdé odkazy (Hard Links)

225

- viacero názvov môže odkazovať na rovnaký i-uzol
  - názvy sú dané adresárovými záznamami
  - takéto násobne-pomenované súbory nazývame tvrdé odkazy
  - väčšinou je zakázané vytvoriť tvrdý odkaz na adresár
- tvrdé odkazy nemôžu prekročiť hranice zariadení
  - čísla i-uzlov sú unikátne iba vrámci súborového systému

Okamžitým následkom toho, ako sú súbory a adresáre uložené v súborovom systéme je existencia tvrdých odkazov (hard links). Uvedomte si, prosím, že nejde o nijak špeciálne entity: ide čisto o pomenovanie situácie, kedy niekoľko adresárových záznamov odkazuje na rovnaký

i-uzol. V tomto prípade je každý z 'tvrdých odkazov' od seba navzájom neodlíšiteľných, a jeden súbor sa proste objavuje v adresárovej hierarchii na viacerých miestach.
Keďže si i-uzly udržiavajú počítadlo referencií (odkazov), je zvyčajne možné určiť, že je súbor dostupný pod viacerými rôznymi cestami. A pretože sú súbory zrušené až keď spadne počítadlo referencií na nulu, odstránenie súboru z adresára (nazvané odpojenie - unlinking) môže a nemusí spôsobiť, že súbor bude skutočne vymazaný.

Mäkké odkazy - Soft Links (Symlinks)

- existujú na obídenie limitácie jedného zariadenia
- mäkké odkazy na adresáre sú dovolené
  - môžu spôsobiť cykly v súborovom systéme
- i-uzol mäkkého odkazu obsahuje cestu
  - význam sa môže meniť, keď sa zmenia cesty
- dangling / visiaci odkaz: odkazuje na neexistujúcu cestu

Niekedy je užitočné odkazovať sa na súbor nie cez jeho číslo i-uzla, ale cez jeho cestu. Toto je možné pomocou tzv. mäkkých odkazov (soft links): na rozdiel od tvrdých odkazov ide o skutočné objekty, uložené v súborovom systéme ako špeciálny typ i-uzla. Keď je takýto súbor otvorený, operačný systém namiesto toho vyhľadá súbor podľa cesty uloženej v mäkkom odkaze. Samozrejme, tiež to spôsobuje problémy: cieľová cesta napríklad nemusí existovať (toto je zďaleka najčastejšou chybou).
Ak existuje, i-uzol súboru, ktorý odpovedá danej ceste je získaný obvyklým spôsobom. Na rozdiel od štandardných adresárových záznamov, tento nový i-uzol môže sídliť na inom súborovom systéme.

Voľné miesto

- podobný problém ako u alokácie i-uzlov
  - ale vzťahuje sa na dátové bloky
- cieľ: rýchlo nájsť dátové bloky, ktoré sa dajú použiť
  - tiež: udržať dáta patriace jednému súboru blízko pri sebe
  - tiež: minimalizovať vonkajšiu fragmentáciu
- zvyčajne bitmapy alebo B-stromy

Rovnako ako u i-uzlov, súborový systém musí byť schopný rýchlo nájsť prázdny dátový blok, buď keď sú súbory a adresáre vytvorené, alebo keď sú existujúce zväčšené. Táto úloha je u dátových blokov trochu zložitejšia: u i-uzlov stačí alokovať jeden, a umiestnenie i-uzlov vedľa seba nie je veľmi dôležité (hoci je samozrejme užitočné držať súvisiace i-uzly blízko).
Veľa súborov však potrebuje viac než jeden dátový blok, a je pomerne dôležité, aby sa tieto bloky nachádzali vedľa seba, ak je to možné. Čo je horšie, nie vždy je zrejmé, aký veľký súbor bude, a súborový systém by pravdepodobne mal rezervovať nejaké dodatočné bloky nad rámec blokov nevyhnutných pre prvú sadu zápisov. Vyzbrojený odhadom veľkosti, systém musí nájsť voľné miesto tejto veľkosti, ideálne bez zasahovania do miesta, ktoré majú existujúce súbory nachystané na budúci rast, a ideálne ako jedna súvislá skupina blokov. Toto, samozrejme, nemusí byť možné, v tom prípade sa pokúsi systém rozdeliť súbor do niekoľkých kusov voľného miesta.
V porovnaní s prípadom s i-uzlami sú dátové štruktúry na disku v podstate rovnaké: buď máme bitmapu voľných blokov alebo vyvážené vyhľadávacie stromy. Väčšina rozdielov je vo vstupoch a v algoritmoch.

Konzistencia súborového systému

- čo sa deje pri výpadku energie?
- dáta v RAM bufferoch sú stratené
- IO plánovať môže pre-usporiadať zápisy na disk
- súborový systém môže byť poškodený

Súborové systémy, a dátové štruktúry, s ktorými pracujú, čelia pomerne unikátnej výzve: zmeny, ktoré súborový systém robí na svojich metadátach môžu byť prerušené v ľubovoľnom bode, dokonca uprostred operácie, najčastejšie v dôsledku výpadku energie. Čo je horšie, jednotlivé zápisy, ktoré si systém vyžiadal, môžu byť pre-usporiadané (na zlepšenie výkonu), čo znamená, že nemáme jednoznačný bod, kedy došlo k prerušeniu.
Napríklad, predstavte si vznik bežného súboru: i-uzol musí byť alokovaný (to znamená zápis do bitmapy i-uzlov), naplnený (zápis do samotného i-uzla), a pripojený do adresára (zápis, alebo niekoľko zápisov, do dátovej štruktúry adresára). Ak sú operácie vykonávané v tomto poradí a nastane výpadok energie, môžu nastať dve situácie:

1. bitmapa je aktualizovaná, ale zvyšné operácie sú stratené: v tomto prípade je i-uzol stratený, kým nie je vykonaná kontrola konzistencie, ktorý zistí, že nepoužitý i-uzol je označený ako alokovaný v bitmape,
2. bitmap a samotný i-uzol sú aktualizované, ale nie je pripojený do adresárovej hierarchie, čo má v podstate rovnaký následok, ale vo forme, ktorá je ťažšie detekovateľná počas testu konzistencie.

V oboch prípadoch sú nejaké prostriedky stratené, čo nie je dobré, ale nie je to ani až také hrozné. Uvážte však situáciu, kedy sú zápisy pre-usporiadané a najprv je aktualizovaný adresár. V tomto prípade má súborový systém adresárový záznam, ktorý ukazuje na neinicializovaný i-uzol, čím sprístupňuje súbor, v ktorom je bordel, užívateľom. V závislosti od obsahu neinicializovaného i-uzla sa môžu diať zlé veci, keď sa k súboru pristupuje.
Našťastie dokáže súborový systém vynútiť čiastočné usporiadanie na zápisoch (t.j. garanciu, že všetky nedokončené zápisy sú najprv dokončené, než sa budú diať ďalšie zmeny). Pri starostlivom používaní tohto mechanizmu dokážeme obmedziť množstvo škody, bez výrazného ovplyvnenia výkonu.
Poslednou obrannou líniou je dirty flag - špinavý príznak - v superbloku: keď je súborový systém pripojený (mounted), špinavý príznak je zapísaný na disk, a keď je odpojený, po dokončení všetkých nedokončených zmien, je príznak zmazaný. Súborový systém sa odmietne pripojiť, ak je špinavý príznak už nastavený, čím vynúti kontrolu konzistencie.

Žurnálovanie (Journalling)

- tiež známe ako intent log - log úmyslov
- synchrónne zapíš, čo sa má stať
- oprav skutočné metadáta podľa žurnálu
- spôsobuje výkonnostnú réžiu za behu
  - skracuje dobu kedy je systém nedostupný kvôli rýchlejším kontrolám konzistencie
  - môže tiež zabrániť strate dát

Jednou konkrétnou technikou (ktorá sa spolieha na vynútenie čiastočného usporiadania na zápisoch) je takzvaný log úmyslov, možno lepšie známy z relačných databázových systémov. U tohto prístupu si súborový systém udržiava dedikovanú oblasť na disku a pred začatím akejkoľvek ne-atomickej aktualizácie dátových štruktúr zapíše popis tejto operácie (atomicky) do logu. V tomto prípade, ak je zložená operácia neskôr prerušená, log zachytí úmysel - intent, a môže byť použitý

na rýchle anulovanie zmien, ktoré nekompletná operácia vykonala. To znamená, že vo väčšine prípadov nebude úplný test konzistencie (čo je dosť drahá operácia) potrebný.

Časť 4.5: Pokročilé funkcie

Táto sekcia stručne predstaví niekoľko dodatočných konceptov, ktoré sa často objavujú v kontexte súborových systémov. Nebudeme mať čas venovať sa im do veľkých detailov v tomto kurze, ale je dôležité byť si vedomý, že existujú.

Čo ďalšie dokážu súborové systémy? 231

- transparentná kompresia súborov
- šifrovanie súborov
- de-duplikácia blokov
- snapshots - snímky
- checksums - kontrolné súčty
- redundantné úložisko

Zoznam vyššie sa dá zhrnúť do troch hrubých kategórií funkcií: efektivita ukladania (kompresia, de-duplikácia), bezpečnosť (šifrovanie) a spoľahlivosť (kontrolné súčty, redundantné ukladanie). Schopnosť robiť snímky (snapshot) spadá niekde medzi efektivitu a spoľahlivosť, v závislosti od prípadu použitia a implementácie.

Kompresia súborov 232

- používa jeden zo štandardných kompresných algoritmov
  - musí byť pomerne univerzálny - general-purpose (t.j. nie JPEG)
  - a samozrejme bezstratový
  - napr. `LZ77`, `LZW`, Huffmanovo kódovanie, ...
- pomerne náročné na implementáciu
  - dĺžka súboru sa mení (nepredvídateľne)
  - efektívny náhodný prístup vnútri súboru

Samozrejme, vždy je možné komprimovať súbory na úrovni aplikácií, jednoduchým použitím vhodného kompresného programu. Užívateľ však musí v takomto prípade manuálne dekomprimovať súbor pred jeho použitím a potom ho re-komprimovať nazad. Toto je pomerne nepohodlné. Samozrejme sa to dá zautomatizovať a niektoré programy dokážu robiť (de)kompresiu automaticky pri otváraní súboru, ale nie je to ani veľmi bežné, ani veľmi efektívne.
Alternatívou je, aby si súborový systém implementoval transparentnú kompresiu súborov, t.j. komprimovať dáta, keď sú ukladané na disk, ale prezentovať užívateľským programom dekomprimované dáta pri čítaní, a transparentne komprimovať novo-zapísané dáta pred ich uložením.
Samozrejme, opäť existujú problémy. Najväčšie problémy plynú z faktu, že dáta nikdy nie sú uniformne komprimovateľné, a teda rôzne sekcie súboru budú komprimované rôznym pomerom. To znamená, že posun na špecifický nekomprimovaný ofset bude zložité implementovať. Podobne, zápisy doprostred súboru (ktoré normálne zachovávajú veľkosť súboru) spôsobia, že súbor sa skráti alebo predĺži, čím operáciu značne komplikujú.

Šifrovanie súborov 233

- používa symetrické šifrovanie pre jednotlivé súbory
  - musí byť transparentné pre vyššie vrstvy (aplikácie)
  - symetrická kryptografia zachováva dĺžku
  - šifrované adresáre, dedenie, ap.
- nová sada problémov
  - správa kľúčov a hesiel

Na rozdiel od kompresie, väčšina šifrovacích algoritmov zachováva dĺžku, čím celú záležitosť určitým spôsobom zjednodušuje. Napriek tomu je prítomná dôležitá skutočnosť, a to je zaobchádzanie s tajomstvami, ktoré komplikujú kód iným spôsobom. Okrem toho, na rozdiel od kompresie, v závislosti od prípadu použitia, bude možno systém musieť šifrovať aj metadáta (t.j. nielen obsah súboru). Čo je dôležité, toto zahŕňa adresáre: nielen názvy súborov, ale celé adresárové štruktúry. Zlyhanie kompresie súboru, ktorý užívateľ komprimovať chcel nepôsobí až taký problém. U šifrovania by takáto chyba bola pomerne fatálna.
Keďže je šifrovanie na úrovni blokov výrazne jednoduchšie, a tým menej náchylné na fatálne chyby, väčšina moderných systémov ho používa namiesto šifrovania na úrovni súborového systému, ako tu bolo popísané.

De-duplikácia blokov 234

- niekedy sa rovnaký dátový blok objavuje veľakrát
  - obrazy virtuálnych strojov sú častým príkladom
  - tiež kontajnery a tak ďalej
- niektoré súborové systémy dokážu tieto prípady detegovať
  - interne nasmerujú veľa súborov na rovnaký blok
  - copy on write na zachovanie ilúzie samostatných súborov

Existuje množstvo prípadov použitia kedy buď celé súbory alebo fragmenty súborov sú uložené niekoľkokrát na danom súborovom systéme. V týchto prípadoch môže nájdenie duplikovaných blokov viesť k významnému ušetreniu miesta. V moderných súborových systémoch zvyčajne nie je problém použiť rovnaký dátový blok ako časť viacerých súborov. Ako u ostatných copy-on-write implementácií, takéto zdieľané bloky musia byť špecificky označené, aby sa pri akýchkoľvek zápisoch, ktoré by ich ovplyvnili od-zdieľali (t.j. vytvorila sa súkromná kópia).
De-duplikácia je pomerne drahá, keďže nie je jednoduché nájsť identické bloky: naivné porovnávanie u prechádzania by zabralo $O(n^2)$ času na nájdenie duplikovaných blokov (hoci používa iba konštantnú pamäť). To je, samozrejme, nepraktické, keď si uvedomíme, že $n = 10^9$ je iba cca 4TB priestoru a $n^2 = 10^{18}$ je veľmi obrovské číslo. Hašovacie tabuľky dokážu túto operáciu spraviť v podstate lineárnu, hoci to vyžaduje rádovo 4GiB RAM na 1TB úložného priestoru. Probabilistické algoritmy a dátové štruktúry dokážu ďalej zredukovať konštantné faktory u času aj pamäte.
Vo väčšine prípadov je de-duplikácia offline proces, t.j. taký, ktorý je plánovaný, aby bežal v určitých intervaloch, ideálne počas nízkej záťaže, keďže je pomerne náročný na zdroje, než aby mohol byť vykonávaný nepretržite pri každom zápise (teda online).

[2] V tomto nastavení sa DMA radič reálne stáva správcom zbernice (master) a vykonáva prenos. Hoci je tento efekt v podstate rovnaký, implementácia je o dosť iná, než u DMA založenom na tom, že sa správcami zbernice stávajú periférie, s ktorou sa stretneme neskôr v prednáške.

Snímky (Snapshots) 235

- občas sa hodí byť schopný skopírovať celé súborové systémy
  - ale tiež je to drahé
  - snapshoty poskytujú efektívny spôsob ako to urobiť
- snapshot je zmrazený obraz súborového systému
  - lacný, lebo snapshoty zdieľajú úložné miesto
  - jednoduchšie ako de-duplikácia
  - opäť implementované ako copy-on-write

Ak sú metadáta súborového systému organizované do vhodnej dátovej štruktúry (obvykle variácia B stromov), je možné implementovať copy-on-write nielen pre dátové bloky, ale aj pre tieto metadáta. V tom prípade nie je veľmi ťažké vytvárať efektívne snapshoty.
Vytvorenie snapshotu je sémanticky ekvivalentné vytvoreniu kópie celého súborového systému kým nie je pripojený (mounted): t.j. kópia sa vyhotovuje atomicky vzhľadom na paralelné zápisy. Inými slovami, ak program prepíše dva súbory, povedzme A na A' a neskôr prepíše B na B', ak bola kópia zhotovená neatomicky, v kópii môžu kľudne byť prítomné súbory A a B' (zistiť, ako sa to stane je jednoduché cvičenie). Ďalšou veľkou výhodou snapshotov je, že zo začiatku (t.j. keď sa veľmi nelíši od „živého" súborového systému) zaberá veľmi málo priestoru. Samozrejme, keď začína reálny súborový systém rozchádzať, viac a viac dátových blokov musí byť pri zmene skopírovaných, čo zvyšuje nároky na miesto, a v najhoršom prípade sa približuje priestorovým nárokom štandardnej, úplnej kópie. Časové nároky asociované s touto operáciou sú však amortizované cez dĺžku života snapshotu. Konečne, keď sú snapshoty (výhradne na čítanie) robené pravidelne, dá sa vďaka copy-on-write permanentne ušetriť miesto, keďže v týchto prípadoch bude aspoň časť dát zdieľaná medzi snapshotmi samotnými.

Kontrolné súčty (Checksums) 236

- hardvér je nespoľahlivý
  - jednotlivé bajty alebo sektory môžu byť poškodené
  - môže to nastať bez toho, aby si to hardvér všimol
- kontrolné súčty môžu byť uložené spolu s metadátami
  - a prípadne aj obsahom súboru
  - chráni to integritu súborového systému
- pozor: nie je to kryptograficky bezpečné

Nanešťastie, úložné zariadenia nie sú 100% spoľahlivé, a niekedy môžu potichu poškodiť dáta. To znamená, že čítanie bloku môže vrátiť iné dáta, než aké boli predtým do bloku zapísané. Súborový systém sa môže brániť tým, že bude ukladať kontrolné súčty (napr. CRC) spolu s metadátami (alebo dokonca spolu s dátami). Nezabráni to poškodeniu ako takému, ale aspoň umožní súborovému systému ho rýchlo detegovať. Keď ho detekuje, poškodené objekty môžu byť obnovené zo zálohy: keďže vytvorenie zálohy vždy zahŕňa čítanie súborového systému, u tejto metódy by vždy malo byť takéto poškodenie najprv detegované, než bude dobrá kópia porušených dát stratená.

Redundantné (nadbytočné) ukladanie 237

- ako RAID na úrovni súborového systému
- dátové a metadátové bloky sú replikované
  - prípadne medzi viacerými lokálnymi blokovými zariadeniami
  - ale tiež naprieč clusterom / veľa počítačmi
- drasticky zlepšuje odolnosť voči chybám - fault tolerance

Nakoniec, súborové systémy (najmä distribuované súborové systémy) môžu využívať redundantné ukladanie ako pre dáta, tak aj metadáta. V podstate to znamená, že každý dátový blok a každý metadátový objekt je uložený vo viacerých kópiách, ktoré sú všetky udržiavané synchronizované súborovým systémom, pričom je každá kópia uložená na inom fyzickom úložnom zariadení. V niektorých prípadoch to môže znamenať viacero počítačov, čo zvyšuje odolnosť voči chybám na nie-diskových komponentoch (ktoré obvykle odstavia celý počítač).

Review Questions 238

13.What is a block device?
14.What is an IO scheduler?
15.What does memory-mapped IO mean?
16.What is an i-node?

Časť 5: Procesy, vlákna & plánovanie

V tejto prednáške sa pozrieme na 2 základné zdroje, ktoré nám počítač ponúka, a ktoré každý program potrebuje: procesor a pamäť. Hlavnou otázkou bude, ako operačný systém dosahuje ilúziu, že každé vlákno má zdanlivo vlastný procesor a každý proces vlastnú pamäť (toto máme na mysli pod výrazom multiplexing – premeniť jeden fyzický zdroj na väčšie množstvo jeho virtuálnych inštancií).

Obsah prednášky 240

1. procesy a virtuálna pamäť
2. plánovanie vlákien
3. prerušenia a hodiny

Prednáška bude mať 3 časti. Najprv sa pozrieme na virtuálnu pamäť, a jednotku izolácie pamäte v operačnom systéme: proces. Potom sa pozrieme na zdieľanie CPU (procesora) a jednotku CPU alokácie, vlákno. Nakoniec si bližšie rozoberieme ako je zdieľanie CPU implementované z hľadiska hardvéru.

Časť 5.1: Procesy a virtuálna pamäť

Pravek: Dávkové (Batch) systémy 242

- prvé počítače vykonávali v jednom momente jeden program
- programy boli plánované s predstihom
- bavíme sa o diernych štítkoch ap.
- a počítačoch, ktoré zaberali celú miestnosť

Prvá generácia počítačov bola výhradne sekvenčná: v jednom momente mohol bežať jeden program a to bolo tak všetko. Na vykonanie ďalšieho programu musel prvý najprv skončiť.

Minulosť: Zdieľanie času

243

- "mini" počítače mohli vykonávať programy interaktívne
- teletype terminály, obrazovky, klávesnice
- viacero užívateľov súčasne
- a teda, viacero programov súčasne

Počítače boli v tej dobe pomerne drahé, a programy začali ponúkať viac interaktivity. To znamená, že program dokázal interagovať s obsluhou tým, že sa napríklad pýtal otázky a potom čakal až mu budú poskytnuté vstupné hodnoty. Tieto dve skutočnosti spôsobili, že sekvenčná, jednoprogramová paradigma sa stala značne nepraktickou.
V tomto bode vznikli systémy založené na zdieľaní času (time sharing), kedy počítač dokázal vykonávať viacero programov v zdanlivo rovnakom čase, tým, že rýchlo prepínal z jedného programu na druhý. Hneď, ako je toto možné, dáva zmysel umožniť viacerým užívateľom interagovať s rovnakým počítačom (každý užívateľ cez iný program).

Procesy: Skorý pohľad

244

- proces je vykonávaný/spustený program
- procesov môže byť viacero
- procesu patria rôzne zdroje
- každý proces patrí konkrétnemu užívateľovi

Keďže tradičné programy sú vnútorne sekvenčné, pôvodná myšlienka procesu zahŕňa oba typy základných zdrojov: pamäť a procesor. Pri normálnej prevádzke proces P beží po nejaký čas na procesore, než je prerušený a systém sa prepne na vykonávanie nejakého iného procesu. V nejakom neskoršom bode bude proces P prebudený a bude pokračovať vo výpočte tam, kde prestal.

Zdroje procesu

245

- pamäť (adresný priestor)
- čas procesora
- otvorené súbory (deskriptory)
  - tiež pracovný adresár
  - tiež sieťové spojenia

Najzákladnejším zdrojom asociovaným s procesom je pamäť: je to miesto, kde je uložený program samotný (inštrukcie), a tiež kde sa nachádzajú jeho dáta (statické aj dynamické). Táto pamäť je typicky súkromná: jeden proces nemôže vidieť pamäť iného procesu.

Segmenty pamäte procesu

246

- text programu: obsahuje inštrukcie
- dáta: statické a dynamické dáta
  - s osobitnou sekciou iba na čítanie
- pamäť vyhradená pre zásobník volaní
  - návratové adresy
  - automatické premenné

Pamäť programu je organizovaná do funkčne oddelených segmentov. Tradične boli tieto segmenty súvislé kusy adresného priestoru, ale v moderných operačných systémoch to nie je tak úplne pravda (a celá myšlienka segmentov tak stráca na užitočnosti). Typické rozdelenie je na text programu, ktorý obsahuje inštrukcie programu (t.j. spustiteľný kód, typicky vygenerovaný prekladačom), zásobník (stack), ktorý udržiava C-čkový zásobník (t.j. hodnoty lokálnych premenných a návratové adresy, spolu s ďalšími informáciami) a nakoniec data segment (dátový), ktorý nie úplne prekvapivo uchováva dáta.
V moderných systémoch nemusí žiaden z týchto segmentov byť súvislý: text pozostáva z niekoľkých mapovaní: jedno pre hlavný program a jedno pre každú zdieľanú knižnicu, ktorú používa. Tieto mapovania nepotrebujú byť vedľa seba (susediace) vo virtuálnom adresnom priestore (o to menej vo fyzickej pamäti). Podobne, keďže jeden proces môže obsahovať niekoľko vlákien (o tom viac neskôr), môže byť prítomných niekoľko zásobníkov, opäť nie nutne alokovaných vedľa seba. Konečne, statické (a obzvlášť tie iba na čítanie) dáta tiež pochádzajú zo spustiteľných obrazov (executable images), takže môže existovať jedno mapovanie na spustiteľný súbor, rovnako ako s u segmentu text. Dynamické dáta sú plne tvarovo neobmedzené: implementácia `malloc`-u v `libc` si vyžiada pamäť od operačného systému podľa potreby, pričom virtuálne adresy sú často priradené náhodne každej novej oblasti pamäte.

Pamäť Procesu

247

- každý proces má svoj vlastný adresný priestor
- to znamená, že procesy sú od seba navzájom izolované
- vyžaduje to aby CPU malo MMU
- implementované pomocou stránkovania (stránkovacie tabuľky)

Vymedzujúcim znakom moderného procesu je jeho adresný priestor: virtuálne adresy alokované procesu sú súkromné pre tento proces a teda neviditeľné pre všetky ostatné procesy. Dáta sú väčšinou uložené v RAM, ale každý proces (typicky) vidí iba malú časť fyzickej pamäte – väčšina jednoducho nie je viditeľná v jeho virtuálnom adresnom priestore. V zákulisí je táto separácia adresných priestorov postavená na stránkovaní, ktoré je poskytované cez MMU (memory management unit) v CPU.

Prepínanie procesov

248

- prepínanie procesov znamená prepínanie stránkovacích tabuliek
- fyzické adresy sa nemenia
- ale mapovanie virtuálnych adries áno
- veľká časť fyzickej pamäte je nenamapovaná
  - môže byť úplne nealokovaná (nepoužitá)
  - alebo patriť iným procesom

V typickom časovo-zdieľanom operačnom systéme (čo zahŕňa v podstate každý moderný všeobecný (general-purpose) OS), je ilúzia prakticky neobmedzenej súbežnosti (schopnosti spúšťať ľubovoľné množstvo procesov naraz, nezávisle od počtu dostupných CPU jadier) dosiahnutá rýchlym prepínaním výpočtu z jedného procesu na druhý. Alebo, aby sme boli presnejší, výpočet je prepnutý z jedného vlákna na druhé (za chvíľu si o tom povieme viac), ale tieto vlákna často patria iným procesom. V tom prípade musí operačný systém inštruovať procesor, aby prepol do virtuálneho adresného priestoru nového procesu. Samozrejme v tomto procese nie je fyzická pamäť žiadnym spôsobom preusporiadaná – bolo by to príliš drahé. Namiesto toho je MMU inštruovaná, aby začala používať inú množinu mapovacích pravidiel (stránkovacích tabuliek), ktoré mapujú virtuálne adresy na fyzické adresy. Zvyčajne je minimálny prienik medzi fyzickými adresami, ktoré boli viditeľné v pôvodnom virtuálnom adresnom priestore a tými, ktoré sú viditeľné v novom: ako sme už zmienili, virtuálny adresný priestor je takmer celý súkromný pre každý proces.
Preto je väčšina fyzickej pamäte z pohľadu nejakého daného procesu

neprístupná: buď nie je použitá, alebo patrí inému procesu.

Stránkovanie a TLB

249

- preklad adries je pomalý
- nedávno použité stránky sú uložené v TLB
  - skratka pre Translation Look-aside Buffer
  - veľmi rýchla hardvérová keš
- TLB musí byť vyliata/flushed pri prepnutí procesu
  - je to dosť drahé (mikrosekundy)

Jedna z dôležitých vecí, na ktoré treba brať ohľad, je cena prepínania procesov. Na povrchu to zahŕňa prepínanie vlákna (ktoré, ako uvidíme neskôr, pozostáva z uloženia obsahu užívateľsky-prístupných registrov do pamäte a načítania novej množiny hodnôt z inej oblasti v pamäti) a dodatočný zápis do registra (na načítanie novej stránkovacej tabuľky do MMU).
Nanešťastie, pod povrchom tento jediný zápis do registra spôsobuje kaskádu ďalších následkov: konkrétne všetky moderné procesory používajú veľmi rýchlu keš na ukladanie nedávnych prekladov adries (t.j. ktoré fyzické adresy odpovedajú malej množine nedávno použitých virtuálnych adries), tiež známu ako TLB (translation look-aside buffer – vyrovnávacia pamäť pre preklad).
TLB je potrebná, lebo inak preklad virtuálnej adresy na fyzickú znamená niekoľko prístupov do hlavnej pamäte, kde sú stránkovacie tabuľky uložené. V moderných počítačoch každý takýto prístup zaberie stovky alebo dokonca tisícky cyklov procesora – zjavne by opakovanie tohto procesu u každého prekladu adresy spôsobilo, že by výpočet bol extrémne pomalý.
Pri prepnutí procesu sa mení mapovanie z virtuálnych adries na fyzické adresy a informácie uložené v TLB sa stanú neplatnými, keďže odkazujú do predchádzajúceho adresného priestoru. Najjednoduchšie riešenie je jednoducho vymazať všetko, čo je v TLB uložené. Táto operácia je veľmi rýchla, keďže TBL je implementované použitím veľmi rýchlej pamäte na čipe. Väčšina ceny operácie pochádza z nasledovného prekladu adries, ktoré nemôžu byť získané z (momentálne prázdneho) TLB a musíme urobiť niekoľko prístupov do hlavnej pamäte (alebo, v závislosti od vyťaženia keše (cache pressure), do jednej z všeobecných (general-purpose) keší – L1, L2 alebo L3).
Mimochodom, moderné procesory často používajú tagované (označené) TLB, ktoré zefektívňujú celkový proces, ale to je výrazne nad rámec tohto kurzu.

Vlákna

250

- moderná jednotka CPU plánovania
- každé vlákno beží sekvenčne
- jeden proces môže mať viacero vlákien
  - takéto vlákna zdieľajú jeden adresný priestor

Zatiaľ čo procesy sú základnou jednotkou správy pamäte u operačných systémov, výpočet je zachytený vláknami. Každý proces má aspoň jedno vlákno, ale môže mať viac než jedno. Účtovanie procesorového času sa robí na úrovni procesov, zatiaľ čo výpočet sa viaže na vlákna, keďže vlákna sú to, čo procesor spúšťa.

Čo je to vlákno?

251

- vlákno je sekvencia inštrukcií
  - inštrukcie závisia na výsledku predchádzajúcich inštrukcií
- rôzne vlákna vykonávajú rôzne inštrukcie
  - na rozdiel od SIMD alebo viac-jadrových jednotiek (GPU)
- každé vlákno má svoj vlastný zásobník

Na vlákno môžeme nahliadať ako na prúd inštrukcií: v podstate ako si predstavujeme tradičný, sekvenčný program. V rámci vlákna môže inštrukcia slobodne používať výsledky predchádzajúcich inštrukcií (dátové závislosti) a robiť rozhodnutia (založené na týchto výsledkoch) o tom, ktoré inštrukcie má ďalej vykonávať (vetvenie – branching, alebo tok riadenia – control flow). Toto je spôsob, akým jadro procesora vykonáva program (aspoň z pohľadu programátora). Takže v ľubovoľnom čase každé jadro procesora vykonáva jedno vlákno. Vlákna bežiace súbežne na rôznych jadrách alebo rôznych CPU spolu nemusia nijak súvisieť (nemusia ani patriť rovnakému procesu: každé jadro má svoju vlastnú MMU a každá z nich môže používať inú stránkovaciu tabuľku, teda každé jadro môže vykonávať vlákno v inom adresnom priestore). Samozrejme, tiež to znamená, že každé vlákno potrebuje svoj vlastný zásobník volaní (t.j. pamäť, ktorá ukladá lokálne premenné programu jazyka C, spolu s návratovými adresami a ďalšími meta informáciami súvisiacimi s vykonávaním programu).

Zdieľanie času procesora

252

- CPU čas je rozdelený na tzv. časové diely (time shares)
- časové diely sú niečo ako pamäťové rámce
- výpočet procesu je niečo ako pamäťové stránky
- procesy sú alokované do týchto časových dielov

Okrem pamäte je ďalšou hlavnou komoditou počítača výpočetný čas (computation time). Rovnako ako pamäť ho je obmedzené množstvo a musí byť rovnomerne rozdelený medzi viacero vlákien (a procesov). Keďže rôzne inštrukcie potrebujú výrazne rôzne časy na svoje vykonanie, alokácia výpočtu je založená na čase namiesto počtu inštrukcií. Ako bonus sa čas jednoduchšie meria.

Viacero CPU

253

- vykonávanie/výpočet vlákna je sekvenčný
- jedno CPU = jedna sekvencia inštrukcií naraz
- fyzický limit na rýchlosť CPU → viacero jadier
- viac CPU jadier = vyššia priepustnosť (throughput)

S časovým zdieľaním je možné vykonávať ľubovoľné množstvo vlákien na jednom CPU jadre. Fyzické limity návrhu CPU však v poslednej dekáde spôsobili, že je výrazne jednoduchšie postaviť procesory, ktoré môžu vykonávať dvojnásobný počet vlákien (zdvojnásobením počtu jadier), namiesto toho, aby vykonávali rovnaký počet vlákien dvojnásobnou rýchlosťou. Kým existuje dostatok vlákien, ktoré sú pripravené na vykonávanie, celková priepustnosť výpočtu systému sa aj tak zdvojnásobí. Samozrejme to vyvíja tlak na softvér, ktorý musí byť rozdelený do viac a viac vlákien, aby dokázal saturovať výpočetný výkon moderného procesora.

Moderný pohľad na proces

254

- v modernom pohľade je proces adresný priestor
- vlákna sú správnou abstrakciou plánovania

- proces je jednotkou správy pamäte
- vlákno je jednotkou výpočtu
- starý pohľad: jeden proces = jedno vlákno

Na zopakovanie, existujú dve hlavné abstrakcie, ktoré sa sústredia na pamäť a výpočet. Zatiaľ čo historicky operačné systémy predpokladali, že 1 proces = 1 vlákno (a teda staršie učebnice o operačných systémoch sa môžu riadiť týmto návrhom), toto už v moderných systémoch nedáva zmysel.
Namiesto toho sú vlákna abstrakciou, ktorá zahŕňa samotný výpočet (sekvencia inštrukcií, ktoré sa majú vykonať), zatiaľ čo procesy pokrývajú pamäť a väčšinu ostatných zdrojov.

Fork

255

- ako vytvárame nové procesy?
- fork-ovaním existujúcich procesov
- fork vytvára identickú kópiu procesu
- výpočet pokračuje v oboch procesoch
  - každý z nich dostane inú návratovú hodnotu

Poďme sa pozrieť na procesy z perspektívy programov (a užívateľov). Prvá vec, ktorou sa budeme zaoberať, je ako procesy vznikajú: na systémoch typu POSIX sa toto deje takmer výhradne použitím systémového volania `fork`, ktoré jednoducho vytvorí identickú kópiu aktuálneho procesu. Procesy sú potom veľmi mierne rozlíšené: v jednom `fork` vráti inú hodnotu, a v tabuľke procesov je jeden považovaný za rodiča (parent) a druhý za dieťa (child). Návratová hodnota volania `fork` povie každému procesu, ktorý z nich dvoch je.

Lenivý Fork

256

- stránkovanie môže urobiť `fork` relatívne efektívnym
- začneme skopírovaním stránkovacích tabuliek
- najprv sú všetky stránky označené ako iba na čítanie
- procesy začnú zdieľajúc pamäť

Vytvorenie kópie celého procesu by bolo pomerne drahým cvičením. Ako sme to videli predtým u súborových systémov, existujú však určité triky (založené na implementačnej technike copy-on-write – kopírovanie pri zápise), ktoré umožňujú, aby `fork` samotný bol relatívne efektívny. V čase volania `fork` je jediná vec, ktorá musí byť skopírovaná stránkovacia tabuľka, ktorá je výrazne menšia, než celý adresný priestor daného procesu.
Zároveň sú všetky stránky v oboch kópiách stránkovacej tabuľky označené ako výhradne na čítanie, keďže teraz sú zdieľané medzi dvomi procesmi, ktoré by nemali mať prístup do adresného priestoru toho druhého. Samozrejme, kým iba čítajú z pamäte, nerobí rozdiel, či existujú dve fyzické kópie alebo jedna zdieľaná kópia.

Lenivý Fork: Chyby (Faults)

257

- zdieľaná pamäť sa stane copy on write
- fault/chyba keď sa niektorý z procesov pokúsi zapísať
  - pamätajte, že pamäť je označená ako výhradne na čítanie (read-only)
- OS skontroluje, či do pamäte má byť dovolené zapisovať
  - ak áno, urobí kópiu a dovolí zápis

Hneď ako sa niektorý z procesov pokúsi vykonať zápis do pamäte, toto vyvolá chybu (tzv. fault): stránky sú označené ako výhradne na čítanie (read-only) v stránkovacej tabuľke. Keď to nastane, procesor spustí obsluhu chyby (fault handler) – podprogram zodpovedný za ošetrovanie chýb (je súčasťou kernelu), aby vyriešila túto situáciu. Obsluha chyby potom konzultuje svoje dátové štruktúry (ktoré môžu byť čiastočne zabudované v stránkovacej tabuľke) pre rozlíšenie skutočných chýb (napr. proces sa pokúsil zapísať do pamäte, ktorá bola skutočne výhradne na čítanie) od udalostí copy-on-write.
V druhom prípade je stránka sémanticky na čítanie aj zápis, ale je označená ako výhradne na čítanie, pretože viacero procesov používa jednu fyzickú kópiu. V tomto bode obsluha chyby rozdelí mapovanie: vytvorí novú fyzickú kópiu dát (t.j. alokuje pre ňu nový rámec) a upraví stránkovaciu tabuľku procesu, ktorý sa pokúsil zapísať, tak aby virtuálna adresa, ktorá to spôsobila sa prekladala, aby ukazovala na túto novú fyzickú kópiu.
Konečne, CPU je inštruované, aby reštartovalo inštrukciu, ktorá to spôsobila: keďže je záznam v stránkovacej tabuľke teraz označený ako na čítanie aj zápis, tzv. read-write (ukazujúci na nové fyzické umiestnenie), inštrukcia sa vykoná bez ďalších problémov.

Init

258

- na UNIX-e je `fork` jediný spôsob ako vytvoriť proces
- ale `fork` rozdelí existujúce procesy na 2
- prvý proces je špeciálny
- je priamo vytvorený kernelom pri bootovaní

Skôr sme si tvrdili, že `fork` je v podstate jediný spôsob, ako sa dá vytvoriť nový proces. Samozrejme, toto nemôže byť úplne pravda, keďže `fork` môže byť spustený iba z existujúceho procesu. Preto existuje jeden špeciálny proces, tiež nazvaný `init` alebo `pid 1`, ktorý je priamo vytvorený kernelom pri bootovaní. Všetky ostatné procesy v bežiacom systéme však pochádzajú z tohto originálneho procesu cez sekvenciu fork-ov.

Identifikátor procesu

259

- procesy majú priradené číselné identifikátory
- tiež známe ako PID (Process ID)
- používajú sa pri správe procesov
- použité pri volaniach ako `kill` alebo `setpriority`

Pre zjednodušenie správy procesov je každému procesu priradený číselný identifikátor (známy ako PID, skratka pre process identifier – identifikátor procesu). Systémové volania pre správu procesov dostávajú toto číslo ako argument.
Tradične je 'namespace' (menný priestor) procesov globálny: PID identifikuje proces globálne (na rozdiel od, napríklad, popisovača súboru – deskriptora – ktorý je lokálny každému procesu). To znamená, že pre všetkých užívateľov a všetky procesy dané číslo vždy reprezentuje rovnaký proces (než je ukončený). Upozorňujem, že pri virtualizácii založenej na kontajneroch toto nie je striktne pravda, keďže rôzne

kontajnery dostanú rôzne PID menné priestory, hoci zdieľajú rovnaký kernel.

Proces vs spustiteľný súbor

260

- proces je dynamická entita
- spustiteľný súbor je statický súbor
- spustiteľný súbor obsahuje počiatočný obraz pamäte
  - nastaví rozloženie (layout) pamäte
  - a obsah segmentov text a data

V predchádzajúcich prednáškach sme letmo načrtli tému spustiteľných súborov (executables): ide o súbory, ktoré obsahujú programy. Je veľmi dôležité porozumieť rozdielu medzi spustiteľným súborom (dalo by sa povedať program v pokoji) a procesom (program, ktorý je v procese vykonávania sa).
Spustiteľný súbor v podstate obsahuje počiatočný obraz pamäte procesu: keď sa program začne vykonávať, kernel vyplní jeho virtuálny adresný priestor dátami zo spustiteľného súboru, najdôležitejšie časti sú segment text, ktorý bude obsahovať inštrukcie, a segment data, kde budú statické dáta programu. Ďalej spustiteľný súbor obsahuje (virtuálnu) adresu vstupného bodu (entry point) programu: ide o adresu prvej inštrukcie programu, ktorá sa má vykonať.

Exec

261

- na UNIX-e sú procesy vytvorené cez fork
- ako ale spúšťať programy?
- exec: načítanie nového spustiteľného súboru do procesu
  - toto úplne prepíše pamäť procesu
  - výpočet začína od vstupného bodu
- spúšťanie programov: fork + exec

Keď je proces vytvorený, jeho pamäť nie je vyplnená zo spustiteľného súboru: ide o klon existujúceho procesu. Pre spustenie nového programu musí existujúci proces zavolať systémové volanie exec, ktoré jednoducho prepíše celú pamäť stávajúceho procesu obsahom spustiteľného súboru (executable). Iba premenné prostredia a argumenty príkazovej riadky (ktoré sú obe uložené v pamäti procesu) sú skopírované do nového adresného priestoru.
Obvyklá operácia pre 'spustenie nového programu' (napr. to čo sa stane keď užívateľ spustí príkaz v shelli) je teda implementovaná ako príkaz fork, nasledovaný exec v detskom (child) procese.

Časť 5.2: Plánovanie vlákien

V tejto sekcii sa detailnejšie pozrieme na to, ako sa operačné systémy rozhodujú, ktoré vlákno kedy spustiť a na ako dlho.

Čo je to plánovač (scheduler)?

263

- plánovač má dve nesúvisiace úlohy
  - naplánovať kedy ktoré vlákno spustiť
  - reálne prepínať vlákna a procesy
- obvykle súčasť kernelu
  - dokonca aj u mikrokernelových operačných systémov

Plánovač (scheduler) je komponent kernelu, ktorý plní dve základné roly: plánovanie a prepínanie vlákien (vrátane preempcie).

Prepínanie vlákien

264

- vlákna rovnakého procesu zdieľajú adresný priestor
  - je potrebné čiastočné prepnutie kontextu
  - iba stav registrov musí byť uložený a obnovený
- žiadne vylievanie TLB – menšia réžia

Na prepnutie vykonávania jedného vlákna na druhé, v rámci jedného procesu, stačí uložiť súčasný stav užívateľsky-viditeľných registrov do pamäte, a načítať stav, ktorý odpovedá druhému vláknu (stav, ktorý bol uložený keď bolo naposledy prerušené a pozastavené). Keďže je adresa (vrcholu) zásobníka uložená v registri, jednoduché nahradenie hodnôt v registroch zo zálohy má tiež efekt prepnutia aktívneho zásobníka.
V tomto prípade TLB nemusí byť vyliate (flushed), čím je prepnutie efektívne, aj keď nie úplne zadarmo (stále je tu malá pokuta za predpovedanie skokov – branch prediction a za špekulatívne vykonávanie kódu – speculative execution).

Fixný vs. dynamický plán

265

- fixný plán = všetky procesy známe vopred
  - užitočné iba pri špeciálnych / embedded systémoch
  - môže šetriť zdroje
  - plánovanie nie je súčasťou OS
- väčšina systémov používa dynamické plánovanie
  - čo sa má spúšťať ďalšie je rozhodnuté periodicky

Pokiaľ ide o plán vykonávania (schedule), t.j. plán špecifikujúci, ktoré vlákno sa má vykonávať v ktorom slote (slotoch), sú dva základné typy:

1. statické plány sú vypočítané vopred, pre fixnú množinu vlákien s vopred určenými prioritami a s ich relatívnymi výpočetnými nákladmi tiež známymi vopred,
2. dynamické plány, kde vyššie uvedené informácie nie sú známe.

Pri vykonávaní v statickom pláne je behový plánovač (runtime scheduler) obzvlášť jednoduchý a robustný. Tento prístup však nie je vhodný pre iné než najjednoduchšie prípady, a je väčšinou prítomný iba u vysoko spoľahlivých embedded systémov (high-assurance systems).
Na druhú stranu, dynamický plánovač umožňuje, aby boli vlákna a procesy vytvorené ad-hoc, za behu. Podobne, priority vlákien môžu byť určené (a zmenené) ako je potrebné. Plánovač periodicky vyhodnotí situáciu a rozhodne sa, čo má spustiť buď teraz, alebo vo veľmi blízkej budúcnosti.

Preemptívne plánovanie

266

- úlohy – tasks (vlákna) bežia ako keby vlastnili CPU
- OS im násilne odoberie CPU
  - toto sa nazýva preempcia
- výhoda: chybný program nemôže zablokovať systém
- trochu menej efektívne než kooperatívne

U väčšiny moderných operačných systémov je plánovač preemptívny, t.j. môže pozastaviť vlákna podľa vlastného uváženia, bez kooperácie samotných vlákien. Pri tomto prístupe si nemusí užívateľský softvér, v princípe, byť vedomý toho, že beží na časovo zdieľanom systéme: keď je vlákno prerušené (preempted) a neskôr obnovené, jeho vykonávanie pokračuje tam, kde prestalo, bez akéhokoľvek viditeľného efektu na strane bežiaceho programu.
Veľkou výhodou preemptívneho plánovania je, že nekooperatívne (alebo chybné) programy nemôžu ohroziť systém ako celok: operačný systém sa môže, v akomkoľvek bode, rozhodnúť pozastaviť proces, ne-

závisle od kódu, ktorý v tom čase vykonáva.

Kooperatívne plánovanie

267

- vlákna (úlohy – tasks) kooperujú pri zdieľaní CPU
- každé vlákno sa musí CPU explicitne vzdať – yield
- ak je to navrhnuté správne, môže to byť veľmi efektívne
- ale zlý program môže jednoducho zablokovať celý systém

Iný prístup je známy ako kooperatívne plánovanie. V tomto prípade neexistuje žiadna preempcia: aktuálne bežiace vlákno sa musí vzdať, tzv. yield procesora dobrovoľne. Umožňuje to optimalizovať prepínanie úloh, ale tiež to znamená, že program, ktorý sa nevzdá procesora proste bude neprerušene pokračovať vo vykonávaní.
Zatiaľ čo všeobecné (general-purpose) operačné systémy už globálne nepoužívajú kooperatívne plánovanie, niektoré programy implementujú takzvané zelené vlákna, ktoré sú plánované kooperatívne – plánovanie týchto vlákien je implementované plne v užívateľskom priestore. Zvyčajne je veľké množstvo takýchto kooperatívnych 'zelených' vlákien pridelených na malé množstvo 'reálnych' vlákien OS (ktoré sú plánované preemptívne kernelom).

Plánovanie v praxi

268

- kooperatívne na Windows 3.x pre všetko
- kooperatívne pre vlákna na klasickom Mac OS
  - ale preemptívne pre procesy
- preemptívne na v podstate všetkých moderných OS
  - vrátane real-time a embedded systémov

Posledný bežne používaný operačný systém, ktorý používal kooperatívne plánovanie na úrovni operačného systému bol 'klasický' Mac OS (pred OS X), ale pre procesy používal preemptívne plánovanie. Takto mohlo prepínanie vlákien vrámci procesu využiť lepšiu efektivitu, zatiaľ čo procesy sa nemohli navzájom blokovať. Posledný bežne používaný systém s plne kooperatívnym plánovaním bol MS Windows 3.11, vydaný v roku 1993.

Čakanie a dobrovoľné odovzdanie procesoru

269

- vlákna často potrebujú čakať na zdroje alebo udalosti
  - tiež môžu používať softvérové časovače
- čakajúce vlákno by nemalo spotrebúvať CPU čas
- takéto vlákno sa vzdá procesoru (yield)
- je pripísané na zoznam a neskôr prebudené kernelom

U väčšiny programov je bežné, že vlákno nemôže pokračovať vo svojom vykonávaní, než nastane nejaká udalosť, alebo bude dostupný nejaký zdroj. V takomto prípade je nežiaduce, aby vlákno čakalo aktívne, t.j. bežalo v cykle, ktorý kontroluje, či môže pokračovať. Namiesto toho kernel poskytuje mechanizmy, ktoré umožňujú vláknam byť pozastavené (suspended) než udalosť nastane, alebo než bude zdroj k dispozícii (kedy budú obnovené kernelom). Tento proces nie je úplne bez réžie, ale pokiaľ nie je čakanie veľmi krátke (niekoľko desiatok CPU cyklov), pozastavenie vlákna dosiahne výrazne lepšiu celkovú priepustnosť (throughput) systému.

Fronty vlákien (run queues)

270

- vlákna, ktoré môžu bežať (nie-čakajúce) sú dané do fronty
- môže byť prioritná, round-robin alebo fronta iného typu
- plánovač si vyberie vlákno z fronty bežiacich vlákien
- vlákna, ktoré sú prerušené (preempted) sú vrátené nazad

Sú dva dôvody, prečo je vlákno pozastavené, t.j. už nebeží: buď čaká na nejakú udalosť (pozri vyššie) alebo jeho časový slot vypršal a bolo prerušené. V druhom prípade bude vlákno vložené do fronty bežiacich vlákien – run queue, čo je zoznam vlákien, ktoré sú pripravené bežať, obvykle usporiadané podľa svojej dynamickej priority, ide o číslo ktoré indikuje, ako skoro má vlákno opäť bežať a je vypočítané plánovačom. Vždy, keď je vlákno pozastavené, plánovač z fronty bežiacich vlákien vyberie ďalšie vlákno, ktoré sa má vykonávať.

Priority

271

- akú časť CPU má vlákno dostať?
- priority sú statické a dynamické
- dynamická priorita je menená za behu vlákna
  - rozhoduje o tom systém / plánovač
- statická priorita je priradená užívateľom

Všetky vlákna nie sú rovnako dôležité a niektoré by mali dostať prioritu pred ostatnými, keď potrebujú bežať. Toto sa dosahuje cez kombináciu priorít: statická priorita je priradená každému vláknu užívateľom. Statická priorita sa potom použije pri výpočte dynamickej priority, ktorá riadi plánovanie: kým vlákno beží, jeho dynamická priorita klesá a kým je pozastavené, jeho dynamická priorita rastie.

Spravodlivosť – Fairness

272

- rovnaký (alebo založený na priorite) diel pre každé vlákno
- čo ak má jeden proces výrazne viac vlákien?
- čo ak má jeden užívateľ výrazne viac procesov?
- čo ak má skupina užívateľov výrazne viac aktívnych užívateľov?

V systéme, ktorý sme si naznačili vyššie sa plánovanie zaoberá iba individuálnymi vláknami. Takýto systém je zjavne nespravodlivý: procesy a užívatelia, ktorí majú veľa vlákien dostanú výrazne väčší podiel CPU než procesy a užívatelia s menej vláknami. Na zmiernenie tohto efektu implementuje väčšina operačných systémov určitú formu spravodlivého plánovania.

Plánovanie so spravodlivým podielom – Fair Share Scheduling

273

- môžeme použiť niekoľko-úrovňovú schému plánovania
- CPU je najprv rozdelené spravodlivo medzi užívateľskými skupinami
- potom medzi užívateľmi
- potom medzi procesmi
- a nakoniec medzi vláknami

Existujú rôzne úrovne, na ktorých môže spravodlivý plánovač pracovať: takmer všetky systémy berú do úvahy procesy, zatiaľ čo veľa ich berie do úvahy aj užívateľov. Pri tejto schéme sa plánovač snaží dosiahnuť nasledujúce dva ciele:

1. pokiaľ má daná entita (process, užívateľ, skupina) dostatok vlákien, ktoré môžu bežať, tieto vlákna dostanú rovnaký čas v porovnaní s akoukoľvek inou entitou s rovnakou vlastnosťou (t.j. ak oba procesy A aj proces B majú vlákna, ktoré môžu bežať, celkový časový diel pre proces A je rovný tomu pre proces B, ak predpokladáme, že majú rovnaké priority),
2. zdroje sú alokované efektívne: ak entita nemá dostatok vlákien, ktoré môžu bežať, zdroje, ktoré nemôže využiť sú prerozdelené medzi zvyšnými entitami rovnakého typu (t.j. ak proces A má 1 pripravené vlákno, proces B má 3 pripravené vlákna a sú k dispozícii 4 jadrá procesora, proces B by nemal byť obmedzovaný tým, že proces A môže využívať iba jedno z nich – systém by mal v tomto prípade využiť všetky 4 jadrá.

Plánovacie stratégie

274

- first in, first served (dávkové – batch systémy)
- najprv tie s najskorším deadline (realtime)
- round robin
- preemptívne s fixnou prioritou
- plánovanie so spravodlivým rozdelením (viac-užívateľské)

Pri počítaní dynamického plánu existuje množstvo prístupov. Najjednoduchší je dávkový plánovač, čo v podstate nie je vôbec plánovač: keď sa spustí program, beží kým neskončí; potom sa spustí ďalší program kým neskončí, a tak ďalej (first in, first served – prvý prišiel, prvý bude obslúžený).
U real-time systémov je pomerne jednoduchý plánovač známy ako 'earliest deadline first' – najprv tie s najskorším termínom vykonania: u týchto systémov má každá úloha definovaný deadline, ktorý systému hovorí, dokedy musí byť úloha najneskôr vykonaná. V preemptívnom nastavení je optimálnou stratégiou najprv vykonať úlohu s najskorším deadline.
Naivný, všeobecný (general-purpose) preemptívny plánovač je známy ako round robin: spúšťa všetky dostupné vlákna vo fixnom poradí, prepínajúc z jedného na druhé, keď im vyprší ich časový diel. V tomto systéme môže byť prioritizovaná priepustnosť (throughput) na každé vlákno (zmenou relatívnej veľkosti časových dielov rôznych vlákien), ale nie latencia.
Preemptívny plánovač s prioritami rieši problém s latenciou: ak sa zobudí vlákno s vysokou prioritou, bude môcť bežať veľmi skoro (s nízkou latenciou), keďže môže 'preskočiť frontu', na rozdiel od plánovača round robin.

Interaktivita

275

- priepustnosť (throughput) vs latencia
- latencia je dôležitejšia pre interaktívnu pracovnú záťaž
  - ako systémy telefónov alebo desktopy
  - ale tiež webservery
- priepustnosť (throughput) je dôležitejšia pre dávkové (batch) systémy
  - ako farmy na vykresľovanie (render), výpočetné gridy, simulácie

Plánovač často musí robiť kompromis medzi maximalizovaním celkovej výpočetnej priepustnosti – throughput a minimalizovaním latencie. U interaktívnych scenárov je primárnym problémom latencia, zatiaľ čo vo výpočetných kontextoch je prioritou obvykle celková priepustnosť.

Znižovanie latencie

276

- kratšie časové diely
- viac ochoty prepínať úlohy (viac preempcie)
- dynamické priority
- zvýšenie priority pre procesy na popredí – foreground

Pri optimalizácii na nízku latenciu musí plánovač používať krátke časové diely (aby prebúdzajúce sa vlákna nemuseli príliš dlho čakať na uvoľnenie procesora, aj keď nejde o vlákna s vysokou prioritou). Podobne by plánovač mal byť ochotný prepínať na vlákno s vyššou prioritou vždy keď sa nejaké prebudí, a preskočiť bežiace vlákna (aj keď ešte nespotrebovali svoj časový diel).
Obe tieto rozhodnutia samozrejme znamenajú viac prepínaní kontextu, ktoré sú drahé a teda negatívne ovplyvňujú celkovú priepustnosť (v akomkoľvek bode je vykonaného menej celkového užitočného výpočtu, keďže je viac času stráveného réžiou – či už v plánovači, ale najmä prepínaním vlákien a procesov).

Maximalizovanie priepustnosti

277

- dlhšie časové diely
- zredukovanie prepínania kontextu na minimum
- kooperatívny multitasking

Opak je pravdou pre systémy s vysokou priepustnosťou: časové diely by mali byť čo najdlhšie – také dlhé, aby to bolo ešte praktické, na minimalizovanie počtu prepínaní kontextu, ktoré CPU musí robiť. Podobne, plánovač by nemal byť ochotný prepínať úlohy, kým im nevyprší aktuálny časový diel, odsúvajúc vlákna, ktoré sa prebudili v reakcii na udalosť. Vždy, kde je to možné, by sa malo vyhýbať preempcii v prospech kooperácie (v dnešných časoch to však je problém na úrovni aplikácie).

Viac-jadrové plánovače

278

- tradične jedno CPU, veľa vlákien
- dnes: veľa vlákien, veľa CPU (jadier)
- komplikovanejšie algoritmy
- dátové štruktúry umožňujúce bezpečný súbežný prístup

V tradičnom návrhu by operačný systém očakával, že má k dispozícii iba jeden procesor, v takom prípade by plánovač bol pomerne jednoduchý. V podstate sú však všetky moderné počítače viac-jadrové systémy, a plánovače toto musia vziať do úvahy dvomi hlavnými spôsobmi:

1. algoritmus, ktorý rozhoduje, ktoré vlákna sú kedy spúšťané a na ktorom jadre je výrazne komplikovanejší,
2. plánovač musí byť schopný súbehu so sebou samým: niekoľko jadier musí byť schopných rozhodnúť, ktoré vlákno spustiť ďalšie paralelne, to znamená, že dátové štruktúry používané plánovačom na sledovanie vlákien musia umožňovať bezpečný súbežný prístup

Plánovanie a keše

279

- vlákna sa môžu presúvať medzi CPU jadrami
  - dôležité ak iné jadro je nevyužité
  - a pripravené vlákno čaká na CPU
- ale niečo to stojí
  - dáta vlákna / procesu sú do veľkej miery kešované
  - keše typicky nie sú zdieľané všetkými jadrami

Je celkom možné pozastaviť vlákno na jednom jadre a prebudiť ho neskôr na inom jadre. Keďže jadrá procesoru sú identické, nerobí to žiaden sémantický rozdiel – vlákno nebude schopné zistiť, že bolo presunuté. Ak sú však nejaké dáta asociované s týmto vláknom stále v časti keše, ktorá patrí jadru (typicky aspoň L1, ale tiež L2 a L3 ak sa jedná o inú päticu (socket)), spôsobí to prepad vo výkonnosti, keďže nové jadro musí znovu načítať dáta, ktoré už boli kešované druhým jadrom.

Afinita jadra

280

- moderné plánovače sa snažia vyhnúť presúvaniu vlákien
- hovorí sa, že vlákna majú afinitu k jadru
- extrémnym prípadom je pripnutie – pinning
  - toto úplne bráni vláknu, aby bolo premiestnené
- prakticky táto praktika zlepšuje priepustnosť
  - aj keď je nominálne využitie jadra nižšie

Z vyššie uvedeného dôvodu sa plánovače snažia vyhnúť presúvaniu vlákien medzi jadrami. Samozrejme existujú protichodné problémy: ak je vlákno pripravené a jadro iné než jeho preferované je nevyužité, dochádza k neefektívnosti: systém udržiava jadro procesora nevyužité, zatiaľ čo je vlákno pripravené na vykonávanie. Realistický plánovač potrebuje dosiahnuť rovnováhu medzi týmito dvomi problémami: ak sa má preferované jadro za chvíľu uvoľniť, počká, inak migruje vlákno. Samozrejme, toto sa niekedy nepodarí: môže sa prebudiť vlákno s vyššou prioritou a afinitou na rovnaké jadro a pôvodné vlákno bude musieť byť aj tak migrované. Preto silnejšia afinita spôsobí pokles vo využití jadier, ale v dobre vyladenom plánovači by to nemalo spôsobiť pokles v priepustnosti.

NUMA Systémy

281

- non-uniform memory architecture – neuniformná pamäťová architektúra
  - rôzna pamäť je pripojená k rôznym CPU
  - každý SMP blok sa v rámci NUMA nazýva node – uzol
- migrovanie procesu na iný uzol je drahé
  - ping-pong typu vlákno vs uzol môže zabiť výkon
  - vlákna jedného procesu by mali žiť na jednom uzle

V tradičnom SMP (symmetric multiprocessing) systéme má každé CPU rovný prístup k RAM. U veľkých počítačov s veľa CPU je výhodné (z pohľadu návrhu hardvéru) porušiť tento princíp a umožniť, aby bola RAM pripojená lokálne k jednej procesorovej pätici (socketu). Jadrá z iného takéhoto uzla potom budú musieť pristupovať k tejto pamäti nepriamo, čím pridajú penále čo sa týka priepustnosti (koľko bajtov na jednotku času môže byť prenesených z pamäte do CPU keše) aj čo sa týka latencie (koľko cyklov zaberie od momentu kedy sú dáta vyžiadané do momentu kedy sú k dispozícii procesoru).
Plánovač toto musí zobrať do úvahy: afinita vlákien (a celých procesov) ku konkrétnemu NUMA uzlu je zvyčajne výrazne silnejšia, než afinita vlákien k jednotlivým jadrám v rámci SMP systému: na rozdiel od réžie, ktorá súvisí čisto s kešami, pokles výkonu u spúšťania na inom NUMA uzle je permanentný: dáta nie sú automaticky migrované na 'bližší' blok pamäte.

Časť 5.3: Prerušenia a hodiny

V tejto sekcii sa pozrieme na to, ako je preemptívne plánovanie reálne implementované, cez periodické hardvérové prerušenia.

Prerušenie

283

- spôsob, akým sa hardvér dožaduje pozornosti
- CPU mechanizmus na odklonenie výpočtu
- čiastočné (iba CPU stav) prepnutie kontextu
- prepnutie do privilegovaného (kernel) CPU režimu

Prerušenia umožňujú perifériám dožadovať sa pozornosti od operačného systému. Ide o nízkoúrovňovú funkciu CPU, kde signál na príslušnom vývode (pine) procesora spôsobí, že procesor prestane vykonávať inštrukcie a spustí obsluhu prerušenia.
Keď príde prerušenie, CPU konzultuje tabuľku obslužných podprogramov (handlerov) prerušení, ktorá je nastavená operačným systémom. Ďalej CPU uloží svoj stav (hodnoty uložené v registroch) do pamäte (obvykle na aktívnom zásobníku).
Zvyčajne je procesorom realizované iba čiastočné prepnutie kontextu (t.j. stránkovacia tabuľka je nedotknutá), ale rutina prerušenia sa spustí v privilegovanom režime (takže môže, ak potrebuje, meniť stránkovacie tabuľky). V tradičných návrhoch je pamäť kernelu mapovaná (ale neprístupná z užívateľského režimu) do všetkých procesov, preto môže obsluha prerušenia (handler) priamo a okamžite čítať a zapisovať do pamäte kernelu.
Keď rutina prerušenia skončí, použije špeciálnu inštrukciu (`iret` na x86), ktorá inštruuje CPU, aby obnovilo svoj stav zo zásobníka (ktorý uložilo pred vstupom do obsluhy prerušenia) a vrátilo sa do užívateľského (neprivilegovaného) režimu.

Hardvérové prerušenia

284

- asynchrónne, na rozdiel od softvérových prerušení
- vyvolané cez signál na zbernici
- IRQ = interrupt request – žiadosť o prerušenie
  - iba iný názov pre hardvérové prerušenia
- PIC = programmable interrupt controller

Už sme sa skôr bavili o softvérových prerušeniach: obsluhy prerušení sú dosť podobné v oboch prípadoch (t.j. pri hardvérových aj softvérových prerušeniach), ale sú prítomné iné rozdiely. Konkrétne, hardvérové prerušenie je asynchrónne: CPU zavolá obsluhu nezávisle od toho, aká inštrukcia je práve vykonávaná. Hardvérovo sú prerušenia realizované cez signály na zbernici – periféria (alebo skôr radič prerušenia) pošle signál dolu vodičom do CPU, ktoré reaguje začatím svojej sekvencie na obsluhu prerušení.

Kontroléry prerušení

285

- PIC: jednoduchý obvod, typicky 8 vstupných vodičov
  - periférie sú pripojené k PIC vodičmi
  - PIC doručuje prioritizované signály CPU
- APIC: advanced programmable interrupt controller
  - rozšírený programovateľný radič prerušení
  - rozdelený na zdieľané IO APIC a lokálne APIC na úrovni jadier
  - typicky 24 vstupných IRQ „vodičov“
- OpenPIC, MPIC: podobné APIC, používané napr. Freescale

Radič prerušení je v podstate rozbočovač (hub), ktorý poskytuje množstvo liniek prerušení periférium, a signalizuje procesoru, použitím jediného vodiča, ak nejaká periféria vyvolá prerušenie. Radič prerušení samozrejme informuje CPU ktorá periféria je pôvodcom tohto prerušenia.

Meranie času (timekeeping)

286

- PIT: programmable interval timer
  - programovateľný intervalový časovač
  - kryštáľový oscilátor + delič (divider)
  - IRQ vodič do CPU
- lokálny APIC časovač: vstavané hodiny na úrovni jadra
- HPET: high-precision event timer – presný časovač udalostí
- RTC: real-time clock – hodiny reálneho času

(Programovateľný) časovač je ďalšou tradične samostatným komponentom, ktorý bol dávno integrovaný do CPU. Jeho úloha je udržiavať čas (historicky použitím kryštáľu kremeňa) a vyvolávať prerušenie v pravidelných, nastaviteľných intervaloch.

Prerušenie časovača

287

- generované z PIT alebo lokálneho APIC
- OS môže nastaviť frekvenciu
- hardvérové prerušenie sa deje pri každom tiku
- vytvára to príležitosť pre udržiavanie meta informácií (bookkeeping)
- a pre preemptívne plánovanie

Vždy, keď nakonfigurovaný interval uplynie, časovač vygeneruje prerušenie. Keď to nastane, kernel operačného systému má možnosť bežať, bez ohľadu na aktuálne vykonávaný proces. Okrem iného je možné vykonať prepnutie kontextu cez `iret` (alebo jeho ekvivalent) do iného vlákna alebo procesu než bol prerušený. Týmto spôsobom je obvykle implementované preemptívne plánovanie.

Prerušenie časovača a plánovanie

288

- odmeraj koľko času zabralo aktuálne vlákno
- ak mu došiel jeho časový diel, preruš ho (preempt)
  - vyber nové vlákno, ktoré má bežať
  - vykonaj prepnutie kontextu
- kontroly sa robia na každom tiku
  - preplánovanie je zvyčajne menej časté

(Plánovacia časť) obsluhy prerušení pre prerušenia časovača rýchlo skontroluje, či sa niečo má vykonať: t.j., pozrie sa koľko času zostáva aktuálne bežiacemu vláknu z jeho časového dielu. Ak mu čas došiel, potom musí nastať preplánovanie: obsluha sa vráti do iného vlákna (t.j. vykoná prepnutie kontextu... ak je to potrebné, zahŕňa to výmenu aktívnej stránkovacej tabuľky). Kontrola časového dielu sa robí pri každom tiku, keďže je lacná. U väčšiny tikov však nenastane žiadne preplánovanie, keďže typický časový diel má veľa tikov (a je výrazne drahšie).

Frekvencia prerušení časovača

289

- typicky 100 Hz
- to znamená 10 ms plánovacia jednotka (kvantum)
- 1 kHz je tiež možný
  - škodí priespustnosti ale zlepšuje latenciu

Tradičná frekvencia časovača na UNIX-ových systémoch je 100 Hz, alebo jeden tik každých 10 milisekúnd. To znamená, že časové diely dostupné plánovaču sú násobky 10 milisekúnd. U interaktívnych systémov môže byť táto frekvencia zvýšená, pričom najvyššie bežne používané nastavenie je 1 kHz (čím dostaneme kvantum dlhé iba 1 ms, a značne znížime latenciu plánovania). Samozrejme, CPU strávi výrazne viac času v obsluhe prerušenia časovača, čím zníži množstvo užitočnej práce, ktorú môže vykonať, a zhorší priepustnosť.

Beztaktové jadrá (tickless kernels)

290

- prerušenie časovača prebudí CPU
- môže to byť neefektívne ak je systém nečinný (idle)
- alternatíva: použiť jednorázové časovače
  - umožňuje CPU spať dlhšie
  - zlepšuje to energetickú efektivitu pri ľahkej záťaži

Moderné procesory implementujú agresívne riadenie výkonu, vypínajúc časti CPU, ktoré sú nevyužívané. Táto technika je samozrejme efektívnejšia, ak nevyužité obvody nemusia byť často budené. Ak sa vôbec nič nedeje, potom môže byť celé jadro uspaté. Nanešťastie, mierne do toho zasahuje prerušenie časovača: kým je systém úplne nečinný (idle; žiadne vlákna nie sú pripravené bežať), procesor bude stále musieť spracovať toto prerušenie, 100 krát za sekundu. Toto stojí výrazné množstvo energie, na zistenie, že nič nie je potrebné urobiť.
Namiesto použitia periodického časovača je možné nakonfigurovať jednorázový, neopakovaný časovač, a keď skončí („zapípa“), vypočítať kedy bude potrebné ďalšie prerušenie plánovania. Touto cestou môže v dobe nečinnosti procesor nerušene spať na výrazne dlhšie časové úseky.

Beztaktové plánovanie

291

- dĺžka plánovacej jednotky sa stáva súčasťou plánovania
- ak je jadro nečinné, prebudiť na ďalšom softvérovom časovači
  - synchronizácia softvérových časovačov
- ďalšie prerušenia sú normálne doručené
  - aktivita siete alebo disku
  - klávesnica, myš, ...

Takéto plánovanie je realizované zahodením fixnej plánovacej jednotky (kvanta). Plánovač môže vypočítať, kedy má nastať ďalšie preplánovanie, a nestrácať žiaden čas v prerušení časovača, kým nie je časový diel skončený. Spiace vlákna môžu byť zablokované z rôznych dôvodov, ale najčastejším je čakanie na udalosť alebo zdroj – jednou z udalostí, na ktoré sa normálne čaká je časovač: vlákno vykonáva nejakú periodickú činnosť a môže požiadať kernel, aby ho prebudil za sekundu, potom vykoná nejakú svoju prácu a spí ďalšiu sekundu, a tak ďalej.
Ak v nejakom danom momente všetky procesy spia, potom ďalšie prerušenie časovača musí nastať keď má vstať najbližšie vlákno, ktoré čaká na časovač. Samozrejme, ostatné prerušenia sú vyvolané ako obvykle, takže ak je vlákno blokované, lebo čaká na dáta, ktoré majú prísť z disku alebo zo siete, príslušné prerušenia môžu spôsobiť, že vlákno bude prebudené a naplánované na vykonávanie.

Ďalšie prerušenia

292

- sériový port
  - dáta sú dostupné na porte
- sieťový hardvér
  - dáta sú dostupné vo fronte paketov
- klávesnice, myši
  - užívateľ stlačí klávesu, pohne myšou
- USB zariadenia všeobecne

Okrem prerušenia časovačom existuje veľa ďalších prerušení, ktoré môžu nastať, a veľa z nich nepriamo ovplyvňuje plánovanie (väčšinou lebo jedno z vlákien čakalo na udalosť, ktorú prerušenie signalizovalo).

Routovanie prerušení

293

- nie všetky CPU jadrá potrebujú vidieť všetky prerušenia
- APIC môže byť oznámené ako má doručovať IRQs
  - OS môže routovať IRQs do CPU jadier
- viac-jadrové systémy: IRQ vyváženie záťaže
  - užitočné na prerozdelenie IRQ réžie
  - obzvlášť užitočné pri vysoko-rýchlostných sieťach

Konečne, poďme sa pozrieť na viac-jadrové systémy (pravdepodobne väčšina súčasných počítačov) a ako toto ovplyvňuje prerušenia: očividne, ak príde paket zo siete, nedáva zmysel prerušiť všetky jadrá: potrebujeme iba jedno jadro aby spustilo obsluhu (handler), ktorá vyzdvihne dáta alebo iným spôsobom vyrieši túto udalosť.
Navyše to nemusí byť stále to isté jadro vždy keď nastane prerušenie: ak prichádza veľa prerušení, chceme aby boli rozmiestnené trochu rovnomerne cez všetky jadrá. To je to čo robí IRQ load balancing – vyváženie záťaže – presmeruje prerušenia do jadier tak, aby každé jadro malo približne rovnako práce s ich obsluhou.
Nakoniec, keďže je plánovanie vykonávané na úrovni jadier, je užitočné mať oddelený časovač prerušenia pre každé jadro: tieto prerušenia samozrejme nepodliehajú vyváženiu záťaže alebo iným presmerovaniam: prerušenie je generované lokálne na každom jadre (v lokálnom APIC) a časovač na úrovni jadra, ktorý generuje prerušenie, je programovaný osobitne.

Review Questions

294

17.What is a thread and a process?
18.What is a (thread, process) scheduler?
19.What do fork and exec do?
20What is an interrupt?

Časť 6: Súbežnosť a zamykanie

Táto prednáška sa bude zaoberať problémami, ktoré môžu nastať pri behu niekoľkých vlákien a procesov zároveň, či už pri použití zdieľania času (time-sharing) jedného procesora alebo behu na niekoľkých fyzických CPU jadrách.

Obsah prednášky

296

1. Medzi-procesová komunikácia
2. Synchronizácia
3. Uviaznutia (Deadlocks)

V prvej časti sa budeme zaoberať základnými otázkami prečo a ako, čo sa týka medzi-procesovej a medzi-vláknovej komunikácie. Prirodzene z toho vyplynú otázky o zdieľaných zdrojoch a téma synchronizácie vlákien, vzájomné vylúčenie, atď. Nakoniec sa pozrieme na čakanie a uviaznutia, ktoré vznikajú, keď na seba môže čakať viacero vlákien.

Čo je to súbežnosť (concurrency)?

297

- udalosti, ktoré môžu nastať v rovnaký čas
- nezáleží na tom, či to nastane, iba že môže
- udalosti môžeme čiastočne usporiadať podľa kauzality (happens-before)
- sú súbežné – concurrent, ak nemajú kauzálny vzťah

Všeobecne, udalosť „sa stane pred“ (happens-before) inou udalosťou, keď sú kauzálne prepojené: prvá udalosť umožňuje, aby nastala druhá udalosť, alebo ju priamo spôsobuje. Rovnako často však udalosti nie sú kauzálne prepojené: môžu nastať v akomkoľvek poradí. O takýchto udalostiach hovoríme, že sú súbežné – concurrent: môžu nastať v ktoromkoľvek poradí, alebo môžu nastať, obrazne povedané, súčasne.

Prečo súbežnosť (concurrency)?

298

- dekompozícia problému
  - rôzne úlohy (tasks) môžu byť do veľkej miery nezávislé
- odráža externú súbežnosť
  - obsluhovanie viacerých klientov naraz
- limity výkonu a hardvéru
  - vyššia priepustnosť na viacjadrových počítačoch

Súbežnosť vzniká v softvérových systémoch z niekoľkých dôvodov: prvý, a možno najdôležitejší, je dekompozícia problému: je výrazne jednoduchšie navrhnúť veľký systém bez explicitného usporiadania všetkých možných udalostí. Komplexný systém vykoná veľké množstvo prirodzene nezávislých – súbežných (concurrent) – úloh (tasks). Určite by bolo možné vynútiť na takýchto úlohách umelé usporiadanie, ale tento prístup by nebol veľmi praktický.
Ďalším dôležitým dôvodom je, že súbežnosť prišla zvonku: ak existujú externé udalosti, ktoré sú súbežné, je ťažké predstaviť si, že odpovede systému budú prísne usporiadané: skutočné usporiadanie súbežných externých udalostí sa typicky nedá predvídať a vynútenie usporiadania reakcií by znamenalo, že minimálne niektoré reakcie by boli neprijateľne oneskorené.
Konečne, súbežnosť v softvéri umožňuje hardvéru podávať lepší výkon: súbežné inštrukcie možno vykonávať paralelne, bez obáv ohľadom ich relatívneho usporiadania – čo umožňuje hardvéru využívať svoje obmedzené zdroje efektívnejšie.

Paralelný hardvér

299

- hardvér je prirodzene paralelný
- softvér je prirodzene sekvenčný
- niečo musí ustúpiť
  - nápoveda: nebude to hardvér

Na rozdiel od softvéru, ktorý obvykle pozostáva zo sekvencie inštrukcií, ktoré sa vykonávajú v poradí, hardvér sa skladá z priestorovo usporiadaných obvodov. Väčšina z týchto sústav obvodov dokáže fungovať nezávisle od ostatných obvodov. Typické CPU napríklad bude obsahovať obvody pre násobenie čísel, a ďalšie obvody pre sčítanie čísel. Tieto obvody sú takmer úplne nezávislé a dokážu pracovať naraz, na rôznych dvojiciach čísel. Ak procesor sčíta čísla, mohol by súčasne násobiť iné čísla, bez negatívnych následkov na proces sčítania, ktorý prebieha v inej časti procesora.
Viac-jadrové procesory sú samozrejme extrémnym prípadom toho istého: oddelené jadrá sú (z veľkej časti) nezávislé kópie rovnakého (veľmi komplikovaného) obvodu.
Keďže sekvenčný výpočet už nevieme veľmi urýchliť, najlepšie v čo môžme dúfať, je využiť súbežnosť na vykonanie čo najviac možných vecí paralelne (na paralelnom hardvéri).

Časť 6.1: Medzi-procesová komunikácia

Komunikácia je dôležitou súčasťou softvéru, mimo najtriviálnejších prípadov. Hoci sú mechanizmy popísané v tejto sekcii tradične známe ako medzi-procesová (inter-process) komunikácia (IPC), uvážime tiež prípady, kedy tieto mechanizmy používajú vlákna jedného procesu (a nebudeme to označovať špeciálnym názvom, hoci ide v týchto prípadoch o intra-procesovú komunikáciu).

Pripomenutie: Čo je to vlákno

301

- vlákno je sekvencia inštrukcií
- každá inštrukcia „sa stane pred“ (happens-before) ďalšou
  - happens-before je na jednom vlákne lineárne usporiadanie
- základná jednotka plánovania

Než budeme pokračovať, potrebujeme si z minulej prednášky pripomenúť, že vlákno je sekvencia inštrukcií: každá inštrukcia sa teda „stane“ (happens before) ďalšou inštrukciou (pre viac matematicky založených, znamená to proste, že na inštrukciách jedného vlákna je happens-before lineárne usporiadanie). Ďalej sú vlákna základnou jednotkou plánovania: každé procesorové jadro vykonáva, v akomkoľvek momente, jedno vlákno.

Pripomenutie: Čo je to proces

302

- základná jednotka vlastníctva zdrojov
  - najmä pamäť, ale tiež otvorené súbory ap.
- môže obsahovať jedno alebo viac vlákien
- procesy sú od seba navzájom izolované
  - IPC vytvára medzery v tejto izolácii

Proces je potom jednotkou vlastníctva zdrojov – procesy vlastnia pamäť (virtuálny adresný priestor), otvorené súbory, sieťové spojenia, atď. Každý proces má aspoň jedno (ale môže mať viacero) vlákien, ktoré vykonávajú výpočet. Keďže je každý proces izolovaný vo svojom vlastnom virtuálnom adresnom priestore, neexistuje priama cesta, ako môžu vlákna rôznych procesov interagovať (komunikovať, synchronizovať sa). Nejaká úroveň komunikácie je však žiaduca, a nevyhnutná: operačný systém preto poskytuje niekoľko primitív, ktoré umožňujú procesom sa navzájom rozprávať, čím vytvárajú regulované medzery v tejto izolácii.

I/O vs komunikácia

303

- zoberte si štandardný vstup a výstup
  - predstavte si, že proces A zapisuje do súboru
  - neskôr, proces B číta tento súbor
- komunikácia sa deje v reálnom čase
  - medzi dvomi bežiacimi vláknami / procesmi
  - automaticky: bez zásahu užívateľa

Ďalší termín, ktorý si vymedzíme je komunikácia: konkrétne potrebujeme rozlišovať medzi ‘offline’ vstupom a výstupom. Technicky vzaté je čítanie súboru z disku komunikácia, pretože nejaký program do tohto súboru predtým zapisoval. Toto však nie je typ, ktorý nás zaujíma: zaujímajú nás iba inštancie, ktoré sa odohrávajú v reálnom čase. Formálnejšie, malo by byť prítomné striedanie akcií od dvoch vlákien, ktoré sú usporiadané v happens-before, t.j. aspoň jedna inštrukcia vlákna A je v happens-before usporiadaná medzi nejaké dve inštrukcie vlákna B.

Smer

304

- typická je obojsmerná (bidirectional) komunikácia
  - je to analogické konverzácii
- ale aj jednosmerná komunikácia dáva zmysel
  - napr. posielanie príkazov detskému procesu
  - počítajú sa potvrdenia za komunikáciu?

Prísne vzaté, použitím definície vyššie, je všetka komunikácia obojsmerná. Skutočne jednosmerná komunikácia je pomerne vzácna: aj sémanticky jednosmerná komunikácia často zahŕňa potvrdenie (acknowledgement) alebo inú formu odpovede, čím vyhovuje definícii vyššie.

Príklad komunikácie

305

- sieťové služby sú typickým príkladom
- vezmite si webový server a webový prehliadač
- prehliadač pošle požiadavku na web stránku
- server odpovie poslaním dát

Intuitívnym, hoci technicky komplikovaným, príkladom komunikácie je interakcia webového prehliadača s webovým serverom. V tomto prípade prehliadač použije sieťové spojenie na poslanie požiadavky, ktorá bude spracovaná serverom. Nakoniec server pošle odpoveď, na ktorú prehliadač čaká: je jednoduché vidieť kauzálne prepojenia: požiadavka sa stane pred (happens-before) odpoveďou, ktorá sa stane pred tým, než prehliadač zobrazí obsah užívateľovi.

Súbory

306

- je možné komunikovať cez súbory
- niekoľko procesov môže otvoriť rovnaký súbor
- jeden môže zapisovať dáta a ďalší ich môže spracovávať
  - pôvodný program si vyzdvihne výsledky
  - typicky keď sa používajú programy ako moduly

Niekoľko programov (procesov) môže otvoriť rovnaký súbor a čítať alebo zapisovať do neho dáta v rovnakom čase. S trochou opatrnosti

sa to dá robiť bezpečne. Nie je ťažké si predstaviť, že dva programy by mohli používať túto schopnosť na komunikáciu, ako je definovaná vyššie: jeden z programov zapisuje dáta na nejakom ofsete, druhý program ich číta a prípadne to potvrdí, na čo môže použiť iný mechanizmus.

IPC príklad založený na súboroch

307

- súbory sa používajú napr. keď spúšťate `cc file.c`
  - najprv sa zavolá preprocesor: `cpp -o file.i file.c`
  - potom prekladač: `cc1 -o file.o file.i`
  - a nakoniec linker: `ld file.o crt.o -lc`
- súbory z medzistupňov prekladu môžu byť skryté v `/tmp`
  - a zmazané, keď je úloha dokončená

Ako príklad, hoci možno trochu špeciálny, si pripomeňme ako riadiaci program prekladača (driver) komunikuje s jednotlivými fázami prekladu (proces kompilácie/prekladu sme si vysvetlili v druhej prednáške, pre prípad, že si ho potrebujete pripomenúť).
Každý krok vyzerá rovnako: driver zabezpečí, aby prechodná forma programu bola zapísaná do súboru, potom vytvorí podproces, ktorý prečíta tento súbor a zapíše jeho výstup do ďalšieho súboru. Očividne sa vyvolanie ďalšej fázy stane po tom, čo je výstup predchádzajúcej zapísaný.

Adresáre

308

- komunikácia cez umiestnenie súborov alebo linkov
- typické použitie: spool adresár
  - klienti nechávajú v adresári súbory na spracovanie
  - server odtiaľ periodicky vyzdviháva súbory
- používa sa napr. pri tlači a u emailov

Ďalší prístup komunikácie prostredníctvom súborového systému využíva adresáre: démon nastaví adresár, do ktorého môžu klienti vkladať súbory. Tieto súbory sú potom vyzdvihnuté a spracované démonom, a vymazané z adresára.
Tento prístup môžeme často vidieť pri spoolingu pre tlač súborov: klientský program vloží súbor (vo vhodnom formáte), ktorý chce nechať vytlačiť, do spool adresára, kde si ho démon, ktorý má na starosti tlač vyzdvihne a zariadi, aby bol súbor vytlačený a odstránený z fronty.
Emailové systémy používajú analogický proces, pri ktorom sú maily buď vložené do spool adresárov ako súčasť vnútorného spracovania (väčšina mailových serverov je poskladaných z niekoľkých služieb, ktoré sú prepojené IPC), alebo na vyzdvihnutie jednotlivými užívateľmi.

Rúry (Pipes)

309

- zariadenie na presúvanie bajtov v prúdoch
  - všimnite si rozdiel so správami
- jeden proces zapisuje, druhý číta
- čitateľ blokuje keď je rúra prázdna
- zapisovateľ blokuje keď je buffer rúry plný

Už sme si hovorili o rúrach: spomeňte si, že rúry presúvajú bajty z jedného procesu do druhého, použitím štandardného API súborového systému: dáta sa posielajú pomocou `write` (zápisu) a sú prijaté použitím `read` (čítania).
Dôležitá vec, na ktorú treba brať ohľad je, že každá rúra k sebe má pripojený (konečný) buffer: keď je buffer plný, proces, ktorý sa snaží `zapisovať` (write) dáta bude zablokovaný, kým druhý proces nezavolá `read`, čím odstráni nejaké dáta z bufferu. Podobne, keď je buffer prázdny, pokus o `čítanie` z rúry bude zablokovaný, kým druhá strana nevyvolá `write`, ktorý poskytne nejaké dáta, ktoré môže `read` vrátiť.

UNIX a rúry

310

- rúry sú v UNIX-e značne používané
- cez shellový operátor `|` sa dajú vybudovať pipelines – „potrubia“
- napr. `ls | grep hello.c`
- najužitočnejšie na postupné spracovanie dát v niekoľkých fázach

Hranica medzi IPC a “štandardným IO” je pomerne nejasná, hlavne v prípade užívateľsky-nastavených rúr (cez shellové pipelines). Programy v UNIX-e sú často napísané spôsobom, že čítajú vstup, spracujú ho a vypíšu výsledok ako svoj výstup. To ich robí vhodnými na použitie v pipelines. Hoci sú rúry pravdepodobne “automatickejšie”, než vloženie výstupu do súboru a spustenie ďalšieho príkazu, ktorý spracuje súbor, tiež je prítomný určitý stupeň manuálneho zásahu. Ďalší spôsob, akým môžeme na rozdiel nahliadať je, že rúra je najmä IPC mechanizmus, zatiaľ čo súbor je hlavne úložný mechanizmus.

Sokety

311

- podobné rúram, ale dá sa s nimi robiť viac
- umožňuje jednému serveru rozprávať sa s veľa klientami
- každé spojenie sa chová ako obojsmerná rúra
- môžu byť lokálni, ale tiež pripojení cez sieť

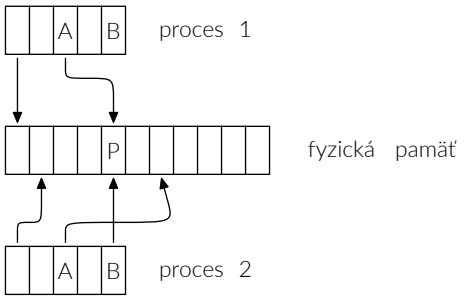
Možno si pamätáte, že sokety sú objekty podobné rúram, ale schopnejšie (a komplikovanejšie). Soket umožňuje jednému serveru otvoriť komunikačné kanály s viacerými klientami naraz (prípadne aj cez sieť), čím poskytuje obojsmerný kanál medzi každým klientom a serverom. Sokety navyše existujú v ‘datagramovej’ variante, kedy neustanovujú spojenia a vôbec sa nechovajú ako rúry: namiesto toho môžu byť v tomto režime použité na posielanie správ.

Zdieľaná pamäť

312

- pamäť je zdieľaná, keď k nej môže pristupovať viacero vlákien
  - deje sa to prirodzene u vlákien jedného procesu
  - hlavná forma medzi-vláknovej komunikácie
- veľa procesov môže mať namapovanú rovnakú fyzickú oblasť
  - toto je tradičnejšie použitie
  - tým tiež umožňuje medzi-procesovú komunikáciu

Zdieľaná pamäť je, v istom zmysle, najpriamejší možný spôsob komunikácie: prístup k pamäti je veľmi častou operáciou v programoch. Medzi vláknami rovnakého procesu je všetka pamäť ‘zdieľaná’, v zmysle, že všetky vlákna môžu čítať a zapisovať do celej pamäte (keďže zdieľajú rovnakú stránkovaciu tabuľku). Samozrejme je vo väčšine programov použitá na výmenu dát medzi vláknami iba malá, starostlivo ohraničená časť pamäte.
Procesy môžu ďalej nastaviť kus zdieľanej pamäte medzi sebou. Keďže však neexistuje zdieľaný virtuálny adresný priestor, každý proces môže mať zdieľaný blok pamäte namapovaný na inú virtuálnu adresu (nazvime príslušné virtuálne adresy A a B). Ide však o jediný blok fyzickej pamäte: zdieľaná pamäť sídli na fyzickej adrese P. Medzi-procesová zdieľaná pamäť je nastavená spôsobom, aby sa u prvého procesu A prekladalo na P a u druhého procesu B prekladalo na P.



Adresa B môže a nemusí byť validnou adresou v prvom procese, a to isté platí pre A v druhom procese.

Posielanie správ (Message Passing)

313

- komunikácia použitím diskrétnych správ
- môže a nemusí nám záležať na poradí doručenia
- môžeme sa rozhodnúť tolerovať stratu správ
- často sa používa po sieti
- môže byť implementované nad soketmi

Hoci je ťažké prekonať efektivitu zdieľanej pamäte ako komunikačného mechanizmu, posielanie správ má iné výhody: za prvé, je výrazne bezpečnejšie (môže byť pomerne zložité správne implementovať komunikáciu cez zdieľanú pamäť) a u niektorých prípadov použitia výrazne jednoduchšie na použitie. Posielanie správ môže navyše byť realizované cez sieť, čím je obzvlášť vhodné pre distribuované systémy.

Časť 6.2: Synchronizácia

V tejto sekcii sa budeme zaoberať špecifickou formou komunikácie, kde nedochádza k výmene žiadnych dát (dát aplikácie). Účelom synchronizačných primitív je zabezpečiť, aby boli akcie viacerých vlákien alebo procesov vykonané v správnom poradí (ktorých je zvyčajne obvykle veľa možných). Našou hlavnou motiváciou bude predísť takzvaným race conditions (súperenie vlákien), čo je v podstate nesprávne usporiadanie súbežných akcií – concurrent actions (pre hodiacu sa hodnotu slova 'nesprávny').

Zdieľané premenné (Shared Variables)

315

- štruktúrovaný pohľad na zdieľanú pamäť
- typické vo viac-vláknových programoch
- napr. každá globálna premenná v programe
- ale tiež môžu žiť v pamäti pochádzajúc z `malloc`

Než sa začneme baviť o samotnej synchronizácii, pozrieme sa na komunikačné zariadenie, ktoré budeme používať ako príklad: zdieľaná premenná je jednoducho pomenovaný kus zdieľanej pamäte, prístupný z viacerých vlákien pod rovnakým menom.

Zdieľaná premenná na halde

316

```
void *thread( int *x ) { *x = 7; }
int main()
{
    pthread_t id;
    int *x = malloc( sizeof( int ) );
    pthread_create( &id, NULL, thread, x );
}
```

Je tiež možné mať anonymné zdieľané premenné, alokované dynamicky na halde. Dá sa k nim pristupovať cez ukazateľ.

Súbeh (Race Condition): príklad

317

- uvážme zdieľaný čítač, `i`
- a nasledujúce dve vlákna

```
int i = 0;
void thread1() { i = i + 1; }
void thread2() { i = i - 1; }
```

Aká je hodnota `i` po tom, čo obe vlákna skončia?

Než sa pokúsime definovať race condition, pozrime sa na príklad. V programe vyššie dve vlákna menia rovnakú zdieľanú premennú: jedno pridáva jedničku, druhé odoberá. Skúste si premyslieť možné výstupy než budete pokračovať.

Súbeh (Race) na premennej

318

- prístup do pamäte nie je atomický
- vezmite si `i = i + 1` / `i = i - 1`

```
a₀ ← načítaj i  | b₀ ← načítaj i
a₁ ← a₀ + 1     | b₁ ← b₀ - 1
ulož a₁ i       | ulož b₁ i
```

Aby sme mohli pochopiť prečo nevinne vyzerajúci kód na predchádzajúcom slide zlyháva, musíme sa pozrieť na nízkoúrovňový popis programu (v podstate na úrovni strojových inštrukcií). Všimnite si, že každý z týchto dvoch výrazov sa prekladá na tri samostatné inštrukcie: načítaj hodnotu z pamäte, vykonaj príslušnú aritmetickú operáciu a ulož výsledok nazad do pamäte. Na tejto úrovni detailov by malo byť jednoduché vidieť, ako môže nastať, že inštrukcie týchto dvoch vlákien môžu byť usporiadané tak, aby dali nesprávny (alebo aspoň neočakávaný) výsledok, bez porušenia relácie happens-before (pamätajte, že inštrukcie každého vlákna zvlášť sú lineárne usporiadané).

Kritická sekcia

319

- ľubovoľná časť kódu, ktorá nesmie byť prerušená
- výraz `x = x + 1` by mohol byť kritická sekcia
- čo je kritická sekcia je závislé na doméne
  - ďalším príkladom môže byť banková transakcia
  - alebo vkladanie prvku do zreťazeného zoznamu

Ak chceme, aby výsledok predchádzajúceho programu bol vždy 0, musíme požadovať, aby každý z dvoch výrazov bol sekvenciou inštrukcií, ktoré patria do spoločnej kritickej sekcie: t.j. sekcia kódu, ktorá nesmie byť prerušená žiadnou inštrukciou, ktorá patrí do rovnakej kritickej sekcie. V tomto prípade to znamená, že keď sa začne vykonávať ktorákoľvek z inštrukcií `load`, vlákno, ktoré to urobilo sa najprv musí dostať až k odpovedajúcemu `store`, než môže druhé vlákno vykonať svoj `load`. Kritické sekcie nemenia reláciu happens-before: kritické sekcie, ako jednotky, sa môžu stať v ktoromkoľvek poradí (stále sú súbežné). Celá kritická sekcia sa však voči iným inštanciám rovnakej kritickej sekcie chová ako jedna atomická inštrukcia.

Race Condition: Definícia

320

- (anomálne) chovanie, ktoré závisí na časovaní
- typicky medzi viacerými vláknami alebo procesmi
- nastane neočakávaná sekvencia udalostí
- spomeňte si, že usporiadanie nie je zaručené

Teraz sa môžeme pokúsiť o definíciu: race condition (súbeh) je chovanie, ktoré

1. závisí na časovaní (v tom zmysle, že závisí od špecifického usporiadania súbežných udalostí),
2. nebolo očakávané programátorom.

Pri definícii vyššie môže byť race condition neškodná, a tento výraz je často používaný v tomto význame. V našom prípade je však užitočnejšie obsiahnuť aj tretiu podmienku:

3. chovanie je chybné.

Race conditions často číhajú na miestach kde je veľa súbežnosti (concurrency): viacvláknové programy sú hlavným príkladom (takmer všetky inštrukcie sú súbežné s veľkým množstvom ďalších inštrukcií, ktoré prichádzajú z ostatných vlákien).
Je však dôležité pamätať si, že nejaká forma komunikácie je nevyhnutná, aby nastala race condition – samotná súbežnosť nestačí. Je to preto, že čisto súbežné (completely concurrent) procesy sa nemôžu ovplyvňovať, a teda ich chovanie nemôže závisieť na konkrétnom usporiadaní udalostí, ktoré sú medzi nimi súbežné.

Súbeh (Race) v súborovom systéme

321

- súborový systém je tiež zdieľaným zdrojom
- a ako taký je náchylný na race conditions
- napr. dve vlákna sa pokúšajú vytvoriť rovnaký súbor
  - čo sa stane, keď sa to obom podarí?
  - ak obe zapíšu dáta, výsledok bude bordel

Keďže je súborový systém zdroj, ktorý je zdieľaný inak nesúvisiacimi procesmi, vytvára prostredie náchylné na race conditions: dochádza k rozsiahlej súbežnosti, ale tiež častej (niekedy náhodnej) komunikácii.

Vzájomné vylúčenie (Mutual Exclusion)

322

- kontext: iba jedno vlákno môže pristupovať k zdroju naraz
- zabezpečené cez tzv. mutual exclusion device – zariadenie pre vzájomné vylúčenie (a.k.a mutex)
- mutex má 2 operácie: `lock` (zamkni) a `unlock` (odomkni)
- `lock` možno bude musieť počkať, kým iné vlákno nespraví `unlock`

Keďže už na nejakej základnej úrovni rozumieme problémom, ktoré sú spojené so súbežnosťou (concurrency), poďme sa pozrieť na nejaké dostupné riešenia. Prvým, a v istom zmysle najjednoduchším, synchronizačným primitívom je mutual exclusion device – zariadenie pre vzájomné vylúčenie, ktoré je navrhnuté na vynucovanie kritických sekcií.
Často sa stáva, že je kritická sekcia asociovaná s nejakým zdrojom: t.j. všetok kód medzi každým získaním a príslušným uvoľnením zdroja je súčasťou jednej kritickej sekcie. V podstate to znamená, že jedným z krokov procesu získania zdroja je zamknutie mutexu a, analogicky, mutex je odomknutý ako súčasť procesu uvoľnenia, ktorý je asociovaný s daným zdrojom.

Semafory

323

- trochu všeobecnejší ako mutex
- umožňuje mať viacero zameniteľných inštancií zdroja
  - predstavte si N identických tlačiarní
  - potom môže v ľubovoľnom bode tlačiť N procesov naraz
- v podstate atomický čítač

Semafory sú, istým spôsobom, zovšeobecnenie mutexu – sú však užitočné iba pri práci so zdrojmi (t.j. môže byť použitý na stráženie všeobecnej kritickej sekcie iba v prípade, že je presne ekvivalentný mutexu). Rozdiel medzi mutexom a semaforom je ten, že do sekcie kódu stráženej semaforom môže vstúpiť viacero vlákien, ale toto číslo je obmedzené konštantou. Ak je konštanta nastavená na 1, potom je semafor to isté, čo mutex.

Monitory

324

- konštrukt programovacieho jazyka (nie je poskytovaný OS)
- interne využíva štandardné vzájomné vylúčenie
- dáta monitora sú prístupné iba jeho metódam
- do monitora môže vstúpiť iba jedno vlákno naraz

Monitor je prakticky plne zapuzdrený objekt (z definície objektovo orientovaného programovania), s jedným dodatočným obmedzením: každá inštancia monitora má so sebou asociovanú kritickú sekciu, a každá z jeho metód je plne súčasťou tejto kritickej sekcie.
Kým každá verejná metóda necháva objekt v konzistentnom stave (čo sa normálne vyžaduje od všetkých objektov), monitor je automaticky vláknovo bezpečný (thread-safe), t.j. viacero vlákien môže volať jeho metódy bez toho, aby riskovali race condition.

Podmienkové premenné (Condition Variables)

325

- čo ak monitor potrebuje na niečo čakať?
- predstavte si ohraničenú frontu implementovanú ako monitor
  - čo sa stane, keď sa zaplní?
  - zapisovateľ musí byť uspaný (suspended)
- podmienkové premenné majú operácie `wait` (čakaj) a `signal` (signalizuj)

Keďže sú kritické sekcie často asociované s komunikáciou, môže sa stať, že kód aktuálne bežiaci v kritickej sekcii nemôže pokračovať, kým nejaké ďalšie vlákno nevykoná konkrétnu akciu. Tento prípad použitia sa rieši ďalším synchronizačným konštruktom, známym ako podmienková premenná (condition variable): na premennú možno čakať, čo zablokuje volajúce vlákno, a môže signalizovať, čím zobudí čakajúce vlákno a umožní mu pokračovať.

Spinlock

326

- spinlock je najjednoduchšia forma mutexu
- metóda `lock` sa opakovane snaží získať zámok
  - to znamená, že spotrebúva procesorový čas
  - tiež známe ako busy waiting (aktívne čakanie)
- súperenie o spinlock na rovnakom CPU je veľmi zlé
  - ale môže byť veľmi efektívne medzi CPUs

Spinlock je obzvlášť jednoduchá implementácia abstraktného konceptu mutexu. Stav zariadenia uchováva jediný bit, operácia zamknutia pou-

žíva atomický compare and swap (porovnaj a vymeň) aby zmenila bit z 0 na 1 atomicky (t.j. spôsobom, aby operácia zlyhala v prípade, že nejakému inému vláknu sa ho podarilo zmeniť na 1 kým operácia bežala) v cykle, kým neuspeje. Operácia odomknutia jednoducho zresetuje bit na 0.
Má to niekoľko výhod: implementácia je veľmi jednoduchá, stav je veľmi kompaktný a latencia nemôže byť lepšia, za predpokladu, že súperenie je s iným CPU jadrom. Prípad, kedy vlákna nesúperia na jednom jadre je v podstate optimálny.
Má to však jednu vážnu (a v mnohých prípadoch fatálnu) nevýhodu: súperenie (stretnutie viacerých) na rovnakom CPU jadre má najhoršie možné chovanie (maximálna latencia a minimálna priepustnosť). Ďalším dôležitým problémom, hoci menej vážnym, je, že priepustnosť veľmi závisí na priemernej dĺžke chránenej kritickej sekcie: zámky, ktoré sú držané iba krátku dobu, fungujú dobre, ale dlhšie kritické sekcie budú výrazne mrhať zdrojmi, tým že držia CPU jadro v cykle aktívneho čakania.
Všeobecne sú spinlocky užitočné na ochranu krátkych kritických sekcií, u ktorých je zaručené, že súperiace vlákna budú bežať na rôznych CPU jadrách. Často sa to stáva v kerneli, ale iba výnimočne v užívateľských programoch.

Pozastavené (Suspending) mutexy

327

- tieto potrebujú kooperáciu od OS plánovača
- ak zlyhá získavanie zámku, vlákno spí
  - je vložené do čakacej fronty v plánovači (scheduler)
- odomknutie mutexu prebudí čakajúce vlákno
- vyžaduje systémové volanie → pomalé v porovnaní so spinlockom

Inou implementáciou mutexu, výrazne komplikovanejšou, ale tiež výrazne univerzálnejšou, sú pozastavené mutexy – suspending mutex. Hlavný rozdiel spočíva v prípadoch, kedy vlákna súperia na operácii lock: vždy, keď sa pokúsia zamknúť už zamknutý mutex, neúspešné vlákno je pozastavené (suspendované) a vložené do čakacej fronty plánovača. Operácia unlock (odomknutie) samozrejme potrebuje byť pozmenená, aby skontrolovala, či nejaké vlákna čakajú na mutex a ak áno, jedno zobudiť.
Interakcia s plánovačom znamená, že obe operácie – lock aj unlock musia urobiť systémové volanie, aspoň v niektorých prípadoch. V porovnaní s atomickými inštrukciami sú systémové volania výrazne drahšie, čím výrazne navyšujú réžiu mutexu.
Jedna z častých techník implementácie je kombinácia spinlocku a pozastaveného (suspending) mutexu: operácia zamykania urobí niekoľko málo cyklov, aby sa pokúsila získať mutex, a pozastaví vlákno, ak sa mu to nepodarí. Tento prístup kombinuje dobré vlastnosti oboch, ale je najkomplikovanejší na implementáciu, a tiež stav mutexu je aspoň taký veľký ako stav pozastaveného mutexu.

Podmienkové premenné

328

- rovnaký princíp ako suspending mutex
- čakajúce vlákno je vložené do čakacej fronty
- `signal` vráti vlákno späť do fronty bežiacich vlákien
- verzia s aktívnym čakaním sa nazýva polling

Typický spôsob implementácie podmienkovej premennej je cez interakciu s plánovačom, čo umožní čakajúcemu vláknu uvoľniť CPU jadro, ktoré zaberalo. Je možná alteratíva s aktívnym čakaním na spôsob spinlocku, ale nie je často používaná.

Bariéra

329

- niekedy paralelný výpočet funguje vo fázach
  - všetky vlákna musia dokončiť fázu 1
  - než ktorékoľvek môže začať fázu 2
- používa sa na to bariéra
  - zablokuje všetky vlákna, do bodu kým príde posledné
  - čakajúce vlákna sú obvykle pozastavené – suspendované

Ďalším synchronizačným mechanizmom, o ktorom si povieme, je bariéra. Zatiaľ čo mechanizmy, ktoré sme si doteraz opísali, sa dajú použiť v prípadoch s viac než dvomi vláknami, ich chovanie je vždy založené na interakciách po dvojiciach.
Bariéra je v tomto odlišná: synchronizuje niekoľko vlákien v jednom bode. Iba keď sa všetky zúčastnené vlákna zhromaždia na bariére, majú dovolené pokračovať.

Čitatelia a zapisovatelia

330

- predstavte si zdieľanú databázu
- veľa vlákien môže naraz čítať z databázy
- ale ak jedno zapisuje, nikto iný nemôže čítať ani zapisovať
- čo ak vždy chce niekto čítať?

Pozrime sa na iný synchronizačný problém, ktorý opäť zahŕňa viac než dve vlákna: máme nejaké dáta (databázu, ak chcete), do ktorých chce veľa vlákien nahliadať, a občas ich aktualizovať.
Naivným riešením by bolo obaliť všetok prístup k dátovej štruktúre do kritickej sekcie: je to nepochybne správne, ale nehospodárne, keďže viacero čitateľov navzájom neinteraguje, ale napriek tomu musia čakať vo fronte, keď chcú pristupovať k dátam.
Synchronizačný mechanizmus, ktorý rieši tento problém sa nazýva rwlock, alebo read-write lock (zámok pre čítanie a zápis), ktorý má 3 operácie: zamknúť na čítanie, zamknúť na zapisovanie a odomknúť. Invariantom je, že ak je rwlock zamknutý na zapisovanie, je zamknutý práve jedným vláknom. Inak môže existovať niekoľko vlákien, ktoré držia zámok na čítanie (čo samozrejme bráni tomu, aby bol zamknutý akýkoľvek zámok na zapisovanie, kým nie sú všetky zámky na čítanie uvoľnené).

Read-Copy-Update (Čítaj-skopíruj-aktualizuj)

331

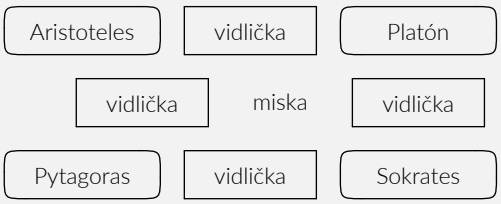
- najrýchlejší zámok je žiaden zámok
- RCU umožňuje čitateľom aby pracovali, kým prebiehajú aktualizácie
  - urob kópiu a aktualizuj kópiu
  - presmeruj nových čitateľov na aktualizovanú kópiu
- kedy je bezpečné uvoľniť pamäť?

V niektorých prípadoch možno scenár čitateľov a zapisovateľov vyriešiť bez zámkov (a dokonca nie je potrebná ani kritická sekcia). Funguje to tak, že zapisovateľ miesto modifikácie dát na mieste (čím by interferoval so súbežnými čítaniami) vytvorí kópiu dát (čo je operácia čítania, takže ju možno vykonať súbežne), aktualizuje túto novú kópiu dát, a potom inštruuje všetkých budúcich čitateľov, aby použili túto kópiu, väčšinou aktualizáciou nejakého ukazateľa.

Časť 6.3: Uviaznutie (deadlock) a vyhladovanie (starvation)

Obedujúci filozofovia

333



Problém 'obedujúcich filozofov' (dining philosophers problem) je definovaný nasledovne: uprostred stola je miska s jedlom a okolo stola je posadených niekoľko filozofov a medzi každými dvomi filozofmi je položená jedna vidlička. Každý filozof potrebuje dve vidličky naraz, aby mohol jesť. Filozofovia sedia a premýšľajú a keď vyhladovejú, zdvihnú vidličky a jedia.
Tento scenár sa používa na ilustráciu problému uviaznutia a vyhladovania u súbežných systémov. Dokážete vymyslieť inštrukcie pre filozofov tak, aby ste zabezpečili, že sa každý hladný filozof naje (v konečnom čase)? Premyslite si, prečo jednoduché algoritmy zlyhajú.

Zdieľané zdroje

334

- máme iba obmedzené množstvo inštancií hardvéru
- veľa zariadení dokáže robiť iba jednu vec naraz
- tlačiarne, DVD napaľovačky, páskové mechaniky, ...
- chceme používať tieto zariadenia efektívne → zdieľanie
- zdroje môžu byť získané a uvoľnené

Najprirodzenejšia situácia, v ktorej môžeme uvažovať o uviaznutí, je situácia kde viacero vlákien (a/alebo procesov) súperí o konečné zdroje. V tomto prípade je zdroj abstraktný objekt, ktorý možno získať, použiť a uvoľniť. Zdroj nemôže byť použitý pred tým, než je získaný a už nemôže byť použitý po tom, čo bol uvoľnený. O vlákne, ktoré získalo (a ešte neuvoľnilo) zdroj sa hovorí, že ho vlastní.

Zdieľanie po sieti

335

- zdieľanie nie je obmedzené na procesy na jednom počítači
- tlačiarne a skenery môžu byť pripojené na sieť
- celá sieť možno bude musieť koordinovať prístup
  - môže to viesť k uviaznutiam cez viacero počítačov

V praxi sa stáva, že sú zdroje 'vzdialené' – sprístupnené procesom na počítači A iným počítačom B, ktorý je pripojený do rovnakej siete. Nerobí to žiaden praktický rozdiel v našej úvahe (akurát je možno potrebné si uvedomiť, že uviaznutie môže siahať cez niekoľko počítačov).

Zámky ako zdroje

336

- zámkami sme sa zaoberali v predchádzajúcej sekcii
- zámky (mutexy) sú tiež formou zdroja
  - mutex možno získať (zamknúť) a uvoľniť
  - zamknutý mutex patrí konkrétnemu vláknu
- zámky sú proxy zdroje (ktoré zastupujú)

V nasledujúcej časti budeme o zámkoch (a dokonca aj o kritických sekciách) uvažovať jednoducho ako o forme zdroja: operácie na abstraktnom zdroji sa elegantne mapujú na operácie na (abstraktnom) zámku.

Odobrateľné (Preemptable) zdroje

337

- niekedy môžu byť držané zdroje odobrané
- to je prípad napr. fyzickej pamäte
  - proces môže byť odswapovaný na disk, ak je to potrebné
- odobrateľnosť (preemtability) môže tiež závisieť od kontextu
  - možno stránkovanie nie je k dispozícii

Niektoré zdroje môžu byť dočasne odobraté ich majiteľom, bez negatívnych vedľajších efektov, alebo za prijateľnú cenu (čo sa týka latencie, zhoršenia priepustnosti, alebo záležitostí súvisiacich s konkrétnymi zdrojmi, napr. plytvanie materiálom). Kanonickým príkladom je stránka pamäte: hoci je získaná nejakým procesom, systém ju môže odobrať bez kooperácie tohto procesu a bez vedomia procesu ju vráti keď sa používa (táto technika sa nazýva swapovanie – swapping).

Neodoberateľné (Non-preemptable) zdroje

338

- tieto zdroje sa nedajú (jednoducho) odobrať
- vezmite si foto tlačiareň uprostred tlače strany
- alebo DVD napaľovačku uprostred zápisu
- neodoberateľné zdroje môžu spôsobiť uviaznutie

Veľa zdrojov je prakticky neodoberateľných; to znamená, že ak ich raz vlákno alebo proces získa, nemôžu byť jednostranne odobrané bez značnej ujmy, napr. zabitím vlastniaceho procesu alebo nenávratným poškodením jeho výstupu zo zariadenia.

Získanie zdroja

339

- proces si musí vyžiadať prístup k zdroju
- nazýva sa to acquisition – získanie zdroja
- ak je požiadavke vyhovené, môže použiť zariadenie
- po skončení musí zariadenie uvoľniť
  - čím ho sprístupní ďalším procesom

Keď sme si už bližšie vysvetlili zdroje, poďme si v rýchlosti zopakovať protokol pre ich získanie a uvoľnenie. V nasledujúcom texte pracujeme s predpokladom, že iba obmedzený počet procesov môže vlastniť konkrétny zdroj v ľubovoľnom čase (a najčastejšie len jeden proces).

Čakanie

340

- čo robiť, keď chceme získať používaný zdroj?
- ak ho potrebujeme, musíme čakať
- je to rovnaké ako čakanie na mutex
- vlákno je presunuté do čakacej fronty

Budeme tiež predpokladať, že získanie je blokujúce: ak je zdroj používaný (vlastní ho iný proces), proces bude čakať, kým tento nie je uvoľnený, než bude pokračovať. Nemusí to vždy fungovať takto, ale je to asi najčastejší prístup.

Uviaznutie (Deadlock) zdrojov

341

- dva zdroje, A a B
- dve vlákna (procesy), P a Q
- P získa A, Q získa B
- P sa pokúsi získať B, ale musí čakať na Q
- Q sa pokúsi získať A, ale musí čakať na P

Konečne vieme dosť na to, aby sme sa mohli pozrieť na uviaznutia (zdrojov) – deadlocks. Pozrieme sa na najjednoduchší možný prípad, s dvomi vláknami a dvoma zdrojmi. Po sekvencii udalostí vyššie už nemôže žiadne z vlákien pokračovať: bez vonkajšieho zásahu budú P a Q zablokované navždy. Toto nazývame uviaznutie – deadlock. Všimnite si, prosím, že zvyčajne sú obe získania zdrojov v P súbežné s obomi získaniami zdrojov v Q.
Situácia vyššie môže samozrejme byť zovšeobecnená na ľubovoľné množstvo vlákien a zdrojov (ak máme z každého aspoň 2).

Podmienky uviaznutia zdrojov

342

1. vzájomné vylúčenie – mutual exclusion
2. podmienka drž a čakaj
3. neodoberateľnosť – non-preemtability
4. kruhové čakanie

Deadlock je možný iba ak sú prítomné všetky 4.

Musí sa zísť niekoľko rozmuných koncepčných rozhodnutí, aby mohli nastať podmienky pre uviaznutie. Je prirodzené, že by zdroj mal byť používaný v jednom bode iba jedným vláknom – je to nakoniec podstatou zdrojov.
Podmienka drž a čakaj nastáva, keď si jedno vlákno môže vyžiadať viacero zdrojov naraz a získava ich postupne. Tu je tiež zložité proti tomu niečo namietať, keďže to zodpovedá lineárnej povahe väčšiny programov – program (alebo skôr každé vlákno programu) je prakticky sekvencia inštrukcií, a každá inštrukcia sa vykoná až po tom, čo predchádzajúca skončila. Získavanie zdrojov nemôže byť dokončené, kým je zdroj používaný, a jediný spôsob, ako mať viac než jeden zdroj je získať ich postupne.
V niektorých prípadoch sa neprerušiteľnosť ani nedá ovplyvniť: je vlastnosťou daného zdroja. U kritických sekcií, ktoré neinteragujú so žiadnou externou entitou (zdroj, iné vlákna) s výnimkou svojho zámku je trochu priestoru na manévrovanie. V tomto prípade by sa možno dal urobiť roll back a vrátiť následky kritickej sekcie a pokúsiť sa ju reštartovať. Nie je to však úplne jednoduché.
Uvážme statický graf závislostí zdrojov, v ktorom uzly reprezentujú zdroje a hrany indikujú, že medzi nimi existuje podmienka drž-a-čakaj, tzn, hrana A → B je prítomná v grafe, ak existuje vlákno, ktoré sa snaží získať B, zatiaľ čo drží A. Podmienka kruhového čakania je splnená, ak v tomto grafe existuje cyklus.

Uviaznutia iné ako na zdrojoch

343

- nie všetky uviaznutia sú spôsobené súperením o zdroje
- predstavte si systém posielania správ
- proces A čaká na správu
- proces B pošle správu A a čaká na odpoveď
- správa sa stratí pri prenose

Čo je pomerne dôležité je, že štyri podmienky vyššie sa týkajú iba uviaznutí na zdrojoch: existujú ďalšie typy uviaznutí s inou sadou podmienok. Znamená to, že aj keď sa nám podarí eliminovať jednu zo 4 podmienok, a tým zabrániť uviaznutiam na zdrojoch, nebude to vo výsledku znamenať, že v našom systéme nemôžu nastať uviaznutia.

Príklad: uvizanutie na rúrach

344

- spomeňte si, že aj čitateľ aj zapisovateľ môže byť zablokovaný
- čo ak vytvoríme rúru v každom smere?
- proces A zapíše dáta a pokúsi sa prečítať odpoveď
  - je zablokovaný, lebo opačná rúra je prázdna
- proces B prečíta dáta, ale čaká na ďalšie → uviaznutie

Typickým príkladom iného než zdrojového uviaznutia je príklad s rúrami: spomeňte si, že prázdna rúra blokuje pri čítaní, zatiaľ čo plná rúra blokuje pri zapisovaní. Existuje niekoľko možností, ako sa to môže pokaziť, v prípade, že máme viac než jednu rúru. Pravdepodobne si všimnete určitú podobnosť medzi týmto typom uviaznutia a uviaznutiach na zdrojoch, ktoré sme si vysvetlili do hĺbky.

Uviaznutia: zaujíma nás to?

345

- uviaznutia sa môžu veľmi ťažko ladiť
- môžu byť tiež veľmi vzácne
- môžeme riziko uviaznutia považovať za prijateľné
- proste všetko rebootujeme keď dôjde k uviaznutiu
  - tiež známe ako pštrosí algoritmus

Veľa (pravdepodobne väčšina) uviaznutí pochádza zo súbehu vlákien (race condition), preto sa nedejú veľmi často. Na tejto vzácnosti je založený takzvaný pštrosí algoritmus (ostrich algorithm): ak dôjde k uviaznutiu, zabi všetky zapojené procesy, alebo proste reštartuj celý systém. Samozrejme môže byť zložité rozhodnúť, či uviaznutie nastalo.

Detekcia uviaznutí

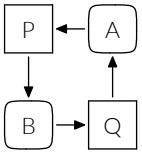
346

- môžeme sa pokúsiť aspoň detekovať uviaznutia
- zvyčajne skontrolovaním podmienky kruhového čakania
- udržiavame si graf vlastníctva vs čakania
- ak je v grafe cyklus → uviaznutie

Jedným zo spôsobov, ako sa dajú detekovať uviaznutia je použiť dynamickú verziu statickej podmienky kruhového čakania: v tomto prípade má graf závislostí dva typy uzlov: vlákna a zdroje. Hrany sú vždy medzi vláknom T a zdrojom R: R → T existuje, ak vlákno T momentálne vlastní zdroj R a T → R existuje, ak vlákno T čaká na získanie R. Ak v grafe existuje cyklus, v systéme je prítomné uviaznutie – deadlock (a vlákna, ktoré ležia na cykle sú všetky zablokované). Pripomeňme si príklad s dvomi vláknami, P a Q, a dvoma zdrojmi, A a B, ktorý sme si ukázali skôr:

1. P úspešne získa A (vznikne hrana A → P)
2. podobne, Q získa B, čo znamená B → Q
3. P sa pokúsi získať B, ale musí čakať, teda P → B
4. Q sa pokúsi získať A, ale musí čakať, teda Q → A

Toto je výsledný graf, ktorý v sebe má zrejmý cyklus:



Zotavenie sa z uviaznutia

347

- ak je zahrnutý odobrateľný zdroj, prideľ ho niekomu inému
- inak je prípadne možné urobiť rollback
  - je na to potrebný premyslený mechanizmus checkpointov
- ak všetko ostatné zlyhá, zabi niektoré procesy
  - zariadenia možno budú musieť byť reštartované

Uviaznutie, ktoré zahŕňa aspoň jeden odobrateľný zdroj môže síce nastať, ale na rozdiel od 'štandardného' prípadu, je možné sa z neho zotaviť a pokračovať vo výpočte bez násilného ukončenia niektorého z vlákien alebo procesov, ktoré sa na tom podieľajú. Namiesto toho je jeden zo zdrojov odobraný, čím prerušíme cyklus a umožníme systému pokračovať.
Ak uviaznutie namiesto toho zahŕňa reštartovateľnú kritickú sekciu (ako sme si uviedli skôr), potom táto kritická sekcia môže byť vrátená nazad (rolled back) a reštartovaná, čím opäť umožní vláknam pokračovať.
Konečne, ak všetko ostatné zlyhá, systém si môže zvoliť obetné vlákno na cykle a ukončiť ho, čím uvoľní zdroje, ktoré držalo. Rovnako ako v ostatných prípadoch, systém ako celok vďaka tomu môže pokračovať.

Vyhýbanie sa uviaznutiu

348

- môžeme prípadne odmietnuť získanie zdroja na predídenie uviaznutiu
- musíme poznať maximum zdrojov pre každý proces
- vyhýbanie sa spolieha na bezpečné stavy
  - najhorší prípad: všetky procesy si vyžiadajú maximum zdrojov
  - bezpečný znamená, že predídeme uviaznutiu v najhoršom prípade

Všeobecne je predchádzanie uviaznutiu prístup, kedy systém odmietne získanie zdroja, keď to môže neskôr viesť k uviaznutiu. Najznámejšia inštancia výhybania sa uviaznutiu je bankárov algoritmus E. Dijkstru. Tento algoritmus je aplikovateľný v prípadoch, kedy každý zo zdrojov má niekoľko zameniteľných inštancií, ktorých je viac, než si ktorékoľvek jedno vlákno môže naraz vyžiadať. Vstupom algoritmu je maximálne množstvo inštancií zdroja, ktoré môže byť vyžiadané každým vláknom. Invariant bezpečnosti je nasledujúci:

1. existuje vlákno T, také, že dostupné zdroje postačujú pre splnenie jeho maximálnej alokácie zdrojov,
2. keď vlákno T skončí (a vráti všetky svoje aktuálne alokované zdroje), tento invariant stále platí.

Všetky získavania zdrojov, ktoré by porušili tento invariant budú zamietnuté. Východzie podmienky implikujú, že na začiatku sme v bezpečnom stave; ukázali sme teda, že algoritmus je korektný. Algoritmus predpokladá, že okrem získavania zdrojov nemôže nič zablokovať žiadne z vlákien. Nasledujúca tabuľka ilustruje tento algoritmus (P, Q a R sú vlákna, čísla znamenajú počet držaných zdrojov / maximálne pridelenie):

krok	P	Q	R	dostupné	akcia
1	0/3	0/2	0/4	5/5	P získava 1
2	1/3	0/2	0/4	4/5	R získava 2
3	1/3	0/2	2/4	2/5	Q získava 1
4	1/3	1/2	2/4	1/5	P je odmietnuté 1
5	1/3	1/2	2/4	1/5	Q získava 1
6	1/3	2/2	2/4	0/5	Q končí
7	1/3	–	2/4	2/5	P získava 1
8	2/3	–	2/4	1/5	

Všimnite si, že v kroku 4 bola požiadavka P zamietnutá, pretože by už neexistovalo žiadne vlákno, ktoré by garantovane mohlo urobiť dostatočný pokrok vo výpočte na uvoľnenie zdrojov.

Predchádzanie uviaznutiu

349

- vyhýbanie sa uviaznutiu (deadlock avoidance) je typicky nepraktické
- pre existenciu uviaznutí sú potrebné 4 podmienky
- môžeme sa pokúsiť napadnúť tieto podmienky
- ak dokážeme jednu odstrániť, uviaznutiam predídeme (prevention)

Vyhýbanie sa uviaznutiam nanešťastie nie je veľmi praktické u všeobecných (general-purpose) operačných systémov: často sa stáva, že konkrétny zdroj má k dispozícii iba 1 inštanciu a že vyššie uvedený algoritmus by tým vynútil všetky programy, ktoré chcú použiť zdroj, aby bežali jeden po druhom. A predpoklad, že nedochádza k žiadnym ďalším interakciám medzi vláknami tiež nie je veľmi realistický.

Predchádzanie cez Spooling

350

- útočí na vlastnosť vzájomného vylúčenia
- niekoľko programov môže zapisovať do tlačiarne
- dáta sú zozbierané spooling démonom
- ktorý potom pošle úlohy do tlačiarne postupne

Tento prístup vymení uviaznutie na tlačiarni za uviaznutie na mieste na disku. U miesta na disku je v tomto prípade však výrazne pravdepodobnejšie, že je odoberateľné (preemptable), keďže úloha, ktorá je zablokovaná kvôli tomu, že sa zaplnil disk, môže byť zrušená (a vymazaná z disku) a neskôr opäť skúsená (na rozdiel od napoly-vytlačenej strany).

Predchádzanie rezervovaním

351

- tiež sa môžeme pokúsiť odstrániť drž-a-čakaj
- napríklad dovoliť iba dávkové získavanie zdrojov
  - proces si musí vyžiadať všetko naraz
  - zvyčajne je to nepraktické
- alternatíva: pusť a získaj znovu

Určite si dokážeme predstaviť, že by sme mohli vyžadovať, aby boli všetky získavania zdrojov robené dávkovo, teda vždy vyžadujúc, v jeden atomickej akcii, všetky zdroje potrebné v konkrétnej sekcii kódu. Zdroje môžu byť uvoľňované po jednom. Ak je potrebný dodatočný zdroj nad rámec tých, ktoré už sú držané, program najprv musí všetko uvoľniť, než si vyžiada ďalšiu dávku.

Predchádzanie usporiadaním

352

- tento prístup eliminuje kruhové čakanie
- zavedieme globálne usporiadanie na zdrojoch
- proces môže získavať zdroje výhradne v tomto poradí
  - musí ich uvoľniť + znovu získať ak je poradie nesprávne
- týmto spôsobom nie je možné vytvoriť cyklus

Konečne, asi najpraktickejší prístup v rámci jedného programu (alebo iného uzavretého systému, ale nie u operačných systémov) je vynútenie globálneho usporiadania na získavaní zdrojov, ktorým sa musí každé vlákno riadiť. Znamená to, že statický graf závislostí zdrojov je acyklický a uviaznutie nemôže nastať.

Livelock

353

- pri uviaznutí sa nedá urobiť žiaden krok vpred
- ale nie je oveľa lepšie keď procesy len chodia tam a nazad
  - napríklad uvoľňovanie a opätovné získavanie zdrojov
  - nerobia žiaden užitočný pokrok
  - navyše spotrebúvajú zdroje
- toto sa nazýva livelock a je rovnako zlý ako deadlock

Pri deadlocku sú zablokované vlákna zvyčajne pozastavené / suspendované (nemôžu bežať). Ak sú niektoré alebo všetky vlákna, ktoré sú zapojené do stavu podobnému uviaznutiu aktívne (či už aktívne čakajú, pollujú alebo sa inak pokúšajú o akciu, ktorá bude navždy zlyhávať), hovoríme o livelocku: vlákna vykonávajú inštrukcie, ale nedochádza k žiadnemu pokroku.

Vyhladovanie

354

- k vyhladovaniu dochádza, keď proces nemôže robiť žiaden pokrok
- zovšeobecnenie oboch prípadov – deadlock aj livelock
- napríklad nespravodlivé plánovanie vo vyťaženom systéme
- tiež si spomeňte na problém čitateľov a zapisovateľov

Nakoniec, najvšeobecnejšou myšlienkou je vyhladovanie (starvation); ide o stav, kedy vlákno nemôže robiť pokrok, z akéhokoľvek dôvodu: je zablokované na zdroji, nie je mu pridelený čas na procesore, atď. Príklad, ktorý nie je deadlock ani livelock, môže nastať u naivného riešenia problému čitateľov a zapisovateľov, ktorý sme videli skôr: ak máme dostatok čitateľov, takže je stále prítomný zámok na zdroji, zapisovatelia budú navždy blokovaní a teda vyhladovejú.

Review Questions

355

21. What is a mutex?
22. What is a deadlock?
23. What are the conditions for a deadlock to form?
24. What is a race condition?

Časť 7: Ovládače zariadení

Abstrahovanie hardvéru je jednou z hlavných úloh operačného systému. Hoci sme si už v predchádzajúcich prednáškach detailne vysvetlili základné hardvérové zdroje (CPU a pamäť), takzvané periférie tiež zohrávajú dôležitú úlohu. V tejto prednáške sa pozrieme na rozhranie medzi operačným systémom a periférnym hardvérom (sieťové karty, energeticky nezávislé / perzistentné úložiská, odpojiteľné úložné médiá, displeje, vstupné zariadenia, a tak ďalej).

Obsah prednášky

357

1. Ovládače, IO a prerušenia
2. Systémové a rozširujúce zbernice
3. Grafika
4. Energeticky nezávislé úložiská
5. Siete a bezdrôtová komunikácia

Časť 7.1: Ovládače, IO a prerušenia

V prvej časti si rozoberieme nízkoúrovňové aspekty interakcie hardvéru: ako sa dáta presúvajú medzi CPU a perifériou, ako periférie signalizujú udalosti procesoru, a ako toto všetko súvisí s operačným systémom (ktorý beží na procesore).

Vstup a výstup (IO)

359

- väčšinou budeme uvažovať o veciach z pohľadu IO
- periférie produkujú a prijímajú dáta
- vstup – čítanie dát vyprodukovaných zariadením
- výstup – posielanie dát na zariadenie
- protokol – validná sekvencia IO udalostí

Hoci periférie môžu byť pomerne komplikované, budeme o nich uvažovať relatívne abstraktným, zjednodušeným spôsobom, ako o zariadeniach, ktoré produkujú a spotrebúvajú (prijímajú) dáta. Ďalším dôležitým komponentom v našom pochopení zariadení budú udalosti. Validné sekvencie udalostí a vstupy a výstupy späté s týmito udalosťami sú popísané protokolom.

Dátové prenosy, ktoré sú späté s udalosťami (t.j. keď sa dejú v špecifickom časovom vzore), môžu reprezentovať pomerne široký záber chovaní a následkov. Vezmite si klávesnicu: keď užívateľ stlačí klávesu, ide o udalosť, a táto je sprevádzaná prenosom dát, ktorý hovorí systému, ktorá klávesa bola stlačená (alebo jej stlačenie bolo uvoľnené). Podobne, keď dôjde k pohnutiu myšou, prúd dát, ktorý popisuje relatívny pohyb je poslaný počítaču.

Ďalšie typy zariadení naopak prijímajú dáta: vezmite si displej ako prototyp tohto typu zariadenia: počítač (viac-menej nepretržite) posiela dáta, ktoré reprezentujú pixely, ktoré sa majú zobraziť na displeji a ktoré sa následne zobrazia užívateľovi.

Iné zariadenia prijímajú príkazy (ktoré sú samozrejme tiež formou dát) a tiež odpovedajú dátami (odpovede na príkazy). Napríklad taký pevný disk: keď si systém praje uložiť nejaké dáta, pošle príkaz (spolu s dátami samotnými - tzv. payload, t.j. dáta, ktoré majú byť uložené) a dostane potvrdenie. Podobne, keď si chce nejaké dáta vydzvihnúť, pošle príkaz na čítanie a dostane odpoveď, ktorá obsahuje dáta, ktoré boli uložené na požadovanej adrese.

Čo je to ovládač?

360

- kus softvéru, ktorý sa rozpráva so zariadením
- väčšinou pomerne špecifický / neprenositeľný
  - spätý s konkrétnym zariadením
  - a tiež operačným systémom
- často súčasťou kernelu

Zjavne musia byť vstupné dáta spracované a výstupné vygenerované. Tiež je pomerne jasné, že formát a obsah dát bude špecifický pre kon-

krétne zariadenie. Preto musí byť softvér schopný vytvoriť a pochopiť dáta vo formáte, ktorému rozumie konkrétne zariadenie.
Softvér, ktorý je zodpovedný za túto komunikáciu je známy ako ovládač (driver), a ako vyplýva z vyššie uvedeného, je pomerne jasné, že každý ovládač je spárovaný s konkrétnym zariadením, alebo malou triedou zariadení. Alebo, aby sme boli presnejší, ovládač implementuje jednu stranu protokolu (druhá strana je implementovaná samotným zariadením).
Na prvý pohľad sa nezdá, že by existoval dobrý dôvod, prečo by ovládač mal byť spätý s konkrétnym operačným systémom: protokol používaný zariadením bude predsa rovnaký, bez ohľadu na to, aký operačný systém beží na CPU. Ale protokol na strane zariadenia je samozrejme iba jednou časťou ovládača: druhou časťou je komunikácia s operačným systémom. Táto komunikácia je realizovaná pomocou sady rozhraní, ktoré sú obvykle špecifické pre daný operačný systém (ale existujú aj prenositeľné ovládače).
Ďalší problém, ktorý viaže ovládače na konkrétny operačný systém je, že ovládače obvykle potrebujú navzájom spolupracovať: neskôr v prednáške uvidíme, že zariadenia sú prepojené cez ďalšie zariadenia, a že ovládač periférie musí komunikovať s ovládačom zbernice, aby mohol komunikovať s perifériou.

Ovládače v režime kernelu

361

- sú súčasťou kernelu
- bežia s plnými privilégiami kernelu
  - vrátane neobmedzeného prístupu k hardvéru
- žiadna alebo minimálna réžia, čo sa týka prepínania kontextu
  - rýchle ale nebezpečné

V istom zmysle je najjednoduchším typom ovládača taký, ktorý je súčasťou kernelu. Ovládač tohto typu môže priamo používať všetku funkcionalitu CPU a hardvéru, ktorú potrebuje na komunikáciu so svojim zariadením, bez toho, aby musel ísť cez nejakého prostredníka. Keďže do toho nie sú zapojené žiadne procesy, tiež to znamená, že kód ovládača môže bežať bez prepínania kontextu, ak je to potrebné (napr. u odpovede na hardvérové prerušenie).
Vďaka tomu sú ovládače v režime kernelu obzvlášť rýchle (s nízkou réžiou), ale ich neobmedzený prístup k hardvéru a pamäti spôsobuje, že všetky problémy v takýchto ovládačoch sú veľmi nebezpečné. Ak napríklad ovládač spadne, obvykle so sebou vezme celý operačný systém.

Mikrokernely

362

- ovládače sú vylúčené z mikrokernelov
- ale ovládač stále potrebuje prístup k hardvéru
  - môže sa jednať o špeciálnu oblasť pamäte
  - prípadne potrebuje reagovať na prerušenia
- v princípe sa dá všetko robiť nepriamo
  - ale môže to byť aj pomerne drahé

Zatiaľ čo sú ovládače v režime kernelu rozšírené v návrhoch monolitických kernelov, z mikrokernelov sú prakticky vykázané. Namiesto toho je každý ovládač samostatný proces a beží v užívateľskom režime CPU.
Veľa ovládačov však na komunikáciu so svojím zariadením vyžaduje určitú úroveň priameho prístupu k hardvéru: najčastejšie obsluhy prerušení a čítania/zápisy na konkrétnu oblasť fyzickej pamäte. Prístup k pamäti môže byť zabezpečený pomerne jednoducho (skrátka namapovať túto oblasť pamäte do procesu ovládača).
Prvá časť je však problematická: obsluhy prerušení (na ktoré sa bližšie pozrieme za chvíľu) vždy bežia v privilegovanom režime, a teda ovládač žiadnu nemôže nastaviť. Namiesto toho kernel sprostredkuje prerušenie procesu ovládača pomocou nejakej formy medzi-procesovej komunikácie (IPC), čo často spôsobí drahé prepnutie kontextu.

Ovládače užívateľského režimu

363

- veľa ovládačov môže bežať plne v užívateľskom priestore
- zlepšuje to robustnosť a bezpečnosť
  - chyby v ovládači nezhodia celý systém
  - a tiež nemôžu narušiť bezpečnosť systému
- bude to prípadne stáť určitú výkonnosť

Ovládače, ktoré bežia v užívateľskom priestore, nie sú výhradne u mikrokernelov a hoci majú isté nevýhody, tiež majú veľa žiaducich vlastností. Keďže sú izolované od kernelu, od seba navzájom a od ostatných programov bežiacich na systéme, keď ovládače spadnú alebo obsahujú chyby, nemôže to narušiť zvyšok systému (aspoň nie priamo, hoci keď je periféria chybne naprogramovaná, stále môže spôsobiť pád systému alebo ho inak učiniť nepoužiteľným). Samozrejme, tiež to výrazne zlepšuje bezpečnosť.

Ovládače v samostatných procesoch

364

- ovládače užívateľského režimu typicky bežia vo svojom vlastnom procese
- znamená to prepínania kontextu
  - vždy, keď si zariadenie vyžaduje pozornosť (prerušenie)
  - vždy, keď chce iný proces použiť zariadenie
- ovládač potrebuje systémové volania, aby mohol komunikovať so zariadením
  - spôsobuje to ešte viac réžie

Poďme sa pozrieť na model, kde každý ovládač beží vo svojom vlastnom procese. Ako sme už spomenuli, jeden veľký problém tohto modelu je spôsobený dodatočnými prepnutiami kontextu, ktoré sa dejú keď:

1. príde prerušenie od hardvéru a v tom čase sa vykonáva nejaký iný proces (čo je skoro vždy),
2. ďalší proces na systéme sa pokúša použiť zariadenie: požiadavka musí ísť cez ovládač, čo znamená, že musí bežať, a teda jeho proces musí byť naplánovaný pred tým, než môže byť požiadavka obslúžená.

Žiadna z týchto situácií nenastáva u ovládačov pracujúcich v režime kernelu. Konečne, na vykonanie akejkoľvek privilegovanej operácie musí ovládač vykonať systémové volanie – hoci je menej drahé ako prepnutie kontextu, je zároveň výrazne drahšie ako normálne volanie funkcie.

Vnútroprocesové ovládače

365

- čo ak môže byť ovládač (alebo jeho veľká časť) knižnica
- to najlepšie z oboch svetov
  - žiadna réžia z prepínania kontextu pre požiadavky
  - chyby a bezpečnostné problémy zostávajú izolované
- často používané u GPU-akcelerovanej 3D grafiky

Existuje alternatívny model, ktorý zmierňuje niektoré z nevýhod ovládačov, ktoré bežia v užívateľskom režime. Konkrétne, druhý zdroj prepínania kontextu môže byť (aspoň čiastočne) eliminovaný tým, že bude ovládač bežať v rovnakom procese ako aplikácia, ktorá používa zariadenie. Ako by to fungovalo?
Ovládač môže byť dostupný ako knižnica a aplikácia sa bude linko-

vať s touto knižnicou. Samozrejme by sme chceli prilinkovať ovládač dynamicky, aby mohol byť substituovaný iný ovládač (napr. pre iné zariadenie rovnakého všeobecného typu) bez nutnosti opätovného prekladu aplikácie.
Musíme vyriešiť nejaké problémy, čo sa týka oprávnení, ale v princípe môže takýto vnútroprocesový (knižničný) ovládač používať rovnaké systémové volania ako ovládač bežiaci vo svojom vlastnom procese. Následky možných chýb alebo nesprávneho chovania v ovládači sú obmedzené na konkrétny proces, v ktorom ovládač beží. Tiež sa často stáva, že niekoľko procesov môže používať rovnaké zariadenie, pričom každý používa svoju vlastnú 'inštanciu' ovládača.
Tento model však nie je použiteľný ak ovládač potrebuje byť chránený pred aplikáciou alebo ak ovládač musí realizovať multiplexovanie (t.j. nie je možné, aby niekoľko nezávislých inštancií ovládača komunikovalo s rovnakým zariadením, ale napriek tomu musí byť možné zariadenie využívať z viacerých procesov).

Portovo-mapované IO

366

- prvé CPU mali veľmi obmedzený adresný priestor
  - 16-bitové adresy znamenali 64KB pamäte
- periférie dostali osobitný adresný priestor
- špeciálne inštrukcie na používanie týchto adries
  - napr. `in` a `out` na x86 procesoroch

Poďme sa pozrieť na to, ako CPU komunikuje s perifériami a ako toto ovplyvňuje ovládače. Niektoré staré CPU (najznámejší je asi Intel 8086) mali 2 rôzne adresné priestory, jeden pre pamäť a ďalší pre periférie. K tomu druhému sa dalo pristupovať pomocou špeciálnych inštrukcií, ktoré presúvali hodnoty medzi CPU registrami a perifériami. V neskorších iteráciách rodiny x86, keď bola pridaná ochrana pamäte (MMU a úrovne ochrany), sa stali tieto inštrukcie privilegovanými. To znamená, že so zariadeniami, ktoré sú pripojené k CPU pomocou tohto mechanizmu, sa môže rozprávať iba kernel. Ale od IO inštrukcií bolo z veľkej časti upustené a dnes sa používajú iba v zastaralých zariadeniach.

Pamäťovo-mapované IO

367

- zariadenia zdieľajú adresný priestor s pamäťou
- častejšie u súčasných systémov
- IO používa rovnaké inštrukcie ako prístup do pamäte
  - `load` a `store` na RISC, `mov` na x86
- umožňuje selektívny prístup na užívateľskej úrovni (cez MMU)

Alternatívou k portovo-mapovanému IO je pamäťovo-mapované IO (skrátene MMIO), kde je fyzický adresný priestor zdieľaný RAM a perifériami. Zapisovanie na nejaké adresy (použitím napr. `mov` na x86) uloží dáta do RAM, ale jednoduchá zmena adresy spôsobí, že dáta budú poslané na periférium (kde typicky budú uložené v registroch alebo pamäti zariadenia). Konkrétnym príkladom je PCIe konfiguračná oblasť: každé PCIe zariadenie musí sprístupniť jednu stránku (4KiB) MMIO adresného priestoru, cez ktorú môže byť nájdené a nakonfigurované. Na rozdiel od portovo-mapovaného IO je prístup do fyzického adresného priestoru pamäte spravovaný MMU (vrátane oblastí pridelených zariadeniam, nielen tých, ktoré patria RAM). Je teda možné bezpečne umožniť nejakému procesu rozprávať sa s konkrétnym zariadením namapovaním odpovedajúceho kusu fyzického adresného priestoru do virtuálneho adresného priestoru tohto procesu.

Programované IO

368

- vstup alebo výstup je riadený CPU
- CPU musí čakať kým nie je zariadenie pripravené
- obvykle beží rýchlosťou zbernice
  - 8 MHz pre ISA (a teda ATA-1)
- PIO sa rozpráva s bufferom na zariadení

Ďalší spôsob, ako nahliadať na IO je, ako je časované. Periférie sú obvykle o niekoľko rádov pomalšie, než hlavné CPU a CPU musí čakať veľké množstvo cyklov medzi, napríklad, zadávaním príkazov danému zariadeniu. Príkazy sú obvykle realizované zapisovaním dát do registrov na zariadení. Zariadenie periodicky číta tieto registre a podľa toho sa správa, prípadne zapíše odpoveď do nejakého iného registra, ktorú si CPU potom môže prečítať (aj vstup aj výstup je realizovaný jedným z mechanizmov popísaných vyššie: portovo-mapované alebo pamäťovo-mapované IO).
Najjednoduchšia forma časovania sa nazýva programované IO alebo PIO. V tomto režime CPU riadi dátové prenosy a musí aktívne čakať na zariadenie (alebo skôr na zbernicu), kým nie je pripravené, po každom prenose. Vezmite si posielanie dát na disk: v diskovom radiči je buffer založený na RAM, ktorý dokáže udržiavať aspoň veľkosť jedného fyzického diskového sektora dát. CPU môže prenášať dáta do tohto bufferu rýchlosťou zbernice, napr. 8MHz pre ISA (aj keď konkrétne ISA je veľmi stará technológia). Ak CPU jadro beží na 32MHz, znamená to, že môže posielať dáta iba každý štvrtý cyklus. Musí stráviť 3 z každých 4 cyklov čakaním, až bude zbernica pripravená.

Prerušeniami-riadené IO

369

- periférie sú výrazne pomalšie než CPU
  - pollovanie (aktívne dotazovanie) zariadenia je drahé
- periférie môžu signalizovať dostupnosť dát
  - a tiež pripravenosť prijať ďalšie dáta
- to uvoľňuje CPU aby mohlo v medzičase robiť inú prácu

Niektoré periférie dokážu spracovávať iba veľmi malé množstvo dát naraz, a sú stále výrazne pomalšie než zbernica. Ako extrémny príklad si vezmime sériový port nakonfigurovaný, aby posielal 9600 bitov za sekundu. To činí asi 1200 znakov za sekundu: ak má zariadenie buffer na 60 znakov, CPU potrebuje naplniť tento buffer v 20Hz, t.j. s periódou 50 milisekúnd, čo je samozrejme v CPU čase večnosť (pri 32MHz takmer 2 milióny cyklov).
Tak by ste možno použili PIO na naplnenie tohto 60-znakového bufferu (rýchlosťou zbernice, teda s asi 25% efektivitou, čo by činilo 240 cyklov), ale aktívne čakanie na to, kedy sa buffer vyprázdni by bolo šialenstvo. Našťastie môže byť hardvér sériového portu nakonfigurovaný aby vyvolal prerušenie keď sa buffer vyprázdni. CPU si môže robiť čo chce, ale radič sériového portu bude prebudený, keď bude potrebné naplniť ďalších 60 bajtov, každých cca 50ms.
Rovnaký mechanizmus môže byť použitý na prijímanie dát: hardvér vyvolá prerušenie keď sa prijímací buffer zaplní a potrebuje byť prečítaný CPU.

Obsluhy prerušenia

370

- tiež známe ako obsluha prerušenia prvej úrovne
- musia bežať v privilegovanom režime
  - sú súčasťou kernelu z definície
- nízkoúrovňová obsluha prerušenia musí skončiť rýchlo
  - bude maskovať svoje vlastné prerušenie, aby zabránila opätovnému vstupu
  - a naplánuje všetky dlho-trvajúce úlohy na neskôr (SLIH)

Pri hardvérovom prerušení CPU preruší čokoľvek práve robilo, uloží svoj aktuálny stav do na to určenej oblasti v pamäti a predá kontrolu obsluhe prerušenia (interrupt handler). Alebo to skôr urobí jedno z CPU jadier. Obsluha automaticky beží v privilegovanom režime, a je teda z definície súčasťou kernelu.
Všimnite si, že sa nedeje žiadne prepínanie kontextu: hoci sú registre zapísané do pamäte, tabuľka stránok nie je nijak ovplyvnená – obsluha prerušenia beží v kontexte procesu, ktorý bol v tom čase práve vykonávaný. Je to analogické tomu, ako sa správajú systémové volania.
Aby sme sa vyhli problémom s viacnásobným volaním (reentrancy), obsluha prvej úrovne bude obvykle maskovať svoje vlastné prerušenie (spôsobí, že CPU ho bude dočasne ignorovať). Toto je jeden z niekoľkých dôvodov, prečo musí obsluha prvej úrovne skončiť rýchlo (ak je prerušenie dlho maskované, môže dôjsť ku strate dát, napr. kvôli pretečeniu buffera). Preto obsluha prvej úrovne obvykle robí iba minimum požadovanej práce (napr. vyprázdni časovo-kritické buffery) a naplánuje akékoľvek ďalšie spracovanie na neskôr (SLIH – Second Level Interrupt Handler).

Obsluha druhej úrovne

371

- robí všetko drahé spracovanie súvisiace s prerušením
- môže byť vykonávaná kernelovým vláknom
  - ale tiež radičom na užívateľskej úrovni
- zvyčajne nie je časovo kritická (na rozdiel od obsluhy prvej úrovne)
  - môže využívať štandardné zamykacie mechanizmy

Obsluha druhej úrovne si preberie prácu, ktorú odložila obsluha prvej úrovne. Obsluha druhej úrovne môže bežať v kernelovom vlákne alebo dokonca v procese v užívateľskom režime. Zvyčajne nie je časovo kritická a môže sa podľa potreby synchronizovať so zvyškom systému. Obsluha druhej úrovne diskového zariadenia by napríklad mohla urobiť volanie do súborového systému, aby ho informovala, že prišiel kus dát, ktoré si vyžiadal, čo v zápätí môže zobudiť pozastavené systémové volanie `read`, ktoré zapíše dáta do adresného priestoru čakajúceho procesu. Systémové volanie sa potom vráti a proces je prebudený.

Priamy prístup do pamäte – Direct Memory Access

372

- umožňuje zariadeniu priamo čítať/zapisovať do pamäte
- je to obrovské zlepšenie oproti programovanému IO
- prerušenia indikujú, že buffer je plný/prázdny
- zariadenia môžu čítať a zapisovať do ľubovoľnej fyzickej pamäte
  - vznikajú bezpečnostné problémy a problémy so spoľahlivosťou

Posledný IO režim je známy ako DMA, alebo Direct Memory Access (priamy prístup do pamäte). Hoci existuje určitá povrchová podoba s MMIO (pamäťovo-mapované IO), je dôležité ich rozlišovať. Pri MMIO CPU (a teda, tranzitívne, OS) komunikuje so zariadením pomocou pamäťového subsystému, mapovaním zabudovanej pamäte zariadenia alebo registrov zariadenia do fyzického adresového priestoru CPU.
Situácia v DMA je prevrátená: CPU a zariadenie spolu navzájom vôbec nekomunikujú. Namiesto toho je fyzická pamäť, ktorá patrí CPU (t.j. hlavná RAM) sprístupnená periférii, ktorá potom môže presunúť dáta do RAM. CPU stále používa inštrukcie prístupu do pamäte, aby vyzdvihlo dáta, ktoré prišli od zariadenia (ako pri MMIO), ale nekomunikuje so zariadením priamo. Namiesto toho číta a zapisuje do svojej vlastnej RAM, ktorá náhodou obsahuje dáta, ktoré tam zariadenie zapísalo, alebo si neskôr prečíta.
Zhrnutie:

- MMIO aj DMA používajú inštrukcie prístupu do pamäte na CPU, aby čítali a zapisovali dáta,
- pri MMIO hlavná pamäť vôbec nehraje úlohu,
- pri DMA obe zariadenie aj CPU pristupujú do hlavnej pamäte,
- pri DMA sa nedeje priamy hromadný prenos dát medzi CPU a zariadením.

Použitie MMIO a DMA nie je výlučné, skôr naopak: zariadenia často používajú kombináciu oboch. V skutočnosti môže MMIO byť použité na konfiguráciu DMA (DMA nie je vhodné na konfigurovanie, ale má lepší výkon pre hromadný presun dát).

IO-MMU

373

- ako MMU, ale pre DMA prenosy
- umožňuje OS obmedziť pamäťový prístup pre každé zariadenie
- veľmi užitočné pri virtualizácii
- iba nedávno si našlo cesto do spotrebiteľských počítačov

Hoci je DMA extrémne dôležitá pre zariadenia, ktoré prenášajú veľa dát (HDD, SSD, NIC), má určité nepríjemné problémy, čo sa týka bezpečnosti a spoľahlivosti. Pri 'tradičnom' DMA môže zariadenie čítať a zapisovať do akejkoľvek fyzickej pamäte, do akej chce. Napríklad, ak sa mu zachce, môže prepísať kód kernelu. Úkladné zariadenie by potom mohlo jednoducho obísť všetku bezpečnostnú ochranu na softvérovej úrovni. Čo je možno ešte dôležitejšie, zákerný ovládač môže naprogramovať zariadenie, aby prepísalo pamäť dátami (a kódom), ktoré si ovládač povie.
Toto je nežiaduce, obzvlášť keď chceme používať ovládače v užívateľskom režime, alebo ak zariadenie nie je dostatočne bezpečné.[3] IO-MMU je zariadenie, ktoré rieši tento problém, tým, že vynucuje hranice, na ktorú pamäť môže konkrétna periféria siahať. IO-MMU, rovnako ako obyčajné MMU, môže byť naprogramované iba kernelom OS (alebo hypervízorom,... o týchto sa viac dozvieme v Kapitole 11). So správne naprogramovanou IO-MMU je DMA bezpečná a spoľahlivá.

Časť 7.2: Systémové a rozširujúce zbernice

Zvyšok prednášky bude prehliadka periférií a trochu ich histórie. Než sa však dostaneme k samotným perifériám, pozrieme sa na zbernice, ktoré sa používajú na prepojenie periférií s CPU (alebo niekoľkými CPU) a, v niektorých prípadoch, s RAM. Hoci zbernica nie je periféria, často majú zbernice svoje vlastné ovládače. A to z dvoch dôvodov:

1. všetky zbernice okrem najjednoduchších majú dodatočný hardvér, ktorý sprostredkúva prístup ku zbernici, stará sa o konfiguráciu zariadenia a jeho nájdenie, a tak ďalej, a ktorý sám potrebuje byť

[3] Ako je známe, akékoľvek zariadenie pripojené do firewire portu – externý port, niečo na spôsob vysokorýchlostného USB než prišlo USB 3 – môže čítať a zapisovať do akejkoľvek pamäte hostiteľského počítača. Nie je nemysliteľne ťažké postaviť nečestné firewire zariadenie a pripojiť ho k niekoho počítaču. Ďalšie konektory, ktoré sprístupňujú vysokorýchlostné zbernice, môžu na to tiež byť náchylné.

nakonfigurovaný,
2. okrem elektroniky a signalizácie so sebou zbernica tiež koncepčne prináša sadu protokolov, ktoré musia byť implementované periférioru aj jej ovládačmi; tieto protokoly implementuje ovládač zbernice: ostatné ovládače realizujú jednoduché volania funkcií a ovládač ich prekladá na požadované MMIO alebo portovo-mapované IO operácie.

Poďme sa pozrieť na niektoré historické zbernice, ktoré sa v priebehu času používali v počítačoch a ako sa vyvinuli do moderného, v súčasnosti používaného systému, PCI Express.

História: ISA (Industry Standard Architecture) 375

- 16-bitová systémová rozširujúca zbernica na IBM PC/AT
- programované IO a prerušenia
- fixný počet hardvérovo-konfigurovaných prerušení
  - podobne pre I/O rozsahy portov
  - HW nastavenia potom musia byť ručne prepísané pre SW
- paralelný prenos dát a adries

Jedna z najstarších rozširujúcich zberníc, ktorá sa objavila s IBM PC/AT (osobný počítač založený na Intel 80286). Zbernica ISA bola pripojená k CPU cez IO porty (nie MMIO) a poskytovala linku prerušenia pre každú periférium. DMA radič poskytoval obmedzený počet DMA 'kanálov', ktoré umožňovali pripojeným periférium (najmä úložným zariadeniam), aby prenášali dáta do a z pamäte nezávisle od hlavného CPU.[4]
Nebolo možné softvérovo prehľadať (enumerate) túto zbernicu, už vôbec nie nakonfigurovať periférie. Rozsahy portov a IRQ linky boli zvolené hardvérom (buď napevno zakódované, alebo nakonfigurované jumpermi alebo prepínačmi) a museli byť dodané ovládaču užívateľom (t.j. museli byť ručne 'vyklikané').
Čo sa týka hardvéru, zbernica bola navrhnutá paralelne, synchrónne prenášajúc 16 bitov cez 16 liniek v rovnakom tiku hodín. Boli použité oddelené linky pre dáta a adresy: adresa mohla byť prenášaná v rovnakom tiku hodín ako dátové slovo.

MCA, EISA 376

- MCA: Micro Channel Architecture
  - vlastníctvo IBM, patentovo-chránená
  - 32-bitová, softvérovo-riadená konfigurácia zariadenia
  - drahá a v konečnom dôsledku trhové zlyhanie
- EISA: Enhanced ISA
  - 32-bitové rozšírenie ISA
  - vytvorená hlavne na vyhnutie sa licenčným poplatkom MCA
  - krátka existencia, nahradená PCI

Pri 8MHz a 16 bitoch, ISA nakoniec začala byť limitujúcim faktorom, keďže procesory aj periférie – najmä grafické adaptéry, ale tiež úložné zariadenia – sa stávali výrazne rýchlejšími.

[4] V tomto nastavení sa DMA radič reálne stáva správcom zbernice (master) a vykonáva prenos. Hoci je tento efekt v podstate rovnaký, implementácia je o dosť iná, než u DMA založenom na tom, že sa správcami zbernice stávajú periférie, s ktorou sa stretneme neskôr v prednáške.

VESA lokálna zbernica 377

- pamäťovo-mapované IO & rýchle DMA na inak ISA systémoch
- späté s líniou 80486 Intel-ových (a AMD) CPU
- predovšetkým pre grafické karty
  - ale tiež používané s pevnými diskami
- rýchlo sa prestali používať s príchodom PCI

VESA Local Bus (lokálna zbernica), alebo VLB, bola pomerne úspešnou snahou štandardizovať nesúrodú sadu domácich zberníc, navrhnutých na vyrovnanie sa rýchlejšiemu grafickému hardvéru, než aký bol možný s ISA, a súčasne vyhnutie sa licenčným poplatkom MCA.
VLB v podstate pripájala periférie priamo k 80486 pamäťovej zbernici, použitím dodatočného konektora (ako rozšírenie štandardnej ISA). Kvôli nekompatibilnému návrhu pamäťovej zbernice v neskorších procesoroch VLB neprežila aktualizáciu na Pentium.

PCI: Peripheral Component Interconnect 378

- 32-bitový nástupca ISA
  - 33 MHz (v porovnaní s 8 MHz pri ISA)
  - neskoršie revízie mali 66 MHz, PCI-X 133 MHz
  - s podporou pre bus-mastering a DMA
- ešte stále zdieľaná, paralelná zbernica
  - všetky zariadenia zdieľajú rovnakú sadu vodičov

Prelomový bod v prepojeniach periférií prišiel s PCI, ktoré poskytlo väčšinu výhod MCA, zatiaľ čo sa vyhlo niektorým jeho problémom. Možno najdôležitejšou zmenou bola softvérom-riadená konfigurácia, ale výrazné zlepšenie priepustnosti tiež nebolo na škodu. Z modernej perspektívy bola jednou nevýhodou topológia: zdieľaná, paralelná zbernica spájajúca všetky zariadenia v systéme.
Paralelná tu znamená, že v každom cykle hodín je prenášaných 32 bitov, po 32 samostatných linkách. To obmedzuje dosiahnuteľné rýchlosti hodín, kvôli rozdielom v oneskoreniach signálov po trasách rôznej dĺžky – moderné zbernice prenášajú dáta sériovo, každá dátová linka na svojich vlastných hodinách.

Bus Mastering 379

- typicky je CPU tzv. master (správca) zbernice
  - čo znamená, že ustanovuje komunikáciu
- je možné mať niekoľko masterov
  - musia sa zhodnúť na protokole na rozhodovanie konfliktov
- obvykle sa používa na prístup do pamäte

Na zdieľanej zbernici je jedno zo zariadení obvykle master (správca) a má na starosti zbernicu a premávku na nej. Toto je typicky CPU. Pri DMA prenosoch (medzi pamäťou a periférioru) by však CPU nemalo byť zapojené do prenosu, keďže celá pointa je uvoľniť CPU, aby mohlo vykonávať inú činnosť kým prebieha prenos.
Na zabezpečenie týchto prenosov sa môžu periférie dočasne stať mastermi zbernice, a riadiť premávku. Je prítomný tzv. arbitrážny (rozhodcovský) protokol, ktorý zabezpečuje, aby v ktoromkoľvek bode mala zbernica iba jedného správcu (master), ktorý ju ovláda.

DMA (Direct Memory Access) – priamy prístup do pamäte

380

- najčastejšia forma bus mastering
- CPU povie zariadeniu čo a kde má zapísať
- zariadenie potom pošle dáta priamo do RAM
  - CPU zatiaľ môže pracovať na iných veciach
  - dokončenie je signalizované prerušením

V princípe je možné, aby sa periférie navzájom rozprávali, keď je jedna z nich master zbernice. Toto sa však zvyčajne nerobí: namiesto toho vykonáva (dočasný) master zbernice dátový prenos do alebo z hlavnej pamäte.

Plug and Play

381

- ISA systém pre IRQ konfiguráciu bol chaotický
- MCA bolo priekopníkom so softvérovo-konfigurovanými zariadeniami
- PCI ešte viac vylepšilo MCA s "Plug and Play"
  - každé PCI zariadenie má ID ktoré môže povedať systému
  - umožňuje nájdenie a automatickú konfiguráciu

Dôležitým aspektom PCI (a predtým MCA) bola softvérovo-založená konfigurácia a nájdenie (rozpoznanie) pripojených zariadení. To umožňuje firmvéru a operačnému systému zistiť, ktoré zariadenia sú pripojené, načítať odpovedajúce ovládače a nastaviť zariadenia bez zásahu užívateľa.

PCI ID a ovládače

382

- PCI umožňuje rozpoznanie zariadení
- identifikátory zariadení môžu byť spárované s ovládačmi zariadení
- to umožňuje OS načítať a nakonfigurovať svoje ovládače
  - alebo dokonca stiahnuť / nainštalovať ovládače od dodávateľa

Rozpoznanie má dva komponenty: jedným je systém na nájdenie a konfiguráciu zariadení pripojených k systému. Toto sa robí použitím spoločného protokolu nezávislého na zariadení, ktorý musia implementovať všetky PCI zariadenia.
Druhým je systém na prideľovanie unikátneho identifikátora každej periférii, takzvané PCI ID. Operačný systém potom môže obsahovať databázu známych PCI ID a ovládačov odpovedajúcich tomuto zariadeniu. Načítanie tohto ovládača obvykle sprístupní zariadenie na použitie zvyšku operačného systému, a teda užívateľovi.

AGP: Accelerated Graphics Port

383

- PCI sa nakoniec stalo príliš pomalým pre GPU
  - AGP je založené na PCI a iba zlepšuje výkon
  - rozpoznávanie a konfigurácia zostávajú rovnaké
- pridáva dedikované point-to-point spojenie
- niekoľko prenosov za cyklus hodín (až do 8, pri 2 GB/s)

Samozrejme, keďže vrchol má pri asi 4 Gib/s (500 MiB/s), PCI nie je koniec príbehu. Ako plynie z jasného historického vzoru, grafický hardvér začal byť obmedzovaný svojím pripojením k zvyšku systému (CPU a pamäť). Podobne ako pri VLB sa stala rozšírenou dedikovaná grafická zbernica, tentokrát založená na PCI, s prakticky dvomi modifikáciami:

1. zbernica bola point-to-point (dedikovaná pre jedinú periférium), t.j. nebola zdieľaná s hlavnou PCI zbernicou v systéme,
2. umožňovala niekoľko dátových prenosov v jednom cykle hodín – rovnaká technika, akú používa DDR RAM na zvýšenie priepustnosti bez zrýchlenia hodín.

Pri maxime 8 prenosov za hodinový cyklus, s hlavnými hodinami bežiacimi pri 66MHz, vychádza maximálna rýchlosť prenosu na 16Gib/s.

PCI Express

384

- súčasná vysokorýchlostná zbernica pre periférie pre PC
- založená na / rozširuje konvenčné PCI
- point-to-point, sériový prenos dát
- výrazne zlepšená priepustnosť (až do ~30GB/s)

Konečne sme sa dostali k súčasnosti. Moderný nástupca PCI sa presunul od synchrónneho paralelného prenosu dát a od zdieľanej zbernice, čím umožnil drastický nárast výkonu. Hoci je na dátový prenos použitých viacero vodičov, každé z nich má vlastný hodinový signál (ktorý je súčasťou dátového signálu) a teda asynchrónne voči sebe navzájom. Každý vodič sa nazýva linka - 'lane' a jedna periféria môže používať až 16 liniek. Zariadenia s nízkou priepustnosťou potrebujú iba jednu linku, čím šetria na energetických požiadavkach a nákladoch na výrobu.
V čase písania tohto textu sú zariadenia, ktoré cielia na PCIe 4.0, so 16GT (miliardami transakcií) za sekundu na každej linke, bežne dostupné. Znamená to maximálnu priepustnosť 256 Gib/s na jedno zariadenie (v porovnaní s AGP pri 16 Gib/s) alebo 32 GiB/s pri 16-linkovej konfigurácii.
Ďalšia revízia, PCIe 5.0 (posledná špecifikácia vydaná v 2019) zdvojnásobuje rýchlosť prenosu na linku na 32GT/s, čo dáva maximum 64 GiB/s na zariadenie.
Softvérovo je PCIe spätne kompatibilné s PCI, používajúc rozšírenú verziu PCI rozpoznávania a konfiguračného protokolu. PCIe ďalej umožňuje, aby konfigurácia používala MMIO namiesto portovo-mapovaného IO, sprístupňujúc jednu 4KiB stránku konfiguračných dát na každom koncovom bode (periférii).

USB: Universal Serial Bus

385

- primárne pre externé periférie
  - klávesnice, myši, tlačiarne, ...
  - nahradilo veľké množstvo zastaraných portov
- neskoršie revízie umožňujú vysokorýchlostné prenosy
  - vhodné pre úložné zariadenia, kamery ap.
- rozpoznávanie zariadenia, capability negotiation

PCI prinieslo softvérovo-založené rozpoznávanie (enumeration) a konfiguráciu pre permanentne pripojené, interné periférie (grafický hardvér, úložné zariadenia, sieťové rozhrania, a tak ďalej). USB urobilo to isté pre externe pripojené zariadenia, ako klávesnice, myši, tlačiarne, skenery, a tak ďalej.
Skoršie systémy používali na rovnaký účel pomerne 'hlúpe' zbernice. Užívateľ musel ručne vybrať ovládač a nakonfigurovať ho (povedať mu, na ktorý externý port je zariadenie pripojené). S USB sa zariadenia dokážu identifikovať samé, použitím protokolu nezávislého na zariadení, rovnako ako u PCI. Hostiteľský systém potom môže automaticky načítať a nakonfigurovať správny ovládač. USB navyše podporuje tzv. hotplug, takže toto sa môže diať kedykoľvek užívateľ pripojí zariadenie. Konečne, priepustnosť dostupná na USB, už v jeho prvej revízii, bola výrazne vyššia, než predchádzajúce štandardy (RS-232, PS/2).
Neskoršie USB revízie výrazne zvýšili rýchlosť dátového prenosu aj množstvo elektrickej energie (výkon) dostupný pripojenej periférii. Aktuálna najvyššia rýchlosť dostupná pre USB zariadenia (v USB 3.2 Gen

2 režime s 2 linkami, cez USB-C konektory) je 20Gib/s, čím prevyšuje maximálne rýchlosti prenosu AGP, najrýchlejšej internej zbernice dostupnej v spotrebiteľskom hardvéri pred PCIe.

USB triedy

386

- sada protokolov nezávislých na výrobcovi
- HID = human-interface device - zariadenie pre užívateľské rozhranie
- veľkokapacitná pamäť = diskom-podobné zariadenia
- audio príslušenstvo
- tlač

USB prináša dodatočnú štandardizáciu, s takzvanými triedami zariadení. Každá trieda predstavuje protokol nezávislý na výrobcovi pre určitý typ zariadení:

- HID (zariadenie pre užívateľské rozhranie), napr.:
  - klávesnice,
  - myši,
  - herné ovládače,
  - malé znakové displeje,
  - v podstate všetko s tlačidlami.
- veľkokapacitná pamäť (energeticky nezávislá pamäť, obvykle so súborovým systémom):
  - USB kľúče,
  - externé pevné disky alebo SSD,
  - optické disky,
  - čítače kariet, ...
- audio zariadenia, napr.:
  - headsety (slúchadlá s mikrofónom),
  - zvukové karty,
  - aktívne reproduktory,
  - samostatné mikrofóny,
  - MIDI zariadenia,
- MTP (media transfer protocol),
  - smartfóny,
  - prenositeľné prehrávače médií.
- tlačiarne,
- video (webkamery, digitálne mikroskopy).

Žiadne zo zariadení v zozname vyššie v podstate nepotrebuje ovládače, ktoré sú špecifické pre výrobcu. Namiesto toho môže jeden ovládač pre 'triedu' zariadení, ktorá implementuje príslušný protokol komunikovať s akoukoľvek periférioiu, ktorá patrí do tejto triedy. Jedna fyzická periféria môže poskytovať viacero virtuálnych zariadení, ktoré môžu prípadne patriť do iných tried (napr. diktafón, ktorý slúži ako audio zariadenie – mikrofón, aj ako úložné zariadenie).

Ďalšie použitia USB

387

- skenery
- ethernetové adaptéry
- usb-sériové adaptéry
- wifi adaptéry (dongles)
- bluetooth

Nad rámec štandardných tried zariadení existuje veľa USB zariadení, ktoré sa nehodia do jednej z týchto kategórií. Tieto budú používať protokol výrobcu, a budú potrebovať odpovedajúci ovládač špecifický pre zariadenie.

Bluetooth

388

- bezdrôtová alternatíva k USB
- umožňuje rádiové spojenie s periférami na krátku vzdialenosť
  - vstup (klávesnice, myši, herné ovládače)
  - audio (headset, reproduktory)
  - prenos dát (napr. synchronizácia smartfónu)
  - iné (hodinky, monitory frekvencie tepu, GPS, ...)

Hoci bluetooth nie je zbernica ako taká (keďže je bezdrôtový), z pohľadu softvéru sa chová veľmi podobne USB (s dodatočnou zložitosťou plynúcou z párovania zariadení, bezpečnosti a nespoľahlivého prenosu dát). Veľa typov zariadení, ktoré môžu byť pripojené cez USB môže byť tiež pripojených cez bluetooth (bezdrôtové klávesnice, myši, headsety, reproduktory, a tak ďalej).

ARM zbernice

389

- ARM je typicky použitý v návrhoch „Systém na čipe“
- tieto používajú na pripojenie periférií proprietárnu zbernicu
- menej potreby na rozpoznávanie zariadení
  - celý systém je na jedinom čipe
- periférie môžu byť predkonfigurované

Ekosystém ARM-u je trochu odlišný od PC. Často sa stáva, že ARM zariadenia sú navrhnuté ako 'systém na čipe', kde je väčšina, ak nie všetky, periférie súčasťou jedného čipu, spolu s jadrami CPU, pamäťovým radičom, a ich prepojením (systémová zbernica). Dodávatelia SoC (System on a Chip) väčšinou pripravia obrazy operačného systému alebo verziu kernelu (typicky Androidu), ktoré fungujú na ich systéme. Softvérovo-riadené rozpoznávanie a autokonfigurácia sú výrazne menej dôležité a typicky nie sú podporované.
Typicky zahrnuté periférie sú grafické jadro, USB radič, wifi, ethernet, radič pre bluetooth, audio radič, NFC, radič úložiska (eMMC) a prípadne niekoľko ďalších.

USB a PCIe na ARM

390

- ani USB ani PCIe nie sú výhradne určené pre platformu PC
- väčšina ARM SoC podporuje USB zariadenia
  - pre pomalé a stredne-rýchle zariadenia mimo SoC
  - napr. použité pre ethernet na RPi 1
- niektoré ARM SoC podporujú PCI Express
  - to umožňuje mať vysokorýchlostné periférie mimo SoC

Nie všetky ARM procesory sú však navrhnuté pre 'uzavreté' zariadenia ako smartfóny alebo chytré TV. Všeobecný (general-purpose) hardvér založený na ARM zahŕňa jednodoskové (single-board) počítače (ako Raspberry Pi, Beaglebone, ...), ale tiež laptopy (nová generácia Apple hardvéru) a servery (Ampére Altra). Tieto systémy často potrebujú viac konektivity a rozšíriteľnosti a poskytujú PCI Express na pripojenie k vysokorýchlostným periférium.

PCMCIA & karta PC Card

391

- People Can't Memorize Computer Industry Acronyms
  - PC = Personal Computer, MC = Memory Card
  - IA = International Association
- rozširujúca zbernica pre notebooky, so schopnosťou hotplug
- používa sa pre pamäťové karty, sieťové adaptéry, modemy
- má vlastnú sadu ovládačov (cardbus)

Vráťme sa k histórii: až do zhruba posledného desaťročia bolo časté, že laptopy mali rozširujúce sloty, niečo ako tradičné desktopy. Samozrejme, rozširujúca karta štandardnej veľkosti nemala šancu vojsť sa do laptopu, preto boli potrebné špeciálne konektory a/alebo zbernice. Jedným z najstarších bolo PCMCIA, s periférami veľkosti kreditnej karty (ale hrubšími), ktoré mohli byť „hot-plugged" (vložené za behu) do rámčekov/zásuviek (bay) na strane laptopu (t.j. zariadenie bolo skryté vnútri tela laptopu, na rozdiel od rôznych USB dongles, ktoré majú spleť káblov).

ExpressCard

392

- štandard rozširujúcej karty ako PCMCIA / PC Card
- založené na PCIe a USB
  - väčšinou môže použiť ovládače pre tieto štandardy
- už nie je veľmi používané
  - posledná aktualizácia bola v roku 2009, zaviedla podporu pre USB 3
  - združenie pre toto odvetvie bolo rozpustené ešte v tom roku

ExpressCard je modernejšia verzia rovnakej myšlienky a podobného formátu, s USB a PCIe v pozadí. Moderné laptopy však už neponúkajú túto funkcionalitu a združenie zodpovedné za ExpressCard bolo rozpustené pred viac než dekádou.

miniPCIe, mSATA, M.2

393

- ide o fyzické rozhrania, nie špeciálne zbernice
- poskytujú určitý mix PCIe, SATA a USB
  - tiež ďalšie protokoly ako I^5C, SMBus, ...
- používajú sa najmä pre kompaktné SSD a bezdrôtové technológie
  - tiež GPS, NFC, bluetooth, ...

Čo prežíva sú konektory pre interné zariadenia v malom formáte: najmä pre SSD, ale tiež pre wifi adaptéry, bluetooth a podobné moduly. Tieto sú časté u laptopov a mini-ITX (malých desktopových) systémov. V závislosti od konkrétneho štandardu konektora (a typu), poskytujú pripojenie na rôzne druhy zberníc, vrátane PCIe (až do 4 liniek) a USB.

Časť 7.3: Grafika a GPU

Grafický hardvér bol vždy veľmi dôležitou súčasťou domácich počítačov aj profesionálnych pracovných staníc. Často tiež ide o najnáročnejšiu periférium v týchto systémoch, a tiež najzložitejšiu.

Grafické karty

395

- pôvodne iba zariadenie na riadenie displejov
- načítava pixely z pamäte a poskytuje signál displeju
  - prakticky DAC s hodinami
  - pamäť môže byť súčasťou grafickej karty
- vyvinuli schopnosti akcelerácie

Pôvodne grafická karta obsahovala v podstate iba rýchlu statickú pamäť (frame buffer), hodiny a digitálno-analógový prevodník (digital-to-analog converter, DAC), ktorý riadil CRT displej (cathode ray tube). Displeje tejto éry fungovali na princípe, že nasmerovali elektrónové delo (electron gun; použitím elektromagnetov) na individuálne pixely v rýchlom slede, zatiaľ čo modulovali napätie medzi katódou a anódou (ktorá pozostávala z vodivej vrstvy nanesenej na vnútro obrazovky), aby dosiahli príslušný jas na každom pixeli.
Grafická karta generovala signál, ktorý riadil túto moduláciu, držiac krok s postupujúcim elektrónovým delom. Pamäť grafickej karty obsahovala digitálne informácie o jase každého pixela. Typické frekvencie obnovovania displeja boli v rozsahu 30-120 Hz na celú obrazovku. Pre VGA obrazovku (640 stĺpcov, 480 riadkov) pri 70 Hz je to asi 20 MHz (20 miliónov pixelov za sekundu). Troj-komponentové farby sú prenášané paralelne.

Grafický akcelerátor

396

- umožňuje bežným operáciám, aby mohli byť realizované v hardvéri
- ako kreslenie čiar alebo vyplnených polygónov
- pixely sú vypočítané priamo vo video RAM
- môže to ušetriť veľké množstvo CPU času

Zostavenie obrázka, ktorý má byť zobrazený na displeji, môže zabrať veľa výpočetného času a/alebo premávky pamäte. Ak sú niektoré z týchto operácií vykonané dedikovaným hardvérom namiesto hlavného CPU, môže to drasticky zlepšiť výkon, keďže CPU zatiaľ môže vykonávať iné veci, zatiaľ čo grafický hardvér asynchrónne vykonáva jednoduché, repetitívne úlohy. Existujú dve hlavné triedy operácií, ktoré môžu byť jednoducho akcelerované použitím dedikovaného hardvéru:

- rasterizácia geometrických tvarov ako sú čiary, obdĺžniky, polygóny alebo krivky (vektorová grafika) – tieto sa používajú napríklad v grafických užívateľských rozhraniach a vo vektorových kresliacich programoch alebo 2D počítačovo podporovaných (computer-aided) návrhových systémoch,
- hromadné operácie s pixelmi, napríklad záplavové vypĺňovanie alebo bit blitting[6] používaný najmä v rastrovej grafike (napr. videohry).

Keďže prakticky každý pixel (alebo prinajlepšom malý blok pixelov) potrebuje aspoň jeden zápis do pamäte a pre CPU sú zápisy do pamäte drahé (veľa čakania na pomalú pamäť), takéto operácie obzvlášť plytvajú prostriedkami CPU. Ešte horšie je ak dáta (textúry, sprite) musia byť načítané z pamäte a zapísané inde, možno po vykonaní jednoduchej operácie na pixeloch.

[6] Pamäťová kópia s nejakou dodatočnou logikou: pracuje s pixelmi (namiesto bajtov) v rôznych formátoch (napr. 2 alebo 8 pixelov na bajt) a môže pracovať aj s priehľadnými pixelmi, ktoré sú vynechané (čo umožňuje kresliť tvary iné než obdĺžnikové cez existujúce pozadie).

3D Grafika

397

- renderovanie 3D scén je výpočetne náročné
- výhradne softvérové renderovanie, založené na CPU, je možné
  - v prvých leteckých simulátoroch neobsahovalo textúry
  - bitmapové textúry od '95 / '96 (Descent, Quake)
- CAD pracovné stanice mali 3D akcelerátory (OpenGL '92)

Hoci 2D grafika potrebuje veľa zdrojov (aspoň ak vezmeme do úvahy schopnosti staršieho hardvéru), je v podstate zadarmo v porovnaní s 3D grafikou, kde výpočet každého výstupného pixelu môže zabrať stovky operácií, z ktorých niektoré sú geometrické a iné sú rasterové. Preto je potenciál na hardvérovú akceleráciu 3D grafiky výrazne väčší než u 2D grafiky, ale hardvér, ktorý ju vykonáva je výrazne komplikovanejší.

GPU (Graphics Processing Unit)

398

- termín, s ktorým prišlo Sony v '94 (GPU v PlayStation)
- pôvodne hardvérový renderer vytvorený s určitým zámerom
  - založený na polygónových sieťach a Z bufferovaní
- stále viac flexibilný a programovateľný
- vstavaná RAM, vysokorýchlostné prepojenie so systémovou RAM

Prvé GPU boli v podstate hardvér vytvorený pre rasterizáciu 3D geometrie, ktorá bola dodaná vo forme polygónovej (trojuholníkovej) siete s textúrami priradenými k plochám. Hardvér vypočítal viditeľnosť a osvetlenie, aby vytvoril rasterový obrázok, ktorý sa má zobraziť na displeji. CPU pripravilo geometriu pre každý rámec, ktorý GPU potom vyrenderovalo a zobrazilo.
Každá generácia GPU priniesla viac flexibility a programovateľnosti, čím umožnila akceleráciu veľa rôznych efektov bez nutnosti ich napevno nakódovať v hardvéri. Súčasné GPU sú v podstate plne programovateľné všeobecné (general-purpose) vektorové procesory, s registrami, pamäťou, tokom riadenia (control flow) a tak ďalej.

GPU ovládače

399

- rozdelené do niekoľkých komponent
- grafický výstup / prístup do frame bufferu
- správa pamäte je často realizovaná v kerneli
- geometria, textúry ap. sú pripravované vnútri procesu
- front end API: OpenGL, Direct3D, Vulkan, ...

Typický GPU ovládač je rozdelený do niekoľkých komponent, z ktorých časť sídli v kerneli (nastavenie frame bufferu, správa pamäte), zatiaľ čo zložitejšie časti sú knižnicami, ktoré sa linkujú do klientských aplikácií (spracovanie geometrie a textúr, preklad shaderov).

Shadery

400

- súčasné GPU sú výpočetné zariadenia
- GPU má vlastný strojový kód pre shadery
- GPU ovládač obsahuje prekladač shaderov
  - buď úplne z vysokoúrovňového jazyka (HLSL)
  - alebo začínajúc na medzi-kóde (SPIR)

Keďže sú moderné GPU v podstate iba prevlečené vektorové procesory, spúšťajú programy vo svojom vlastnom strojovom kóde. Ovládač skompiluje vysokoúrovňové programy, ktoré sú súčasťou softvéru (napr. počítačová hra alebo 3D herné vývojové prostredie (engine)) do strojového jazyka špecifického hardvéru. Hoci je výstup viazaný na zariadenie, vstup (čo je to, čo aplikácia dáva ovládaču) je z veľkej časti štandardizovaný, pričom dve hlavné možnosti sú HLSL (High-Level Shader Language) a SPIR (Standard Portable Intermediate Representation).

Nastavenie režimu

401

- stará sa o konfiguráciu obrazovky a rozlíšenie
- vrátane podpory pre napr. viacero displejov
- obvykle tiež podporuje primitívny (výhradne softvérový) framebuffer
- často vnútri kernelu, s minimom užívateľskej podpory

Hoci má moderné GPU veľa komplikovaných vychytávok, stále existujú aj nudné úlohy, ktoré sa v podstate nezmenili za posledné 2-3 dekády, napríklad konfigurácia displeja. Je bežné, že dnešné počítače dokážu pripojiť niekoľko displejov, a každý potrebuje dostať nejaké rozlíšenie, farebnú hĺbku, frekvenciu obnovovania ap., dohromady známe ako grafický 'režim' - mode. Toto je úloha časti grafického ovládača pre nastavenie režimu ('mode setting').

Grafické servery

402

- niekoľko procesov nemôže naraz ovládať grafickú kartu
  - grafický hardvér potrebuje byť zdieľaný
  - jednou z možností je grafický server
- poskytuje kresliace a/alebo oknové API založené na IPC
- vykonáva kreslenie v mene aplikácií

Hoci nejde o ovládač, grafické servery tvoria dôležitú časť grafického zásobníka (na systémoch, ktoré ho používajú). Problém je, že iba jeden program dokáže v danom momente rozumne kresliť na nejakú obrazovku, ale my obvykle chceme zobraziť výstup viac než jedného programu. Jednou z možností je grafický server, ktorý prideľuje regióny (typicky obdĺžnikové okná), do ktorých môžu programy kresliť pomocou jeho API.

Kompozítory

403

- priamejší spôsob zdieľania grafických kariet
- každá aplikácia dostane vlastný buffer, do ktorého môže kresliť
- kreslenie je vykonávané najmä GPU (s prepínaním kontextu)
- jednotlivé buffery sú potom komponované (poskladané) na obrazovku
  - kompozícia je tiež hardvérovo-akcelerovaná

Ďalším častým prístupom je použitie kompozítora (compositor), ktorý sa kľúčovo líši od grafického serveru v jednej veci: ako jednotlivé aplikácie vykresľujú svoj obsah. V grafickom serveri je prítomné kresliace API, ktoré program volá na zobrazenie tvarov a bitmáp na obrazovku. U kompozítora dostane každý program buffer mimo obrazovku (bitmapa, pixmap), do ktorého môžu kresliť tým, že budú priamo interagovať s ovládačom grafického hardvéru. Kompozítor potom skombinuje tieto buffery do jediného obrázka, ktorý sa zobrazí užívateľovi (opäť vykonaním relevantných volaní do grafického ovládača). Pri typickom použití korešponduje každé okno s jedným bufferom.

GP-GPU

404

- univerzálne (general-purpose) GPU (CUDA, OpenCL, ...)
- používa sa aj na výpočty namiesto čisto na grafiku
- v podstate návrat vektorových procesorov
- podobné CPU, ale nie sú súčasťou normálneho OS plánovania

Ako sme už spomenuli, súčasné GPU sú v podstate všeobecné (general-purpose) vektorové procesory a môžu byť použité na čisto výpočetné úlohy, ktoré nijak nesúvisia s grafikou (jednou z častých aplikácií je strojové učenie, ale čokoľvek, čo má prospech z rozsiahleho SIMD je dobrým kandidátom).

Časť 7.4: Perzistentné úložiská

V tejto sekcii sa pozrieme na veľkokapacitné úložiská – tie, ktoré obvykle obsahujú súborové systémy a ktoré si zachovávajú uložené dáta, keď sú vypnuté (odpojené od energie).

Ovládače

406

- rozdelené na ovládače adaptérov, zberníc a zariadení
- často jeden ovládač na typ zariadenia
  - aspoň u pevných diskov a CD-ROM
- rozpoznávanie zberníc a konfigurácia
- dátová adresácia a dátové prenosy

Úložné zariadenia tradične mali svoju vlastnú dedikovanú, špecializovanú zbernicu. Na hostiteľskej strane je táto zbernica implementovaná adaptérom (radičom – controller), ktorý je na jednej strane pripojený k systémovej zbernici (PCI, PCIe), a na druhej strane k zbernici úložiska. Jednotlivé úložné zariadenia sú potom pripojené k tejto dedikovanej zbernici.
Táto hardvérová štruktúra v podstate diktuje štruktúru ovládača: zbernica je obvykle štandardizovaná a je spätá so sadou protokolov, rovnako ako je to u systémových zberníc, o ktorých sme sa bavili predtým. Pre akúkoľvek zbernicu však môže existovať veľa rôznych modelov adaptérov, vyrobených rôznymi výrobcami. V niektorých prípadoch používajú spoločný protokol, ale v iných prípadoch ich musia ovládače, špecifické pre dané zariadenie, nakonfigurovať.
Podobne ako u USB, u každej zbernice pre úložné zariadenia existuje významná úroveň štandardizácie medzi jednotlivými úložnými zariadeniami (koncovými bodmi), a stačí jeden ovládač pre 'triedu' zariadení (HDD ovládač, CD-ROM ovládač, ovládač páskovej jednotky, ...).

IDE / ATA

407

- Integrated Drive Electronics
  - radič disku (controller) sa stáva súčasťou disku
  - štandardizovaný ako ATA-1 (AT Attachment ...)
- založený na ISA zbernici, ale s káblami
- neskôr upravený pre použitie mimo diskov cez ATAPI

Jednou z najstarších štandardizovaných zberníc pre úložné zariadenia bolo IDE (názov dodávateľa, neskôr štandardizované ako ATA). Ide v podstate o ISA zbernicu s kabelážou, preto bol adaptér obzvlášť jednoduchý, ak bol pripojený k hostiteľskej ISA zbernici. Neskoršie špecifikácie revízie ATA (teraz známe ako Parallel ATA) sa však odklonili od ISA, kvôli nutnosti výrazne vyšších rýchlostí. Rodina ATA zberníc neprešla na interné použitie PCI a zbernica úložiska a systémová zbernica sa vyvinuli osobitne, hoci pozdĺž podobných línií.

ATA rozpoznávanie (enumeration)

408

- každé ATA rozhranie môže pripojiť iba 2 disky
  - disky sú HW-konfigurované ako master/slave
  - tým sa stáva rozpoznávanie pomerne jednoduchým
- niekoľko ATA rozhraní bolo štandardom
- nepotrebujeme špecifické HDD ovládače

Keďže väčšina implementácií poskytuje presne 4 konektory (2 rozhrania, každé schopné pripojiť 2 disky), rozpoznávanie nie je až taký problém. Každé rozhranie má štandardnú sadu IO portov (pre portovo-mapované IO). Systém používa tieto porty na poslanie 2 príkazov `IDENTIFY` na každom rozhraní, jeden pre master, druhý pre slave zariadenie. To je celé.

PIO vs DMA

409

- pôvodné IDE mohlo používať iba programované IO
- časom sa to stalo významným úzkym miestom (bottleneck)
- neskoršie ATA revízie zahŕňali DMA režimy
  - u najrýchlejších DMA režimov až do 160MB/s
  - v porovnaní s 1900MB/s pre SATA 3.2

SATA

410

- sériová, point-to-point náhrada ATA
- na hardvérovej úrovni nekompatibilné s (paralelným) ATA
  - ale SATA zdedila ATA sadu príkazov
  - režim spätnej kompatibility umožňuje PATA ovládačom komunikovať so SATA diskami
- schopnosť hot-swap – výmena diskov v bežiacom systéme

Podobne ako iné rozhrania, aj úložné systémy prešli na sériové dátové linky. Pri ATA je výsledok známy ako SATA alebo Serial ATA. Nový štandard zachováva spätnú kompatibilitu s Parallel ATA na softvérovej úrovni: ak je radič v 'režime spätnej kompatibility' (legacy mode), bude emulovať PATA hostiteľský radič a pracovať s PATA diskami. Tento PATA-kompatibilný režim však nutne skrýva nové funkcie (schopnosť pripojiť viacero diskov, hotswap, native command queuing – natívne radenie príkazov).

AHCI (Advanced Host Controller Interface)

411

- rozhranie pre SATA radiče nezávislé od dodávateľa
  - teoreticky je potrebný iba jediný 'AHCI' ovládač
- alternatíva k 'režimu spätnej kompatibility'
- NCQ = Native Command Queuing
  - umožňuje ovládaču preusporiadať požiadavky
  - ďalšia vrstva IO plánovania

Väčšina hostiteľských radičov SATA implementuje štandard AHCI a teda nepotrebuje ovládače špecifické pre zariadenie. Spustenie radiča v režime AHCI je potrebné, aby bolo možné využiť nové SATA technológie, ako NCQ (native command queuing - natívne radenie príkazov) a hotswap (výmena za behu). Hoci boli nejaké pokusy o pridanie radenia príkazov (command queuing) do PATA, tieto boli nakoniec neúspešné, kvôli nedostatočnej schopnosti DMA starých systémov založených na ISA (s externým DMA radičom). Keďže SATA disky samy robia DMA, NCQ má výrazne lepší výkon.

ATA a SATA ovládače

412

- hostiteľský radič (adaptér) je väčšinou nezávislý na výrobcovi
- ovládač zbernice sprístupní sadu ATA príkazov
  - vrátane podpory pre command queuing - radenie príkazov
- ovládač zariadenia používa na komunikáciu so zariadeniami ovládač zbernice
- čiastočne pre ATAPI ap. využíva existujúce SCSI ovládače

SCSI (Small Computer System Interface)

413

- vzniklo s minipočítačmi v 80. rokoch
- komplikovanejšie a schopnejšie než ATA
  - ATAPI v podstate zapuzdruje SCSI do ATA
- rozpoznávanie zariadení, vrátane agregátov
  - napr. externé diskové polia
- tiež pracuje s CD-ROM, páskovými jednotkami, skenermi (!)

Ďalšia zbernica úložísk, nazvaná SCSI, bola paralelne používaná s ATA, a cielila hlavne na servery a všeobecne na drahší hardvér. Celková štruktúra je rovnaká ako u ATA: je tam adaptér (nazvaný HBA – host bus adapter – v SCSI žargóne), zbernica so sadou protokolov, a pole úložných zariadení pripojené k zbernici úložiska.
Na rozdiel od Parallel ATA dokáže SCSI zbernica pripojiť výrazne viac zariadení a tieto zariadenia môžu mať dodatočnú vnútornú štruktúru (napr. je možné pripojiť SATA RAID radič s desiatkou diskov ako jediný zložený - 'composite' SCSI koncový bod. Preto má rozšírené schopnosti softvérového rozpoznávania (enumeration) a konfigurácie: HBA 'skenuje' zbernicu úložiska na rozpoznanie zariadení a ohlási ich operačnému systému. SCSI tiež bežne podporuje tzv. hotplugging zariadení (t.j. pripájanie a odpájanie zariadení kým systém beží). Na rozdiel od ATA, externé SCSI konektory a káble sú bežné.
Podobne ako u ATA (a systémových zberníc), SCSI dlho používalo paralelnú variantu, ale moderné verzie namiesto toho používajú vysokorýchlostné sériové linky. Táto technológia je známa ako SAS, Serial-Attached SCSI (sériovo pripojené SCSI). SAS môže voliteľne používať SATA-kompatibilný konektor (a SAS adaptéry s týmito konektormi budú fungovať so SATA diskami, ale nie naopak).

SCSI ovládač

414

- rozdelené na: ovládač hostiteľského adaptéra zbernice (HBA)
- generický komponent pre SCSI zbernicu a príkazy
  - často využívaný v ATAPI aj USB úložiskách
- a ovládače pre každé zariadenie alebo triedu zariadení
  - optické disky, páskové jednotky, CD/DVD-ROM
  - štandardné disky a SSD disky

Hoci je SCSI hardvér relatívne zriedkavý, protokoly, ktoré používa sú veľmi rozšírené: aj SATA aj USB úložné zariadenia totiž používajú SCSI ako svoj príkazový protokol. Fibre Channel (FC, sieťová technológia pre úložiská) a InfiniBand (IB, vysokorýchlostné prepojenie s nízkou latenciou) tiež ponúkajú SCSI implementácie. V podstate to znamená, že rovnaký ovládač pre 'triedu' môže byť použitý pre úložné zariadenia pripojené k SATA, USB, SAS, FC, IB alebo ethernetu (cez iSCSI, pozri nižšie), s vhodnou vrstvou lepidla.

iSCSI

415

- v podstate SCSI cez TCP/IP
- implementované výhradne softvérovo
- umožňuje štandardným počítačom, aby slúžili ako blokové úložisko
- využíva rýchly a lacný ethernet
- využíva väčšinu existujúcich SCSI ovládačov

SCSI protokol môže tiež byť zapuzdrený v TCP/IP a prenesený, napríklad, použitím ethernetu. Tento prístup umožňuje, aby mohli byť SCSI koncové body implementované softvérovo: namiesto špecializovaného hardvéru môže byť pole RAID diskov (krabica, ktorá obsahuje veľa diskov, ktoré sú skombinované do jednej alebo niekoľko málo logických diskov použitím RAID) implementované ako komoditný x86 server s ethernetovým pripojením. Je to postačujúce pre veľa prípadov použitia, a zároveň je to výrazne lacnejšie než 'natívne' SAN (siete určené pre úložiská – fibre channel, infiniband), alebo dokonca štandardný externe-pripojený SAS.

NVMe: Non-Volatile Memory Express

416

- pomerne jednoduchý protokol pre úložisko pripojené cez PCIe
- optimalizované pre zariadenia na báze SSD
  - výrazne väčšie a viac front príkazov než u AHCI
  - lepšia / rýchlejšia obsluha prerušení
- kladie dôraz na súbežnosť v blokovej vrstve kernelu

Technológia stelesňujúca 'návrat ku koreňom': to, čo bola ATA pre ISA, je NVMe pre PCIe. V podstate ide o protokol nad PCIe prepojením, ktorý využíva existujúce PCIe rozpoznávanie a konfiguráciu. Protokol vyžaduje pomerne rozsiahle fronty príkazov, využívajúc podobne rozsiahly paralelizmus u SSD hardvéru. NVMe úložiská sú obvykle veľmi rýchle a bloková vrstva, ktorá bola pôvodne navrhnutá pre výrazne pomalšie zariadenia, môže mať problém držať krok.

USB veľkokapacitné úložisko

417

- USB trieda zariadení (protokol nezávislý od dodávateľa)
  - jeden ovládač pre celú triedu
- typicky USB flash disky, ale tiež externé disky
- USB 2 nie je vhodné pre vysokorýchlostné ukladanie dát
  - USB 3 prinieslo UAS = USB-Attached SCSI

Ako bolo spomenuté skôr, úložné zariadenia môžu tiež byť priamo pripojené k USB.

Páskové jednotky

418

- na rozdiel od diskových jednotiek umožňujú iba sekvenčný prístup
- potrebujú podporu pre vysunutie média, spätné pretáčanie
- môžu byť pripojené pomocou SCSI, SATA, USB
- časti ovládača budú nezávislé na zbernici
- najmä pre zálohy dát, kapacity 6-15TB

Hoci sú diskové zariadenia (HDD, SSD, RAID polia) výrazne najdôležitejšie, existujú aj iné úložné zariadenia, ktoré stoja za zmienku. Dátové centrá často používajú páskové jednotky na zálohovanie, keďže

tieto poskytujú skvelú hustotu dát, nízku cenu za uložený gigabajt a dobrú odolnosť. Z pohľadu OS sú páskové jednotky špeciálne, keďže sa k dátam dá pristupovať iba sekvenčne, a nedáva zmysel na nich mať tradičný súborový systém. Namiesto toho sa používajú špecializované programy, ktoré pripravia dáta na zápis na pásku, napr. tar (skratka pre Tape ARchive - páskový archív).

Optické disky

419

- používané najmä ako distribučné médium výhradne na čítanie
- laserom vykonávané čítanie rotačného disku
- opäť môže byť pripojené k SCSI, SATA alebo USB
- vytvorené pre audio prehrávanie → pomalý presun hlavy

Ďalšou pomerne špeciálnou triedou úložných zariadení sú optické disky: CD-ROM, DVD-ROM, Blu-ray. Hoci je náhodný prístup možný, je veľmi pomalý aj v porovnaní s HDD. Optické disky sú vhodnejšie pre prehrávanie / streaming (najmä audia a videa) alebo distribúciu obsahu. Na rozdiel od pásiek, súborové systémy (výhradne na čítanie) sú na optických médiách bežne používané (ISO 9660 pre CD-ROM, UDF pre DVD a Blu-ray).

Zapisovače optických diskov (napaľovačky)

420

- správa sa v podstate ako tlačiareň pre optické disky
- ovládače sú často realizované v užívateľskom prostredí
- pripojené jednou zo štandardných diskových zberníc
- na napaľovanie diskov potrebné špeciálne programy
  - alternatíva: ovládače pre paketový zápis

Časť 7.5: Siete a bezdrôtová komunikácia

Poslednou kategóriou zariadení, o ktorých si v tejto prednáške povieme, sú sieťové karty. Zdôrazňujem, že ide iba o prehľad sieťového hardvéru, ktorý je možné pripojiť do univerzálneho (general-purpose) počítača – sieťami všeobecne sa budeme zaoberať v nasledujúcej prednáške.

Siete

422

- siete umožňujú, aby si viacero počítačov mohlo vymieňať dáta
  - môže ísť o súbory, prúdy alebo správy
- existujú drôtové (wired) a bezdrôtové (wireless) siete
- nateraz sa budeme zaoberať iba najnižšími vrstvami
- NIC = Network Interface Card – sieťová karta

Sieťový hardvér umožňuje počítačom navzájom priamo komunikovať, použitím určitej formy prepojenia, buď drôtovej alebo bezdrôtovej. Počítač sa pripája k sieti pomocou sieťovej karty, typicky ide o PCIe zariadenie s externým konektorom (napr. RJ 45 pre metalický ethernet), alebo anténa (pre bezdrôtové technológie). Počítačová sieť ako celok pripomína zbernicu, takého typu, aký sme si rozobrali v prvej časti prednášky, ale s určitými dôležitými rozdielmi.

Ethernet

423

- špecifikuje fyzické médium
- formát prenosu a riešenie kolízií
- u moderných zostáv obvykle point-to-point spoje
  - používajú aktívne zariadenia s prepínaním paketov
- prenášajú dáta v rámcoch (nízkoúrovňové pakety)

Podobne ako u systémových zberníc, siete sa vyvinuli zo zdieľaných médií (token ring, koaxiálny 10MiB ethernet, ethernet cez krútenú dvojlinku s pasívnymi rozbočovačmi – hubmi). Moderné siete používajú dedikované point-to-point spoje, s hardvérom zabezpečujúcim prepínanie paketov na miestach, kde sa stretáva niekoľko point-to-point spojov. Ethernetové 'pakety' sa nazývajú rámce a sú prenášané ako jedna jednotka. Každý rámec má nejaké metadáta (odosielateľ, príjemca, veľkosť) a samozrejme nesie nejaké dáta (payload).

Adresovanie

424

- na tejto úrovni iba lokálne adresovanie
  - najviac jeden LAN segment
- používa zabudované MAC adresy
  - MAC = Media Access Control
- adresy patria rozhraniam, nie počítačom

Adresovanie na najnižšej úrovni pracuje iba s jediným ethernetovým segmentom (broadcast doména). Všetky počítače poznajú MAC adresy všetkých ostatných počítačov, s ktorými sa chcú rozprávať (alebo skôr ich sieťových kariet). V starých sieťach so zdieľaným médiom bol rámec prenášaný na zdieľanom médiu a vydzvihnutý zamýšľaným príjemcom na základe cieľovej adresy. V sieti založenej na prepínaní paketov si prepínač (switch) udržiava mapovanie z MAC adries na fyzické porty, a preposiela rámce iba na port, na ktorý je pripojený zamýšľaný príjemca.

Odosielacia fronta – transmit queue

425

- pakety sú vyzdvihnuté z pamäte
- OS pripraví pakety do odosielacej fronty
- zariadenie si ich vyzdvihá asynchrónne
- podobne ako SATA pracuje s frontou príkazov a dát

Keď chce OS posielať pakety (rámce) cez sieť, sú vložené do odosielacej fronty (transmit queue – tiež známa ako Tx fronta), kde si ich vyzdvihne hardvér a prenesie ich cez svoje fyzické spojenie. Fronta funguje zhruba takto:

1. každá fronta (môže byť viac než jedna) má dvojicu registrov, prístupných cez MMIO, jeden pre ukazateľ na hlavu - head (fronty) a druhý pre ukazateľ na chvost - tail,
2. ukazatele si držia adresy do kruhového bufferu fixnej veľkosti, ktorý je uložený v hlavnej pamäti, a je k nemu pristupované cez DMA; každý prvok v kruhu je opäť ukazateľ, spolu s veľkosťou, a popisuje pamäťový buffer, ktorý si drží jeden rámec (paket),
3. ukazatele na hlavu a chvost rozdeľujú kruh na dve časti, tú ktorá patrí NIC a tú ktorá patrí softvéru,
4. operačný systém (skrze ovládač) ovláda ukazateľ na chvost v registri zariadenia:
   a. ak chce odoslať paket, vytvorí buffer a uloží dáta paketu do tohto buffera,
   b. vyplní prvú bunku v časti kruhu, ktorú riadi OS, adresou a veľkosťou tohto bufferu,
   c. posunie ukazateľ na chvost, a odovzdá novo-vyplnenú bunku

NIC,

5. sieťová karta riadi ukazateľ na hlavu: vždy keď spracuje paket, posunie ukazateľ na hlavu tak, aby spracovaný buffer bol v časti kruhu, ktorú riadi OS.

Ako sme si naznačili v prvej časti prednášky, udalosti súvisiace s prenosovým kruhom (transmit ring) môžu byť signalizované cez prerušenia.

Príjmová fronta – receive queue

426

- dáta sú zaradené do fronty aj opačným smerom
- NIC kopíruje pakety do príjmovej fronty
- vyvoláva prerušenie, aby oznámil OS, že prišli nové položky
  - NIC môže zlúčiť niekoľko paketov na každé prerušenie
- ak sa fronta rýchlo nevyprázdni → strata paketov

Príjmová (receive – Rx) fronta pracuje analogicky. Prerušenia signalizujú novo pridané prvky. OS má na starosti alokáciu bufferov pre pakety: odovzdanie bufferu NIC na Rx fronte znamená, že NIC môže slobodne prepísať tento buffer dátami paketu. Po tom, čo to urobí, je bunka Rx kruhu vrátená späť OS.
V bežnom režime musia byť všetky buffery rámcov (paketov) dostatočne veľké, aby dokázali uchovať najväčší možný rámec (známy ako MTU = Maximal Transfer Unit – maximálna jednotka prenosu), hoci aspoň niektoré NIC vedia rozdeliť prichádzajúce pakety cez niekoľko Rx buniek, ak sa nevojdú do jedného bufferu.
Ak sa Rx kruh zaplní, kým na rozhranie prichádzajú ďalšie pakety, pakety budú stratené (preto OS musí vyprázdňovať Rx kruh dostatočne rýchlo). Pakety nemusia byť spracované hneď: OS môže slobodne alokovať nové buffery a tieto vložiť do kruhu, namiesto opätovného využívania existujúcich bufferov. Zaplnené buffery môžu byť spracované a uvoľnené neskôr.

Viac-frontové adaptéry

427

- rýchle adaptéry môžu saturovať CPU
  - napr. 10GbE karty, alebo niekoľko-portové GbE
- tieto NIC dokážu spravovať niekoľko Rx a Tx front
  - každá fronta dostane vlastné prerušenie
  - rôzne fronty → môžu mať rôzne CPU jadrá

Súčasné sieťové adaptéry môžu posielať a prijímať pakety tak rýchlo, že jedno CPU jadro nestíha (keďže pre každý paket je treba typicky urobiť veľa práce ako postupuje cez sieťový zásobník a do užívateľského priestoru).
Tieto isté adaptéry môžu byť konfigurované, aby používali niekoľko Tx a Rx front (kruhov), každú s vlastnými registrami pre hlavu/chvost a prerušením. Je úlohou OS konfigurovať tieto fronty – typické usporiadanie používa jednu Tx/Rx dvojicu na CPU jadro.
Na prenos NIC jednoducho prelína pakety zo všetkých front, keďže o tom, ktorú frontu použiť na posielanie konkrétneho paketu, rozhoduje OS. Typicky jednoducho použije tú, ktorá je asociovaná s CPU jadrom, ktoré vykonáva danú operáciu.
Pre príjem je postup trochu komplikovanejší, keďže NIC sa musí rozhodnúť, ktorú frontu použiť. NIC môže byť nakonfigurované, aby filtrovalo alebo hashovalo prichádzajúce pakety (alebo ich časti) a použiť Rx frontu v závislosti od výsledku. Cieľom je, aby príbuzné pakety zostávali v rovnakej fronte (zlepšuje lokalitu), ale tiež, aby sme vyťažovali všetky fronty (zlepšuje rovnomernosť rozdelenia záťaže).

Offloading (odľahčovanie) kontrolných súčtov a TCP

428

- pokročilejšie adaptéry dokážu odľahčiť niektoré funkcie
- napr. výpočet povinných kontrolných súčtov paketu
- ale tiež funkcie súvisiace s TCP
- potrebuje podporu od ovládača aj od TCP/IP zásobníka

Pre urýchlenie spracovania paketov môžu byť niektoré úlohy, ktoré treba vykonávať pre každý paket realizované hardvérovo. Najčastejšou úlohou, ktorú vykonáva hardvér je výpočet a overovanie kontrolných súčtov: hlavičky paketov často obsahujú kontrolný súčet, aby detegovali poškodenie dát. Tieto kontrolné súčty sa obvykle dajú vypočítať veľmi rýchlo v hardvéri a ide o mrhanie CPU cyklami robiť to softvérovo. Takže keď je paket uložený v Tx fronte, polia pre kontrolné súčty sú ponechané prázdne a hardvér ich vyplní pred odoslaním paketu (toto platí pre kontrolné súčty protokolov vyšších úrovní, napr. TCP; kontrolné súčty ethernetových rámcov sú vždy spočítané hardvérovo).
Hoci ide o najjednoduchšiu úlohu, offloading kontrolných súčtov nie je jedinou úlohou, ktorá môže byť vykonávaná v hardvéri; niektoré ďalšie zahŕňajú:

- offloading kryptografie (IPsec): autentizačné hlavičky, šifrovanie a dešifrovanie prenášaných dát (payload),
- zlučovanie veľkých odosielaných, prijímaných segmentov: segmentácia a spätné poskladanie veľkých TCP paketov (t.j. takých, ktoré sa nevojdú do jedného IP paketu),
- UDP segmentácia (rozdeľovanie UDP paketov, ktoré sa nevojdú do MTU sieťovej karty).

WiFi

429

- bezdrôtové sieťové rozhranie – 'bezdrôtový ethernet'
- zdieľané médium – elektromagnetické vlny vo vzduchu
- (takmer) povinné šifrovanie
  - inak je jednoduché odpočúvať alebo dokonca aktívne útočiť
- veľmi zložitý protokol (v porovnaní s bežným hardvérom)
  - podporovaný firmvérom, ktorý beží na adaptéri

V porovnaní s relatívnou jednoduchosťou drôtových sietí je WiFi extrémne komplikovaná, kvôli povahe svojho média, ktoré je zdieľané, rušné, ľahko odpočúvateľné a všeobecne nespoľahlivé. Zariadenia, ktoré sa pripájajú k WiFi sieťam, sú často prenosné a musia si zachovávať konektivitu počas toho, ako sa premiestňujú a prepínajú medzi rôznymi prístupovými bodmi alebo dokonca sieťami.
Kvôli všadeprítomnému šifrovaniu sa musia klienti a prístupové body navzájom autentizovať a ustanoviť dvojicu kľúčov pre danú reláciu (session). Autentizácia je nevyhnutná, pretože inak by mohol aktívny útočník oklamať klienta a nechať ho pripojiť sa k svojmu zariadeniu a tým sa stať tzv. 'man in the middle', čím robí šifrovanie neúčinným. Keďže je šifrovanie aj tak potrebné, je používané aj ako prostriedok kontroly prístupu.
Aspekty protokolov súvisiacich s WiFi sú implementované v hardvéri, firmvéri (softvér bežiaci na adaptéri) a v softvéri (bežiacom na hlavnom CPU).

Review Questions

430

25. What is memory-mapped IO and DMA?
26. What is a system bus?
27. What is a graphics accelerator?
28. What is a NIC receive queue?

Časť 8: Sieťová vrstva

V tejto prednáške sa pozrieme na sieťové služby z pohľadu operačného systému. Zameriame sa najmä na sieťovú vrstvu (network stack): teda TCP/IP a súvisiace protokoly a preklad hostiteľského mena na IP adresu (host name resolution). Tiež sa pozrieme na sieťové súborové systémy (t.j. súborové systémy, ktoré sú uložené jedným počítačom na sieti, ale môžu byť používané ďalšími počítačmi na rovnakej sieti).

Obsah prednášky

1. Úvod do sietí
2. TCP/IP vrstva
3. Používanie sietí
4. Sieťové súborové systémy

Najprv si všeobecne v rýchlosti zopakujeme terminológiu sietí a základných konceptov. Potom sa bližšie pozrieme na TCP/IP vrstvu a ako táto zapadá do všeobecných konceptov, ktoré sme si už uviedli. Ďalšia časť prednášky bude zameraná na sieťové rozhrania pre programovanie aplikácií (API). Nakoniec sa pozrieme na zdieľanie súborových systémov v sieťovom prostredí.

Časť 8.1: Úvod do sietí

V tejto sekcii sa budeme zaoberať najmä už známymi sieťovými konceptmi, aby sme mali neskôr dosť kontextu, keď sa pustíme do väčších detailov a do špecifík na úrovni OS.

Hostiteľské a doménové mená

- hostiteľské meno/hostname = ľudsky čitateľný názov počítača
- hierarchický systém, little endian: `www.fi.muni.cz`
- FQDN = fully-qualified domain name - plne kvalifikované doménové meno
- lokálny sufix môže byť vynechaný (`ping aisa`)

Prvá vec, ktorej potrebujeme porozumieť je, ako identifikovať počítače v rámci siete. Hlavný spôsob ako to robiť je pomocou tzv. hostnames - hostiteľských mien: ľudsky čitateľných mien, ktoré existujú v dvoch variantách: samotný názov počítača, a plne kvalifikovaný názov, ktorý zahŕňa názov siete, do ktorej je počítač pripojený.

Sieťové adresy

- adresa = vhodné pre počítače a numerické
- IPv4 adresa: 4 oktety (bajty): `192.168.1.1`
  - oktety sú usporiadané MSB-first - prvý najvýznamnejší bajt (big endian)
- IPv6 adresa: 16 oktetov
- Ethernet (MAC): 6 oktetov, `c8:5b:76:bd:6e:0b`

Zatiaľ čo ľudia uprednostňujú odkazovať na počítače pomocou ľudsky-čitateľných názvov, tieto nie sú vhodné pre reálnu komunikáciu. Namiesto toho, keď počítače potrebujú odkazovať na iné počítače, používajú numerické adresy (rovnako ako pri pamäťových miestach alebo diskových sektoroch). Veľkosť a štruktúra adresy sa môže líšiť v závislosti od protokolu: tradičné IPv4 používa 4 oktety, zatiaľ čo adresy v novom IPv6 ich využívajú 16 (128 bitov). Ďalší typ adresy, ktorý môžete často stretnúť, je MAC (od media access control - kontrola prístupu k médiu), ktorá je najznámejšia z Ethernetového protokolu.

Typy sietí

- LAN = Local Area Network - lokálne
  - Ethernet: drôtová, až do 10Gb/s
  - WiFi (802.11): bezdrôtová/wireless, až do 1Gb/s
- WAN = Wide Area Network (Internet) - globálne
  - PSTN, xDSL, PPPoE
  - GSM, 2G (GPRS, EDGE), 3G (UMTS), 4G (LTE)
  - tiež LAN technológie – Ethernet, WiFi

Siete sa všeobecne delia na dva typy: lokálne, ktorých rozsah je v rámci kancelárie, domácnosti, prípadne budovy. LAN zvyčajne tvoria jednu broadcast doménu, čo zhruba znamená, že každý počítač na sieti môže priamo kontaktovať akýkoľvek iný počítač pripojený k rovnakej LAN. Najčastejšie technológie (vrstvy 1 a 2) použité v LAN sú drôtový (wired) ethernet (najčastejší typ prenáša 1Gb/s, menej časté, ale stále používané sú verzie s rýchlosťou 10Gb/s) a bezdrôtové (wireless) WiFi (pôvodne známe ako IEEE 802.11).
Na druhú stranu, globálne siete (tzv. wide-area networks, WAN) siahajú cez značné vzdialenosti a spájajú veľké množstvo počítačov. Kanonickou WAN je internet, alebo sieť internetového poskytovateľa (ISP - internet service provider). Siete WAN často používajú iné typy nízkoúrovňových technológií.

Sieťové vrstvy

1. Vrstva sieťového rozhrania - Linková (Ethernet, WiFi)
2. Internetová / Sieťová (IP)
3. Transportná (TCP, UDP, ...)
4. Aplikačná (HTTP, SMTP, ...)

Štandardný model sieťových protokolov (známy ako Open Systems Interconnection, v skratke OSI) rozdeľuje sieťovú úroveň OS na 7 vrstiev, ale pohľad sieťového modelu TCP/IP ich často rozlišuje iba 4, ktoré sme si načrtli vyššie. Vrstva sieťového rozhrania (tiež linková vrstva) zhruba zodpovedá OSI vrstvám 1 (fyzická) a 2 (datová), internetová vrstva je OSI vrstva 3, transportná vrstva zodpovedá OSI vrstve 4 a zvyšné (OSI vrstvy 5 až 7) sú sústredené pod aplikačnú vrstvu.
Budeme sa držať zjednodušeného TCP/IP modelu, ale keď budeme odkazovať na nejakú vrstvu číslom, bude sa jednať o OSI čísla, ako je zvykom (konkrétne, IP je vrstva 3 a TCP je vrstva 4).

Siete a operačné systémy

- sieťová vrstva (network stack) je štandardnou súčasťou OS
- veľká časť tejto vrstvy žije v kerneli
  - týka sa to však iba monolitických kernelov
  - mikrokernely používajú sieťové služby z užívateľského priestoru
- ďalší kus je v systémových knižniciach & pomocných programoch

Posledných asi 25 rokov je sieťová komunikácia štandardnou službou poskytovanou všeobecnými (general-purpose) operačnými systémami. U systémov s monolitickým kernelom je veľká časť sieťovej vrstvy (všetko po a vrátane transportnej vrstvy) súčasťou kernelu a je sprístupnená užívateľským programom cez API soketov.
Ďalšia funkcionalita na úrovni aplikačnej vrstvy je zvyčajne dostupná v systémových knižniciach: najmä preklad doménových mien na IP adresy (domain name resolution - DNS) a šifrovanie - encryption (TLS,

skratka od transport-layer security - bezpečnosť na úrovni transportnej vrstvy; pomerne mätúco sa jedná o technológiu na úrovni aplikačnej vrstvy).

Sieťová komunikácia na strane kernelu

439

- ovládače zariadení pre sieťový hardvér
- vrstvy sieťových a transportných protokolov
- smerovanie - routing, a filtrovanie paketov (firewally)
- sieťové systémové volania (sokety)
- sieťové súborové systémy (SMB, NFS)

Vrstva sieťového rozhrania (linková vrstva) je zvyčajne zabezpečená ovládačmi zariadení a klientská a serverová strana TCP/IP sú prístupné cez soketové API. TCP/IP siete však obsahujú ďalšie komponenty: niektoré, ako smerovanie a filtrovanie paketov, možno často realizovať softvérovo, a ak je to ten prípad, tak sú zvyčajne implementované v kerneli. Mosty (bridge) a prepínače (switch), ktoré patria do vrstvy sieťového rozhrania, tiež môžu byť implementované softvérovo, ale je to zriedka praktické. Veľa operačných systémov však implementuje jeden alebo oba pre lepšiu podporu virtualizácie.
Niektoré sieťové služby aplikačnej vrstvy môžu tiež byť implementované v kerneli, obzvlášť sieťové súborové systémy, ale niekedy aj ďalšie protokoly (napr. HTTP akcelerátory na úrovni kernelu).

Systémové knižnice

440

- soketové a príbuzné API
- preklad hostiteľského mena (DNS klient)
- šifrovanie a autentifikácia dát (SSL, TLS)
- správa a validácia certifikátov

Prísne vzaté je soketové API doménou systémových knižníc (u väčšiny monolitických kernelov sa mapujú C-čkové funkcie 1:1 na systémové volania; avšak v mikrokerneloch je sieťová vrstva rozdelená inak a systémové knižnice pravdepodobne zastávajú väčšiu časť úloh).
Keďže takmer všetky programy, ktoré pracujú so sieťami, potrebujú prekladať hostiteľské mená (prekladať ľudsky čitateľné mená na IP adresy), túto službu obvykle poskytujú systémové knižnice. Podobne, šifrovanie je pomerne rozšírené v modernom internete, a väčšina operačných systémov dodáva vrstvu SSL/TLS, vrátane správy certifikátov.

Pomocné programy a systémové služby

441

- konfigurácia siete (`ifconfig`, `dhclient`, `dhcpd`)
- správa smerovania - route (`route`, `bgpd`)
- diagnostika (`ping`, `traceroute`)
- logovanie a kontrola paketov (`tcpdump`)
- ďalšie sieťové služby (`ntpd`, `sshd`, `inetd`)

Posledná komponenta sieťovej vrstvy sa nachádza v systémových pomocných programoch a službách (démonoch). Tieto sa zaoberajú konfiguráciou (vrátane prideľovania adries rozhraniam a autokonfigurácie, napr. DHCP alebo SLAAC) a správou smerovania (dôležité najmä u softvérových smerovačov / routerov a u multihome systémov - viacnásobné pripojenie k sieti).
Zvyčajne je tiež prítomná sada diagnostických nástrojov, prinajmenšom programy `ping` a `traceroute`, ktoré sú užitočné na kontrolu konektivity, prípadne nástroje ako `tcpdump`, ktoré umožňujú obsluhe preskúmať pakety prichádzajúce na rozhranie.

Aspekty sieťovej komunikácie

442

- vo formáte paketov
  - čo je jednotkou komunikácie
- adresovanie
  - ako sa volajú odosielateľ a príjemca
- doručovanie paketov
  - ako sú správy doručené

Keď sa dívame na sieťový protokol, musíme zvážiť tri hlavné aspekty: prvým je, čo predstavuje jednotku komunikácie, t.j. ako pakety vyzerajú, aké informácie prenášajú, atď. Druhým je adresovanie: ako sú vymedzené (určené) cieľové počítače a/alebo programy. Nakoniec sa doručovanie paketov zaoberá tým, ako sú správy doručované z jednej adresy na druhú: toto môže zahŕňať smerovanie a/alebo preklad adries (napr. medzi adresami linkovej vrstvy a IP adresami).

Vnorovanie (nesting) protokolov

443

- protokoly bežia nad ďalšími protokolmi
- preto sa to anglicky volá network stack - zásobník
- vyššie vrstvy využívajú nižšie vrstvy
  - HTTP používa abstrakcie, ktoré poskytuje TCP
  - TCP používa abstrakcie, ktoré poskytuje IP

Keďže hovoríme o tzv. protocol stack - „zásobníku“ protokolov (protokolových vrstvách), je dôležité porozumieť, ako jednotlivé vrstvy tohto zásobníka navzájom interagujú. Každý z vyššie uvedených aspektov interaguje so zásobníkovou štruktúrou sieťovej vrstvy trochu inak – všetky si bližšie vysvetlíme na niekoľkých nasledujúcich slidoch.

Vnorovanie (nesting) paketov

444

- pakety vyšších vrstiev sú pre nižšie úrovne dáta
- Ethernetový rámec (frame) môže vnútri prenášať IP paket
- IP paket môže niesť TCP paket
- TCP paket môže niesť HTTP požiadavok (jeho fragment)

Keď uvážime štruktúru paketu, je prirodzenejšie začať na spodných vrstvách: pakety vyšších vrstiev sú pre nižšie vrstvy obyčajné dáta. Celková štruktúra paketov vyzerá ako matrioška: ethernetový rámec je obalený okolo IP paketu, ktorý je obalený okolo UDP paketu, a tak ďalej.
Z pohľadu vyšších vrstiev je veľkosť paketu dôležitým hľadiskom: keď sú paketovo-orientované protokoly vnorené do ďalších paketovo-orientovaných protokolov, je užitočné, keď ich veľkosti paketov odpovedajú (väčšina protokolov má limit na veľkosť paketu). Keď vezmeme do úvahy limity, pri pohľade ‘zhora’ je paket posunutý nižšej vrstve ako dáta a vyššia vrstva nevie nič o ďalších rámcoch (hlavičkách), ktoré nižšia vrstva pridáva.

Vrstvy vs doručovanie

445

- doručovanie je, teoreticky, point-to-point
  - smerovanie je väčšinou skryté pred vyššími vrstvami
  - vyššia vrstva si vyžiada doručenie na adresu
- protokoly nižších vrstiev sú obvykle paketovo orientované
  - rozdiely vo veľkosti paketov môžu spôsobiť fragmentáciu
- paket môže prechádzať cez rôzne nízkoúrovňové domény

Pokiaľ ide o doručovanie, vzťahy medzi jednotlivými vrstvami sú asi najkomplikovanejšie. V tomto prípade je pohľad zhora nadol asi najvhodnejší, keďže nižšie vrstvy poskytujú doručovanie ako službu vyšším vrstvám.
Keďže doručovanie má na internetovej vrstve (OSI vrstvy 3 a vyššie) obvykle podstatne väčší záber (oblasť, v ktorej funguje), než na linkovej vrstve, často sa stáva, že jeden IP paket prechádza niekoľkými doménami (segmentmi, lokálnymi sieťami) linkovej vrstvy.

Vrstvy vs adresovanie

446

- nie je to aŽ také priamočiare ako vnorovanie paketov
  - vzťahy medzi adresami sú zložité
- existujú špeciálne protokoly na preklad adries
  - DNS pre mapovanie medzi hostiteľskými menami vs IP adresami
  - ARP pre mapovanie IP vs MAC adresy

Konečne, keďže doručovanie (paketov, dát) je služba, ktorú poskytujú nižšie vrstvy vyšším, vyššie vrstvy musia porozumieť a poskytnúť správne adresy nižších úrovní. Najjednoduchší spôsob, akým môžeme nahliadať na tento aspekt, je po dvojiciach: linková vrstva a internetová vrstva zjavne musia interagovať, obvykle cez špeciálny protokol, ktorý beží na linkovej vrstve, ale logicky patrí do internetovej vrstvy, keďže pracuje s IP adresami.
Situácia medzi internetovou a transportnou vrstvou je výrazne jednoduchšia: adresa na transportnej vrstve jednoducho obsahuje adresu internetovej vrstvy ako pole (napr. TCP adresa je IP adresa + číslo portu). Vzťah medzi aplikačnou vrstvou a transportnou vrstvou je analogický (ale nie úplne rovnaký) situácii internetová/linková vrstva. Aplikačná vrstva na identifikáciu počítačov primárne používa hostiteľské mená, a používa špeciálny protokol, známy ako DNS, ktorý síce pracuje s adresami na úrovni transportnej vrstvy, ale inak patrí do aplikačnej vrstvy.

ARP (Address Resolution Protocol)

447

- nájde MAC adresu, ktorá odpovedá IP
- nevyhnutný pre umožnenie doručovania paketov
  - IP používa linkovú vrstvu na doručenie paketov
  - linková vrstva musí dostať MAC adresu
- OS vytvorí mapu s prekladmi IP $\rightarrow$ MAC

Protokol rozlišovania adries (address resolution protocol), ktorý sa nachádza na hranici linkovej/internetovej vrstvy, umožňuje internetovej vrstve doručovať svoje pakety použitím služieb linkovej vrstvy. Samozrejme, na vyžiadanie doručenia paketu na úrovni linkovej vrstvy potrebujeme linkovú adresu, ale IP paket obsahuje iba IP adresu. ARP protokol sa používa na nájdenie linkovej adresy IP uzlov, ktoré existujú na lokálnej sieti (čo zahŕňa smerovače, ktoré pracujú na internetovej vrstve – inými slovami, pakety, ktoré smerujú mimo lokálnu sieť sú poslané na router, použitím IP adresy routeru, ktorá sa prekladá na adresu na úrovni linkovej vrstvy, použitím ARP ako obvykle).

Ethernet

448

- komunikačný protokol na úrovni linkovej vrstvy
- z veľkej časti implementovaný hardvérovo
- OS používa dobre definované rozhranie
  - prijímanie a posielanie paketov
  - použitím MAC adries (ARP je súčasťou OS)

Asi najbežnejším protokolom linkovej vrstvy je ethernet. Väčšina protokolu je implementovaná priamo v hardvéri a operačný systém iba používa zjednotené rozhranie, ktoré ovládače zariadení sprístupňujú na posielanie a prijímanie ethernetových rámcov.

Prepínanie paketov

449

- zdieľané médiá sú neefektívne kvôli kolíziám
- ethernet typicky používa prepínanie paketov
  - prepínač (switch) je obvykle hardvérové zariadenie
  - ale môže byť aj softvérové (obvykle pre virtualizáciu)
  - fyzické pripojenia tvoria topológiu hviezda

Vysokorýchlostné siete sú založené takmer výhradne na prepínaní paketov, to znamená, že uzol posiela pakety (rámce) na prepínač (switch), ktorý má niekoľko fyzických portov a udržiava si informácie ktoré MAC adresy sú dostupné na ktorých fyzických portoch. Keď na prepínač dorazí rámec, prepínač si vyextrahuje MAC adresu príjemcu a paket je preposlaný na fyzický(é) port(y), ktoré sú asociované s touto MAC adresou.

Mosty (bridges)

450

- mosty pracujú na linkovej vrstve (vrstva 2)
- most je dvojportové zariadenie
  - každý port je pripojený do inej LAN
  - most spája LAN preposielaním rámcov
- môže byť realizovaný hardvérovo alebo softvérovo
  - `brctl` na Linuxe, `ifconfig` na OpenBSD

Mosty sú analogické prepínačom, s jedným veľkým rozdielom: predpoklad pre prepínače je, že existuje veľa fyzických portov, ale s každým je asociovaná iba jedna MAC adresa (prípadne s výnimkou špeciálneho 'uplink' portu). Na druhú stranu, most je optimalizovaný na prípad dvoch fyzických portov, ale každá strana so sebou má asociovaných veľa MAC adries.

Tunelovanie

451

- tunely sú virtuálne zariadenia na vrstve 2 alebo 3
- zapuzdrujú premávku použitím protokolu vyššej úrovne
- tunelovanie môže implementovať Virtual Private Networks
  - softvérový most môže fungovať cez UDP tunel
  - tunel je obvykle šifrovaný

Tunelovanie je technika, ktorá umožňuje premávke na nižšej vrstve, aby bola vložená do aplikačnej vrstvy existujúcej siete. Typickým prípadom použitia je spojiť fyzicky vzdialené počítače do jednej broadcast (linková vrstva) alebo smerovacej (internetová vrstva) domény.
V tomto prípade sú prítomné dve inštancie sieťového „zásobníka": VPN softvér implementuje protokol na úrovni aplikačnej vrstvy vo vonkajšom zásobníku, a zároveň sa chová ako rozhranie na linkovej vrstve (alebo podsieť na internetovej vrstve), ktoré je preposlané cez most (routované) ako keby šlo o ďalšie fyzické rozhranie.

PPP (Point-to-Point protokol)

452

- protokol na linkovej vrstve pre 2-uzlové siete
- dostupné cez veľa fyzických spojení
  - telefónne linky, mobilné pripojenia, DSL, Ethernet
  - často používané na pripájanie koncových bodov k ISP
- podporované väčšinou operačných systémov
  - rozdelené medzi kernelom a systémovými pomocnými programami

Point-to-point protokol je ďalším z pomerne dôležitých a rozšírených príkladov protokolu linkovej vrstvy a väčšinou ho môžeme nájsť na pripojeniach medzi LAN sieťami alebo medzi LAN a WAN.

Bezdrôtové siete

453

- WiFi je z veľkej časti ako (pomalý, nespoľahlivý) Ethernet
- potrebuje šifrovanie, keďže ktokoľvek môže počúvať
- tiež autentizáciu na predchádzanie podvodným pripojeniam
  - PSK (pre-shared key - vopred zdieľaný kľúč), EAP / 802.11x
- šifrovanie vyžaduje správu kľúčov

Konečne, WiFi je, z pohľadu zvyšku sieťového zásobníka, v podstate pomalá, nespoľahlivá verzia ethernetu, hoci vnútri je tento protokol výrazne komplikovanejší.

Časť 8.2: TCP/IP vrstvy („zásobník“)

V tejto sekcii sa pozrieme na samotný TCP/IP sieťový zásobník a tiež si bližšie rozoberieme DNS.

IP (Internet Protocol)

455

- používa 4-bajtové (v4) alebo 16-bajtové (v6) adresy
  - rozdelené na časti sieť (network) a hostiteľ (host)
- ide o protokol založený na paketoch
- poskytuje tzv. best-effort službu
  - pakety sa môžu stratiť, byť preusporiadané alebo poškodené

IP je protokol s malou réžiou, paketovo-orientovaný, veľmi rozšírený v rámci internetu a väčšiny lokálnych sietí (či už sú pripojené k internetu alebo nie). Čo je pomerne dôležité, kvôli nízkej réžii nezaručuje doručenie ani integritu dát, ktoré prenáša.

IP siete

456

- IP siete zhruba odpovedajú LAN
  - hostitelia na rovnakej sieti sú lokalizovaní pomocou ARP
  - k vzdialeným sieťam možno pristupovať skrze smerovače
- maska siete rozdeľuje adresu na časti sieť/host
- IP typicky beží nad Ethernetom alebo PPP

V rámci jednej IP siete je doručovanie zabezpečované linkovou vrstvou – pričom lokálna sieť je identifikovaná spoločným adresovým prefixom (dĺžka tohto prefixu je súčasťou konfigurácie siete, a je známa pod názvom maska siete – netmask).

Smerovanie (routing)

457

- smerovače preposielajú pakety medzi sieťami
- niečo na spôsob mostov ale na 3. vrstve
- smerovače sa chovajú ako normálne koncové body LAN
  - ale reprezentujú celé vzdialené IP siete
  - alebo dokonca celý internet

Pakety, ktorých príjemcovia sú mimo lokálnej siete (teda takí, ktorí nezdieľajú sieťovú časť adresy s lokálnym hostiteľom) sú presmerované - routed: zariadenie na vrstve 3, analogické prepínaču (switch) na vrstve 2, prepošle paket na jedno zo svojich rozhraní (do inej linkovej domény). Smerovacie tabuľky (routing tables) sú však výrazne zložitejšie, než informácie, ktoré si uchováva prepínač, a ich správa cez internet je nad rámec tohto predmetu.

ICMP: Internet Control Message Protocol

458

- riadiace správy (pakety)
  - cieľový host/sieť nedosiahnuteľná
  - prekročený limit time to live
  - potrebná fragmentácia
- diagnostické pakety, napr. príkaz `ping`
  - `echo request` a `echo reply`
  - dá sa skombinovať s TTL pre `traceroute`

ICMP je ‘servisný protokol’, ktorý sa používa na diagnostiku, chybové hlásenia a správu siete. Rola ICMP bola značne rozšírená s príchodom IPv6 (napr. zahrnutím automatickej konfigurácie siete, cez typy paketov router advertisement - oznámenie routeru a router solicitation - vyžiadanie si routeru). ICMP neposkytuje aplikačnej vrstve žiadne služby priamo.

Služby a TCP/UDP čísla portov

459

- siete sa obvykle používajú na poskytovanie služieb
  - každý počítač môže hosťovať viacero
- rôzne služby môžu bežať na rôznych portoch
- port je 16-bitové číslo a niektoré sú pomenované
  - port 25 je SMTP, port 80 je HTTP, ...

Ako sme už stručne spomenuli, adresy transportnej vrstvy majú dve komponenty: IP adresa cieľového počítača a číslo portu, ktoré vymedzuje konkrétnu službu alebo aplikáciu bežiacu na cieľovom uzle.

TCP: Transmission Control Protocol

460

- prúdovo-orientovaný protokol nad IP
- funguje ako rúra (prenáša sekvenciu bajtov)
  - musí rešpektovať poradie doručenia
  - a tiež preposlať stratené pakety
- musí ustanoviť spojenia

Dva hlavné transportné protokoly v rodine protokolov TCP/IP sú TCP a UDP, pričom prvé sa vyskytuje častejšie a je výrazne komplikovanejšie. Keďže TCP je prúdovo-orientované a spoľahlivé, musí implementovať logiku rozkúskovania prúdu bajtov na jednotlivé pakety (pre doručenie používajúce IP, ktoré je paketovo-orientované), kontroly konzistencie (kontrolné súčty paketov) a logiku znovu-posielania (v prípade, že sa IP pakety, ktoré nesú TCP dáta stratia).

TCP spojenia

461

- koncové body najprv musia ustanoviť spojenie
- každé spojenie slúži ako samostatný prúd dát
- spojenie je obojsmerné
- TCP používa 3-cestný handshake: SYN, SYN/ACK, ACK

Na poskytnutie prúdovej sémantiky užívateľovi musí TCP implementovať mechanizmus, ktorý vytvorí ilúziu prúdu bajtov nad paketovo-fungujúcim základom. Tento mechanizmus je známy ako spojenie a v podstate sa skladá z nejakého stavu, ktorý je zdieľaný medzi dvomi koncovými bodmi. Na ustanovenie tohto zdieľaného stavu TCP používa 3-cestný handshake („potrasenie rukou“).

Poradové čísla

462

- TCP pakety si nesú poradové čísla
- tieto čísla sa používajú na rekonštrukciu prúdu
  - IP pakety môžu prísť mimo poradie
- tiež sa používajú na potvrdenie prijatia
  - a následne na správu opätovného posielania

Poradové čísla sú súčasťou stavu spojenia a umožňujú prúdu bajtov, aby mohol byť rekonštruovaný v správnom poradí, aj keď dôjde k preskladaniu IP paketov, ktoré nesú prúd, počas doručovania.

Strata paketov a opätovné posielanie

463

- pakety sa môžu stratiť z rôznych dôvodov
  - linka spadne na dlhšiu dobu
  - pretečie buffer smerovacieho zariadenia
- TCP posiela potvrdenia (ack) pre prijaté pakety
  - ACK používajú na identifikáciu paketov poradové čísla

Okrem preusporiadania paketov musí TCP vyriešiť ešte stratu paketov: udalosť, kedy je IP paket odoslaný, ale zmizne bez stopy po ceste do svojej destinácie. Stratený paket je detekovaný ako medzera v poradových číslach. Je to však odosielateľ, kto sa musí dozvedieť o stratenom pakete, aby tento mohol byť preposlaný: z tohto dôvodu musí príjemca paketu potvrdiť (acknowledge) jeho prijatie poslaním paketu nazad (alebo, častejšie, tým že sa potvrdzovací paket „zvezie“ na dátovom pakete, ktorý by aj tak odoslal - tzv. piggybacking), ktorý nesie poradové čísla paketov, ktoré boli prijaté.
Ak potvrdenie prijatia nie je obdržané do určitého časového limitu (ktorý sa dá dynamicky prispôsobiť) od odoslania pôvodného paketu, tento paket je poslaný znova (retransmitted).

UDP: User (Unreliable) Datagram Protocol

464

- TCP so sebou nesie netriviálnu réžiu
  - a jeho záruky nie sú vždy nevyhnutné
- UDP je výrazne jednoduchší protokol
  - veľmi tenký wrapper okolo IP
  - s minimálnou réžiou nad rámec IP

Nie všetky aplikácie potrebujú relatívne silné záruky, ktoré poskytuje TCP, alebo opačne, nemôžu tolerovať dodatočnú latenciu, ktorú so sebou prináša algoritmus, ktorý TCP využíva na zaručenie spoľahlivého doručenia zachovávajúceho poradie. Pre tieto prípady poskytuje UDP veľmi tenkú vrstvu nad IP, v podstate pridáva iba číslo portu do adries a 16-bitový kontrolný súčet do hlavičky paketu (ktorej veľkosť je celkovo iba 64 bitov).

Firewall

465

- názov pochádza zo stavby budov
  - ohňuvzdorná bariéra medzi časťami budovy
- myšlienka je oddeliť siete od seba navzájom
  - čím sú útoky zvonku ťažšie
  - obmedziť škody v prípade kompromisu

Firewall je zariadenie, ktoré oddeľuje dve siete od seba navzájom, typicky tým, že slúži ako (jediný) router medzi nimi, ale tiež skúmaním paketov a ich zahadzovaním alebo odmietaním, ak vyzerajú nedôveryhodne, alebo ak sa pokúšajú používať služby, ktoré nemajú byť viditeľné zvonku. Často je jednou z týchto sietí internet. Niekedy je druhou sieťou iba jediný počítač.

Filtrovanie paketov

466

- filtrovanie paketov je implementáciou firewallu
- môže byť realizované na routeri alebo na koncovom bode
- dedikované routery + filtrovanie paketov je bezpečnejšie
  - jediný takýto firewall chráni celú sieť
  - menej príležitostí na chybnú konfiguráciu

Podobne ako u ostatných služieb, obvykle sa vyplatí centralizovať (v rámci jednej siete) zodpovednosť za filtrovanie paketov, čím sa zníži administratívna záťaž a priestor, aby chybne konfigurované uzly ohrozili celú sieť. Je samozrejme rozumné spúšťať lokálne firewally na každom uzle, ako druhú úroveň ochrany.

Prevádzka paketových filtrov

467

- filtre paketov pracujú so sadou pravidiel
  - pravidlá sú obvykle dodávané obsluhou
- každý prichádzajúci paket je roztriedený použitím pravidiel
- a potom podľa toho spracovaný ďalej
  - môže byť preposlaný, zahodený, odmietnutý alebo upravený

Filter paketov je v podstate konečne-stavový stroj (prípadne s trochou pamäte na sledovanie pripojení, v takom prípade ide o stavový filter paketov), ktorý skúma každý paket a rozhodne sa, akú akciu na ňom vykonať. Konkrétne pravidlá klasifikácie sú obvykle dodávané administrátorom siete; v jednoduchých prípadoch sa porovnávajú zdrojové a cieľové IP adresy a čísla portov, a stav spojenia (ktorý si filter paketov pamätá) u TCP paketov.
Po tom, čo sú roztriedené, pakety môžu byť preposlané do svojej destinácie (ako by to urobil štandardný router), potichu zahodené (dropped), odmietnuté (rejected; čo pošle ICMP notifikáciu odosielateľovi) alebo upravené pred tým, než budú poslané ďalej (najčastejšie kvôli NAT - network address translation, ktorého detaily idú mimo rozsah predmetu).

Príklady filtrovania paketov

468

- filtre paketov sú často súčasťou kernelu
- parser pravidiel je systémová utilita
  - načítava pravidlá z konfiguračného súboru
  - a nastaví filter na strane kernelu
- existuje viacero implementácií
  - `iptables`, `nftables` na Linuxe
  - `pf` na OpenBSD, `ipfw` na FreeBSD

Filter paketov je obvykle tvorený dvomi komponentami: jednou je systémová utilita (pomocný program), ktorý číta ľudsky-čitateľný popis pravidiel, a na základe týchto vyskladá efektívny „porovnávač“ pravidiel, ktorý bude použitý v kernelovej komponente, ktorá robí samotnú klasifikáciu.

Preklad mien (name resolution)

469

- užívatelia si nechcú pamätať numerické adresy
  - už telefónne čísla sú dosť zlé
- namiesto toho sa používajú hostiteľské mená (host names)
- môžu byť uložené v súbore, napr. `/etc/hosts`
  - nie je to úplne praktické pre viac ako 3 počítače
  - ale na internete sú milióny počítačov

V poslednej časti tejto sekcie sa pozrieme na preklad hostiteľských mien (host name resolution) a na protokol DNS. Čo potrebujeme je adresár (na spôsob zlatých stránok), ale taký, ktorý môže byť efektívne aktualizovaný (každú hodinu prebieha veľa aktualizácií) a ktorý tiež zvládne efektívne odpovedať na požiadavky počítačov na sieti. Systém musí byť dostatočne škálovateľný, aby zvládol veľa miliónov mien.

DNS: Domain Name System

470

- hierarchický protokol pre preklad mien
  - beží nad TCP alebo UDP
- doménové mená sú rozdelené do niekoľkých častí pomocou bodiek
  - každá doména vie, koho sa opýtať na ďalšiu časť
  - databáza mien je distribuovaná

Na škále internetu v podstate potrebujeme nejakú formu distribuovaného systému (t.j. distribuovanú databázu). Na rozdiel od relačných databáz sú však oneskorenia v propagácii aktualizácií akceptovateľné, čím sa návrh zjednodušuje.
Menný priestor hostiteľských mien je organizovaný hierarchicky, a štruktúra DNS zodpovedá tejto hierarchii: ide sprava doľava, začínajúc top-level doménou - doménou najvyššej úrovne (jediná bodka, ktorá sa často vynecháva), jeden z DNS serverov pre túto doménu je dotázaný na meno najbližšie naľavo, čoho výsledkom je obvykle adresa ďalšieho DNS servera, ktorý nám vie poskytnúť viac informácií. Tento proces sa opakuje, kým nie je celé meno preložené (vyriešené, resolved); celkovým výsledkom je obvykle IP adresa hostiteľa.

DNS rekurzia

471

- vezmite si ako príklad domény `www.fi.muni.cz.`
- rezolúcia/preklad začína napravo v koreňových serveroch
  - koreňové servery nás odkážu na `cz.` servery
  - `cz.` servery nás odkážu na `muni.cz`
  - konečne `muni.cz.` nám povie o `fi.muni.cz`

Proces popísaný vyššie sa nazýva rekurzia a je obvykle vykonávaný špeciálnym typom DNS servera, ktorý vykonáva rekurziu v mene svojich klientov a kešuje výsledky pre ďalšie požiadavky. Tiež to znamená, že väčšinou môže začať od stredu, keďže menné servery pre jednu alebo dve najvyššie domény sú veľmi pravdepodobne kešované.

Príklad DNS rekurzie

472

```
$ dig www.fi.muni.cz. A +trace
.               IN NS j.root-servers.net.
cz.             IN NS b.ns.nic.cz.
muni.cz.        IN NS ns.muni.cz.
fi.muni.cz.     IN NS aisa.fi.muni.cz.
www.fi.muni.cz. IN A 147.251.48.1
```

Aby sme mohli pozorovať rekurziu v praxi (a robiť ďalšiu diagnostiku na DNS), môžeme použiť nástroj `dig`, ktorý je súčasťou sady DNS-zameraných nástrojov od ISC (Internet Software Consortium).

Typy DNS záznamov

473

- `A` značí (IP) adresu
- `AAAA` značí IPv6 adresu
- `CNAME` značí alias
- `MX` je pre mailové servery
- a veľa ďalších

Okrem `NS` záznamov, ktoré hovoria systému, koho sa opýtať na ďalšie informácie, existuje veľa typov DNS záznamov, kde každý nesie iný typ informácie o danom mene. Okrem IPv4 a IPv6 adries existujú voľné TXT záznamy (ktoré sa používajú napríklad, aby sa filtrovacie systémy spamu naučili, ktoré sú autorizované mailové servery pre nejakú doménu), SRV záznamy pre zistenie umiestnenia služby na lokálnej sieti, a tak ďalej.

Časť 8.3: Používanie sietí

V tejto sekcii sa zbežne pozrieme na soketové API, ktoré umožňuje aplikáciám používať a poskytovať sieťové služby (na POSIX-ových operačných systémoch) a na niekoľko príkladov sieťových služieb na aplikačnej úrovni.

Pripomenutie soketov

475

- soketové API pochádza z raného BSD Unixu
- soket reprezentuje (možné) sieťové spojenie
- pre otvorený soket dostanete popisovač súboru - file descriptor
- môžete čítať - `read()` a zapisovať - `write()` do soketov
  - ale tiež `sendmsg()` a `recvmsg()`
  - a `sendto()` a `recvfrom()`

Pamätajte, že soket je objekt podobný súboru, prístupný cez popisovač súboru (file descriptor). Na pripojených (connected) prúdových soketoch môžu programy používať tradičné systémové volania `read` a `write`, so sémantikou podobnou tej u rúr. Hoci sú tieto dostupné aj u datagramových soketov, je preferované iné API, pričom jedným z dôvodov je, že pri `read` nie je možné rozlíšiť datagramy, ktoré prichádzajú z rôznych zdrojov.
Systémové volania `sendto`, `recvfrom` umožňujú programu špecifikovať (alebo zistiť, v prípade `recvfrom`) adresu príjemcu (odosielateľa) paketu.

Typy soketov

476

- sokety môžu byť internetové alebo unixové doménové
  - internetové sokety fungujú po sieti
- prúdové sokety sú ako súbory
  - môžete zapísať súvislý prúd dát
  - obvykle implementované použitím TCP
- datagramové sokety posielajú samostatné správy
  - obvykle implementované použitím UDP

Komunikácia na IP sieťach je realizovaná prostredníctvom internetových soketov (s doménou - domain nastavenou na `AF_INET` alebo `AF_INET6`). Ak ide o prúdový soket (stream socket; jeho `type` je `SOCK_STREAM`), komunikácia prebieha použitím TCP (prúdové sokety musia byť explicitne pripojené zavolaním systémového volania `connect` alebo `accept`, ktoré v prípade internetových soketov realizuje TCP handshake - ustanovenie spojenia).
Datagramové sokety (`type` nastavený na `SOCK_DGRAM`) môžu byť voliteľne „pripojené“, avšak toto iba nastaví predvolenú destináciu kam majú byť datagramy posielané. Komunikácia prebieha použitím UDP.

Vytváranie soketov

477

- soket sa vytvára pomocou funkcie `socket()`
- dá sa z neho urobiť server pomocou `listen()`
  - jednotlivé spojenia sú ustanovené cez `accept()`
- alebo klient pomocou `connect()`

Všetky typy soketov sú vytvorené použitím systémového volania `socket` a špecializujú sa na serverové a klientské sokety, v závislosti od následných API volaní, ktoré sú na nich realizované. Serverový soket získame cez `listen` a `bind`, zatiaľ čo klientský soket získame pomocou `connect`. Server potom opakovane volá `accept`, ktorý vracia nový popisovač súboru (deskriptor), ktorý reprezentuje TCP spojenie.

API na preklad mien

478

- `libc` poskytuje modul na preklad mien (resolver)
  - dostupný ako `gethostbyname` (a `getaddrinfo`)
  - tiež `gethostbyaddr` pre reverzné hľadanie (reverse lookup)
- môže sa pozerať na veľa rôznych miest
  - väčšina systémov podporuje aspoň `/etc/hosts`
  - a vyhľadávanie založené na DNS

Soketové API sa zaoberá iba číselnými IP adresami. Ak aplikácia potrebuje byť schopná spojiť počítače použitím ich hostiteľských mien, musí použiť API na preklad mena na adresu, ktoré interne používa vhodnú databázu alebo protokol na nájdenie odpovedajúcich IP adries. Konkrétna sekvencia krokov závisí od konfigurácie systému, ale obvykle použije súbor `/etc/hosts` a rekurzívny DNS server (ktorého IP adresa je opäť súčasťou konfigurácie systému).

Sieťové služby

479

- servery počúvajú na sokete pre prichádzajúce spojenia
  - klient aktívne ustanovuje spojenie so serverom
- sieť jednoducho prenáša dáta medzi nimi
- interpretácia dát je problém vrstvy 7
  - môže ísť o príkazy, prenosy súborov, ...

Väčšina sieťových služieb pracuje v režime klient-server, nad TCP: server pasívne čaká na spojenie na konkrétnej adrese transportnej vrstvy (t.j. IP adresa spolu s číslom portu). Na druhú stranu, klient sa aktívne pripája k poslúchajúcemu serveru, čím ustanovuje obojsmerný kanál (TCP spojenie) medzi nimi. Od tohto bodu sieťové vrstvy (sieťový zásobník) iba posielajú dáta cez kanál. Dáta sa obvykle riadia nejakým protokolom na aplikačnej úrovni (SMTP, HTTP, ...), hoci tento nemusí byť štandardizovaný ani veľmi známy.

Príklady sieťových služieb

480

- (secure) remote shell – sshd
- internet emailová sada
  - MTA = Mail Transfer Agent, používa SMTP
  - SMTP = Simple Mail-Transfer Protocol
- world wide web
  - web servery poskytujú obsah (súbory)
  - klienti a servery používajú HTTP a HTTPS

Klientský softvér

481

- príkaz ssh používa SSH protokol
  - veľmi užitočná systémová utilita na prakticky všetkých UNIX-och
- webový prehliadač je klientom pre world wide web
  - prehliadače sú komplexné aplikačné programy
  - niektoré sú dokonca väčšie než operačné systémy
- emailový klient je tiež známy ako MUA (Mail User Agent)

Časť 8.4: Sieťové súborové systémy

Skôr sme sa naučili, že súborové systémy sú dôležitou, rozšírenou abstrakciou. Je prirodzené umožniť súborovému systému, aby bol prístupný zvonku (z iného počítača) použitím API, ktoré sa používa na lokálny prístup, čím je ‘sieťová’ časť takmer úplne neviditeľná pre program.

Prečo sieťové súborové systémy?

483

- kopírovanie súborov tam a nazad je nepraktické
  - a tiež náchylné na chyby (ktorá je posledná verzia?)
- čo takto ukladať dáta na centrálnom mieste
- a zdieľať ich so všetkými počítačmi na LAN

Pravdepodobne najpresvedčivejší argument pre sieťové súborové systémy pochádza z potreby urobiť pracovné stanice (desktopové počítače) v inštitúcii zameniteľné: to znamená, umožniť ktorémukoľvek užívateľovi prihlásiť sa na ktorúkoľvek z dostupných pracovných staníc a okamžite mať k dispozícii všetky svoje dáta a nastavenia.

NAS (Network-Attached Storage)

484

- (malý) počítač dedikovaný na ukladanie dát
- obvykle na ňom beží osekaný operačný systém
  - často založený na Linuxe alebo FreeBSD
- poskytuje prístup k súborom celej sieti
- niekedy ďalšie služby na úrovni aplikácií
  - napr. správa fotografií, prehrávanie médií, ...

Ďalší typ použitia vychádza z toho, že chceme ukladať dáta, ktoré sú zdieľané niekoľkými užívateľmi na centrálnom zariadení, odkiaľ ich je jednoduché zálohovať a sú prístupné zo všetkých počítačov (a teda

pre všetkých užívateľov, aj keď sú niektoré ostatné počítače vypnuté). Tento typ použitia sa volá NAS (Network-Attached Storage – úložisko pripojené k sieti).

NFS (Network File System)

485

- tradičný UNIX-ový sieťový súborový systém
- zabudovaný pomerne hlboko do kernelu
  - predpoklad, že máme všeobecne spoľahlivú sieť (LAN)
- súborové systémy sú exportované na použitie cez NFS
- klientská strana pripojí - mounts NFS-exportovaný zväzok (volume)

NFS je jednou z prvých implementácií sieťového súborového systému. Je v podstate založený na tom, že sa napojí na rozhranie VFS (virtual file system switch) a exportuje ho cez rozhranie vzdialeného volania procedúr do ďalších kernelov na sieti. Pre vytvorenie tzv. NFS share, čo je zdieľaný kus súborového systému, musí byť lokálny súborový systém exportovaný na plánovanom NFS serveri; potom môže byť pripájaný (mounted) klientmi, čím tvorí zdieľanú časť ich lokálnej hierarchie súborového systému.

História NFS

486

- vytvorený Sun Microsystems v 80. rokoch
- v2 implementovaná v System V, DOS, ...
- v3 sa objavila v '95 a ešte stále sa používa
- v4 prichádza v 2000, zlepšuje bezpečnosť

Sieťové súborové systémy sú pomerne starou technológiou (takmer 40 rokov starou), ale prešli výraznou evolúciou v priebehu prvých asi 20 rokov, pričom verzia 4 rieši primárne bezpečnostné záležitosti.

Pripomenutie VFS

487

- implementačný mechanizmus pre niekoľko typov súb. systémov
- objektovo-orientovaný prístup
  - `open`: nájdi súbor, aby sa k nemu dalo pristupovať
  - `read`, `write` – netreba vysvetľovať
  - `rename`: premenuj súbor alebo adresár

Spomeňte si zo 4. prednášky, že VFS (virtual file system switch) je mechanizmus vnútri kernelu, ktorý umožňuje, aby viaceré implementácie súborového systému mohli poskytovať zjednotené rozhranie zvyšku kernelu. NFS využíva toto existujúce rozhranie a sprístupňuje ho cez sieť. Samozrejme, na rozdiel od samotného VFS je sémantika NFS funkcií štandardizovaná naprieč implementáciami (NFS klienti a servery sú zväčša kompatibilné naprieč rôznymi UNIX-ovými operačnými systémami).

RPC (Remote Procedure Call) - Vzdialené volanie procedúr

488

- akýkoľvek protokol pre volanie funkcií na vzdialených hostiteľoch
  - ONC-RPC = Open Network Computing RPC
  - NFS je založené na ONC-RPC (tiež známe ako Sun RPC)
- NFS v podstate vykonáva VFS operácie použitím RPC
  - jednoduché na implementáciu na UNIX-ových systémoch

Spôsob, akým je NFS rozhranie sprístupňované sieti, je cez mechanizmus vzdialeného volania procedúr, ktorý v podstate vezme volanie procedúry (názov funkcie, spolu s argumentami, s ktorými má byť zavolaná), zabalí ich do reťazca bajtov (byte string) a pošle ho cez sieť ďalšiemu počítaču, ktorý reálne zavolá túto procedúru a pošle nazad výsledok. Protokol má mechanizmus na posielanie bufferov dát, nad rámec primitívnych hodnôt (integers - celočíselný typ).

Prideľovanie portov službou `portmap`

489

- ONC-RPC je spúšťané cez TCP alebo UDP
  - ale je dynamickejšie s ohľadom na dostupné služby
- TCP/UDP čísla portov sú pridelené na požiadanie
- `portmap` prekladá z RPC služieb na čísla portov
  - samotný prekladač portov počúva na porte 111

V moderných systémoch je ONC-RPC implementované výhradne nad TCP/IP zásobníkom. Keďže tento protokol môže sprístupňovať viacero služieb na každom stroji, vzniká potreba prekladať medzi týmito RPC službami a TCP/UDP číslami portov. Vo väčšine prípadov sa o toto stará RPC služba nazvaná 'portmapper', ktorá sama beží na fixnom porte (číslo 111).

NFS démon

490

- tiež známy ako `nfsd`
- poskytuje NFS prístup do lokálneho súborového systému
- môže bežať ako systémová služba
- alebo môže byť súčasťou kernelu
  - toto je typickejšie z dôvodu výkonnosti

Ak máme RPC zásobník, NFS je dostupná cez `nfsd`, ktorý sa zaregistruje ako služba na RPC zásobníku. Tento démon môže byť riadny démon v užívateľskom priestore, ale môže byť tiež súčasťou kernelu (bežať ako kernelové vlákno).

SMB (Server Message Block)

491

- sieťový súborový systém od Microsoftu
- dostupný na Windowse od verzie 3.1 (1992)
  - pôvodne bežal nad NetBIOS
  - neskoršie verzie používali TCP/IP
- SMB1 malo kopu zbytočných vecí a bolo zložité

SMB je úplne iná implementácia sieťovo transparentnej vrstvy pre súborové systémy. Podobne ako NFS, nie je späté s konkrétnou štruktúrou na disku. SMB prešlo mnohými inkrementálnymi zmenami s každým novým operačným systémom Microsoftu, ktorý prišiel, zatiaľ čo bolo súčasne udržiavané spätne kompatibilné, aby mohli staršie operačné systémy navzájom spolupracovať, ako klienti aj servery. Týmto sa stal protokol extrémne komplikovaným, takže ďalšie rozšírenia boli nepraktické.

SMB 2.0

492

- jednoduchšie ako SMB1 vďaka menej rozsiahlej kompatibilite
- lepšia výkonnosť a bezpečnosť
- podpora pre symbolické odkazy
- dostupné od Windows Vista (2006)

Microsoft navrhol nový protokol pre súborové systémy po sieti vo

svojom operačnom systéme Windows Vista, pod názvom SMB 2.0. Podobne ako NFSv4 niekoľko rokov predtým, SMB 2 vyriešilo veľa bezpečnostných zraniteľností svojho predchodcu, a zároveň zlepšilo výkonnosť a rozšírilo protokol na podporu novej funkcionality súborových systémov, ako sú symbolické odkazy.

Review Questions

493

29.What is ARP (Address Resolution Protocol)?
30.What is IP (Internet Protocol)?
31.What is TCP (Transmission Control Protocol)?
32.What is DNS (Domain Name Service)?

Časť 9: Shelly & užívateľské rozhrania

Táto prednáška sa zameriava na interakciu človeka s počítačom a rolu operačného systému v tejto oblasti. Pozrieme sa na textové formy interakcie (najmä rozhranie príkazového riadku - command line interface, shell), aj na grafické rozhrania, ovládané ukazovacím zariadením - pointing device (myš, trackpad) alebo dotykovou obrazovkou.

Obsah prednášky

495

1. Interprety príkazov
2. Príkazový riadok
3. Grafické rozhrania

Prvá časť sa zameria na shell ako na jednoduchý programovací jazyk, zatiaľ čo v druhej sa v krátkosti pozrieme na terminály (alebo skôr emulátory terminálov), interaktívne použitie shellu a ďalšie textové programy. Konečne, tretia časť bude o grafických rozhraniach a o tom, ako sú postavené.

Časť 9.1: Interprety príkazov

Historicky zohrávali shelly na väčšine operačných systémov dvojitú úlohu. Interakcia pomocou príkazov je pravdepodobne najjednoduchšia na implementáciu, a preto bola formou, ktorá sa spočiatku v počítačoch a operačných systémoch používala. Teda hneď ako sa interaktívne terminály stali dostupnými (dnes preskočíme dávkové systémy).
Každý interpret príkazov má jednu zaujímavú vlastnosť: môžete vytvoriť záznam sekvencie príkazov, aby ste dosiahli zložitejšie úlohy než dokáže vykonať ktorýkoľvek individuálny príkaz. Funguje to bez akéhokoľvek zásahu od samotného interpretu príkazov: môžete si ich jednoducho poznačiť na kúsok papiera a neskôr ich prepísať.
Bolo by samozrejme výrazne pohodlnejšie keby počítač mohol čítať príkazy jeden za druhým zo súboru a spúšťať ich, ako keby ste ich zadávali. Toto je pôvod shellových skriptov.

Shell

497

- programovací jazyk zameraný na interakciu s OS
- jednoduché riadenie toku (control flow)
- netypované, textovo-orientované premenné
- pochybné riešenie chybových stavov (error handling)

Samozrejme, vo svojej kópii alebo ručne písaných poznámkach si môžete pridať dodatočné komentáre a inštrukcie, ako „tento príkaz spustiť iba ak bol predchádzajúci príkaz úspešný“, alebo „zopakovať tento príkaz 3 krát“ alebo „opakovať tento príkaz až kým sa nestane to a to“. Nebolo by úžasné, keby ste si mohli zahrnúť takéto anotácie do zápisu, ktorý počítač číta a vykonáva?
Ale isteže, práve sme vynašli riadenie toku (control flow). A pochopiteľne je to presne to, čo shelly implementujú. Ďalším ‘zjavným’ vynálezom sú zástupné symboly - placeholders - v príkazoch, ktoré sú nahradené vhodnými hodnotami za behu. Napríklad, vo svojom zošite si poznačíte sekvenciu príkazov na aktualizáciu zoznamu užívateľov uložených v textovom súbore: pravdepodobne použijete zástupný symbol pre názov súboru, s ktorým momentálne pracujete. A keď prepíšete príkaz nazad, nahradíte každý výskyt tohto zástupného symbolu skutočným názvom daného súboru. Práve ste vynašli premenné!
Ďalšia vec, ktorá sa tak nejak preniesla zo skriptov zaznamenaných na papieri do spustiteľných je kontrola chýb... alebo skôr jej neexistencia. Asi by ste sa nenamáhali inštruovať sa, že máte prestať a zistiť príčinu ak jeden z príkazov z vášho zošita neočakávane zlyhá.

Interaktívne shelly

498

- takmer všetky shelly majú interaktívny mód
- užívateľ zadá jeden príkaz na klávesnici
- keď ho potvrdí, tento sa okamžite vykoná
- toto je základ rozhrania príkazového riadku (CLI)

Než sa dostaneme ku toku riadenia (control flow) a premenným, pripomeňme si, že väčšina shellov je zo svojej podstaty interaktívnych. V tomto interaktívnom móde užívateľ zadá jeden ‘príkaz’ (jeden riadok) a potvrdí ho, po čom je hneď vykonaný. Najčastejšie sa jedná o jeden príkaz, ale môže ísť aj o sekvenciu príkazov, cyklus, alebo akýkoľvek iný konštrukt, ktorý jazyk dovoľuje: neexistuje rozdiel medzi syntaxou, ktorá je dostupná v shellových skriptoch a interaktívnym príkazovým riadkom. To umožňuje písať krátke skripty (tzv. ‘one-liners’ - jednoriadkové príkazy) priamo na príkazovom riadku, na zautomatizovanie jednoduchých úloh, bez nutnosti niekde zapísať tento program. Naučiť sa to robiť vážne stojí za investíciu, keďže to dokáže ušetriť značné množstvo času v každodennej práci.

Shellové skripty

499

- shellový skript je (spustiteľný) súbor
- v najjednoduchšej forme ide o sekvenciu príkazov
  - každý príkaz je na samostatnom riadku
  - spustiť skript je zhruba to isté ako ručne zadať príkazy
- ale môže používať prvky štruktúrovaného programovania

Na rozdiel od interaktívneho spúšťania príkazov je shellový skript súbor so zoznamom príkazov vnútri, ktoré sú vykonávané sekvenčne. Samozrejme, ako sme rozoberali vyššie, je dostupný základný tok riadenia, na pozmenenie tohto sekvenčného vykonávania, ak je to potrebné. Možno použiť premenné na substitúciu častí príkazov, ktoré sa menia z jedného volania skriptu na ďalšie.

Výhody shellu

500

- je veľmi jednoduché písať základné skripty
- prvá voľba pre jednoduchú automatizáciu
- často môže ušetriť opakované písanie príkazov
- definitívne nevhodné pre veľké programy

Ako z toho shell vychádza ako programovací jazyk? Na začiatok, môže byť veľmi užitočný a veľmi jednoduchý na použitie, najmä v prípadoch kedy nerobíte programovanie ako také, ale chcete iba zautomatizovať jednoduché úlohy, ktoré by ste inak robili ručne, písaním príkazov.
Avšak, skúste si vytvoriť čokoľvek väčšie a limity sa stávajú veľmi podstatnými: väčšie programy nemôžu jednoducho umrieť keď niečo zlyhá, a tiež nemôžu ignorovať chyby všade naokolo (dve základné stratégie dostupné v skriptoch). Nedostupnosť štruktúrovaných dát a typového systému, a všeobecne 'programovacej hygieny' spôsobuje, že väčšie skripty sú krehké a ťažko sa udržiavajú. Ďalší logický krok je dedikovaný 'skriptovací' jazyk, ako Perl alebo Python, ktoré predstavujú kompromis medzi naivitou (a jednoduchosťou) shellu a štruktúrou a prísnosťou objemných programovacích jazykov ako C++ alebo Java.

Bourne Shell

501

- konkrétny jazyk zo 'shellovej' rodiny
- prvý shell s konzistentnou podporou programovania
  - dostupný od 1976
- kompatibilné shelly sú dnes stále značne rozšírené
  - najznámejšia implementácia je bash
  - /bin/sh je vyžadovaný POSIX-om

Bourne shell bol vytvorený v roku 1976 a v podstate kodifikoval duálnu povahu shellov ako interaktívnych a zároveň programovateľných. Dodnes používame jeho základný model (a syntax). Existuje veľa Bourne-kompatibilných shellov, z ktorých mnohé pochádzajú z Korn shellu (ksh, o ktorom si za chvíľu povieme).
Možno ste počuli o bash: jeho názov znamená Bourne Again Shell[7] a ide pravdepodobne o najznámejší shell, aký existuje (až do tej miery, že niektorí ľudia si myslia, že ide o jediný shell).

C Shell

502

- tiež známy ako csh, prvýkrát vyšiel v 1978
- viac syntax na štýl jazyka C než sh (Bourne Shell)
  - ale nie až tak veľmi ako C
- vylepšil interaktívny mód (oproti sh z roku '76)
- tiež sa používa dodnes (najmä cez tcsh)

Historicky bol druhý najznámejší UNIX-ový shell C shell[8] – priniesol zlepšenia v interaktívnom režime, z ktorých mnohé boli adoptované do ďalších shellov, okrem iného:

- história príkazov (schopnosť vrátiť sa k už vykonaným príkazom),
- aliasy (užívateľom definované skratky pre často používané príkazy),
- automatické dopĺňanie príkazov a názvov súborov (cez tcsh),
- interaktívna správa úloh.

tcsh vetva je variantom csh s dodatočnou funkcionalitou, ktorá bola udržiavaná popri pôvodnom csh od raných 80. rokov. Ešte stále je distribuovaná spolu s mnohými operačnými systémami (a je, napríklad, predvoleným root shellom na FreeBSD).

Korn Shell

503

- tiež známy ako ksh, vydaný v roku 1983
- zlatá stredná cesta medzi sh a csh
- základ pre požiadavky POSIX.2
- existuje niekoľko implementácií

V podstate zlúčenie sh (Bourne shell) a csh/tcsh (najmä ako zdroj vylepšenej interakcie s užívateľom; syntax skriptov zostala verná sh). Originál bol založený na zdrojovom kóde sh s mnohými pridanými funkciami. Toto je shell, ktorý POSIX používa ako model pre /bin/sh.

Príkazy

504

- typicky názov spustiteľnej binárky
  - prípadne príkaz toku riadenia
  - alebo vstavaný príkaz (built-in)
- binárka je vyhľadaná v súborovom systéme
- shell urobí fork + exec
  - znamená to nový proces pre každý príkaz
  - vytváranie procesov je pomerne drahé

Najtypickejším príkazom je jednoducho názov programu, prípadne nasledovaný argumentmi (ktoré nie sú interpretované shellom, sú jednoducho posunuté programu ako forma vstupu).
Príkazy v tejto podobe sú vykonávané, koncepčne, nasledovne (detaily sa môžu v skutočných implementáciách líšiť, napr. bod 2 môže byť urobený ako súčasť bodu 4):

1. skontroluj, že daný príkaz nie je názov vstavaného príkazu alebo konštruktu (ak je, bude spracovaný inak),
2. názov programu sa považuje za názov spustiteľného súboru – je prehľadaný zoznam adresárov (daný premennou PATH, ktorá bude vysvetlená neskôr), aby sa zistilo, či spustiteľný súbor s daným názvom existuje,
3. shell vykoná systémové volanie fork, aby vytvoril nový proces (pozri prednášku 3),
4. detský proces použije exec, aby začal vykonávať spustiteľný súbor, ktorý bol nájdený v bode 2, a posunie mu prípadné argumenty príkazového riadku,
5. hlavný shellový proces urobí wait (t.j. preruší vykonávanie než spustený program skončí).

To znamená, že každý príkaz zahŕňa podstatné množstvo práce, čo nie je problém u interaktívneho použitia, alebo rozumne veľkých shellových skriptov. Vykonávať niekoľko tisíc príkazov, obzvlášť ak príkazy samotné bežia rýchlo, však môže byť trochu pomalé.

Vstavané príkazy

505

- cd (change directory) - zmeň pracovný adresár
- export pre nastavovanie prostredia
- echo vypíš správu
- exec nahraď shellový proces (bez fork)

Niektoré príkazy vo forme program [argumenty] sú interpretované shellom špeciálne (t.j. nepoužijú vyššie uvedený proces fork + exec). Sú v podstate dva základné dôvody prečo sa toto deje:

1. efektivita – niektoré príkazy sú často používané, najmä v skriptoch, a neustále vytváranie nových procesov je drahé – ide čisto o optimalizáciu, a je to prípad vstavaných príkazov ako echo alebo

[7] Because bad puns should be a human right. (pozn. prekladu: Ide o slovnú hračku na slovné spojenie Bourne Again, ktoré znie ako born again = znovuzrodený; a zároveň „zase Bourne" shell.)
[8] Ľudia od počítačov a trápne slovné hračky... (C shell znie ako sea shell, morská lastúra).

`test`,
2. funkcionalita – niektoré efekty nemôžu byť (jednoducho, rozumne) vykonané detským procesom, hlavne pretože zmeny musia byť urobené na hlavnom shellovom procese: zvyčajne vie tieto zmeny robiť iba hlavný proces – je to prípad `cd` (mení aktuálny pracovný adresár hlavného procesu), `export` (mení prostredie hlavného procesu, tiež pozri nižšie) alebo `exec` (vykoná `exec` bez `fork`, čím zničí shell).

Premenné / Parametre

506

- názvy premenných pozostávajú z písmen a číslic
- používanie premenných je indikované s `$`
- pri nastavovaní premenných sa nepoužíva `$`
- všetky premenné sú globálne (okrem subshellov)

```
premenna="nejaky text"
echo $premenna
```

Spomínali sme si myšlienku 'zástupných symbolov' (placeholders) v kontexte skriptov poznačených v zošitoch. Shelly túto myšlienku vezmú pomerne doslova, a urobia z nej to, čomu hovoríme premenné (aspoň vo väčšine programovacích jazykov; 'oficiálna' terminológia v shelli je parametre – čo je v súlade s myšlienkov zástupného symbolu). Shelly si v podstate udržiavajú mapovanie z názvov na hodnoty, kde názvy sú reťazce pozostávajúce z písmen a číslic a hodnoty sú ľubovoľné reťazce. Pre vytvorenie alebo aktualizáciu mapovania sa používa nasledujúci príkaz:

```
premenna="nejaky text"
```

Úvodzovky nie sú nutné pokiaľ hodnota neobsahuje medzery. Biele znaky okolo `=` nie sú dovolené (zápis `premenna = hodnota` je interpretovaný ako príkaz `premenna` s argumentami `=` a `hodnota`).

Nahrádzanie parametrov / substitúcia premenných

507

- premenné sú substituované ako text
- `$foo` je jednoducho nahradené obsahom `foo`
- aritmetické operácie nie sú na väčšine shellov rozumne podporované
  - alebo akákoľvek iná syntax s výrazmi, napr. relačné operátory
  - vezmite si napríklad POSIX syntax `z=$((x + y))`

Premenné (parametre) môžu byť použité viac menej kdekoľvek v ktoromkoľvek príkaze, vrátane názvu príkazu. Robí sa to cez zápis pomocou symbolu dolára, `$`, za ktorým nasleduje názov premennej (parametra), takto:

```
echo $premenna
```

Príkaz vypíše `nejaky text`. Substitúcia sa deje čisto textovou formou (tento proces je okrem názvu substitúcia premenných (variable substitution) tiež známy ako parameter expansion - nahrádzanie parametrov). Po substitúcii je všetko o premenných 'zabudnuté', v zmysle, že to, či nejaká časť textu pochádza zo substitúcie, alebo či bola prítomná od začiatku nerobí rozdiel a nie je to možné rozlíšiť. Môže dochádzať k prekvapeniam ak hodnota premennej obsahuje medzery (o tom si povieme neskôr).
Ak prichádza z normálnych programovacích jazykov, užívateľa to môže navádzať, aby napísal niečo ako `$a + $b` v shelli. Nebude to fungovať: ak `a=7` a `b=3`, 'výraz' vyššie bude interpretovaný ako príkaz `7` s argumentami `+` a `3`. Pre vykonanie aritmetických operácií v shellových skriptoch musí byť výraz obalený do `$(( … ))` – aby to bolo trochu menej bolestivé, premenné vnútri `$(( … ))` nemusia byť uvádzané `$`. Stále sú však substituované ako text:

```
a=3+1; echo $a = $((a))
```

vypíše `3+1 = 4` (a nie napr. error pretože `a` nie je číslo).[9] Substitúcie vnútri `$(( … ))` bez znakov dolára sú však zátvorkované – konkrétne,

```
a=3+1; b=7; echo $((a * b))
```

vypíše `28`, keďže je rozbalené (expanded) ako `$(((3+1) * 7))`. Toto nie je prípad `$` substitúcií:

```
a=3+1; b=7; echo $(($a * $b))
```

vypíše `10`.

Substitúcia príkazov (command substitution)

508

- v podstate podobná substitúcii parametrov
- zapisuje sa ako `` `prikaz` `` alebo `$(prikaz)`
  - najprv vykoná príkaz
  - a zaznamená jeho štandardný výstup
  - potom nahradí `$(prikaz)` jeho výstupom

Niekedy je žiaduce vypočítať časť príkazu, najčastejšie spustením iného programu. Toto sa dá dosiahnuť použitím `$( … )`, napr. `cat $(ls)`:

1. najprv sa vykoná `ls` ako shellový príkaz (keďže je to názov programu, bude klasicky `fork`-nutý a `exec`-nutý),
2. výstup `ls` (zoznam súborov v pracovnom adresári, napr. `foo.txt bar.txt`) je zaznamenaný do bufferu,
3. obsah bufferu je substituovaný do pôvodného príkazu, t.j. `cat foo.txt bar.txt`,
4. príkaz sa vykoná ako obvykle.

Rovnako ako u substitúcie parametrov, sú prítomné problémy s medzerami a bielymi znakmi (pozri nižšie).

Používanie úvodzoviek

509

- biele znaky slúžia ako oddeľovač argumentov v shelli
- argumenty zložené z viacerých slov musia byť v úvodzovkách
- úvodzovky môžu byť dvojité `"x"` alebo jednoduché `'x'`
  - dvojité úvodzovky umožňujú substitúciu premenných

Úvodzovky a substitúcia

510

- medzery zo substitúcie premenných musia byť v úvodzovkách
  - `foo="hello world"`
  - `ls $foo` je iné než `ls "$foo"`
- nesprávne použitie úvodzoviek je častý zdroj chýb
- tiež si vezmite názvy súborov, ktoré v sebe majú medzery

Dôležitou črtou substitúcie parametrov (premenných) je, že sa robí pred rozdeľovaním argumentov. To znamená, že hodnoty, ktoré obsahujú medzery môžu byť po substitúcii interpretované ako niekoľko argumentov. Niekedy je to žiaduce, ale pomerne často nie je. Vezmite si `cat $file`, kde autor očividne očakáva, že `$file` sa nahradí za jediný

[9] V závislosti od implementácie, `a=3+b; b=7; echo $((a))` môže a nemusí fungovať (v zmysle, že vypíše `10`).

názov súboru. Avšak, ak obsahuje hodnotu `foo bar`, príkaz bude rozbalený na `cat foo bar` a vykoná program `cat` s argumentami `foo` a `bar`. Úvodzovky môžu byť použité, aby sa tomuto zabránilo.
Vezmite si príklad na slajde vyššie: prvý príkaz, `ls $foo` sa rozbalí na `ls hello world` a bude spustený s:

```
argv[ 0 ] = "ls"
argv[ 1 ] = "hello"
argv[ 2 ] = "world"
```

Vskutku, ako pri príklade s `cat`, bude hľadať dva samostatné súbory. Druhá možnosť, `ls "$foo"` sa vykoná ako:
argv[0] = "ls" argv[1] = "hello world"

Špeciálne premenné

511

- `$?` je výsledok posledného príkazu
- `$$` je PID aktuálneho shellu
- `$1` až `$9` sú pozičné parametre
  - `$#` je počet parametrov
- `$0` je názov shellu – `argv[0]`

Okrem premenných (parametrov), ktoré užívatelia sami nastavia, shell poskytuje niekoľko 'špeciálnych' premenných, ktoré obsahujú užitočné informácie. Pozičné parametre odkazujú na argumenty príkazového riadka, ktoré boli dané aktuálne prebiehajúcemu shellovému skriptu. Tu je niekoľko ďalších premenných:

- `$@` sa rozbalí na všetky pozičné parametre, so špeciálnym chovaním pri obalení dvojitými úvodzovkami (každý parameter je daný do úvodzoviek zvlášť),
- `$*` to isté, ale bez špeciálneho chovania pri úvodzovkách,
- `$!` PID posledného 'procesu na pozadí' (vytvoreného pomocou operátora `&`, o ktorom si povieme neskôr),
- `$-` možnosti/nastavenia shellu.

Prostredie - Environment

512

- je niečo ako premenné shellu, ale nie to isté
- prostredie sa posúva všetkým vykonávaným programom
  - dieťa nemôže modifikovať prostredie svojho rodiča
- premenné sú umiestnené do prostredia pomocou `export`
- premenné prostredia sa často správajú ako nastavenia

POSIX má koncept premenných prostredia (environment variables), ktoré sú nezávislé od akéhokoľvek shellu: posielajú sa z procesu na proces, cez `fork` aj cez `exec`. Avšak, keďže `fork` vyrába novú kópiu celého prostredia, zmeny v týchto premenných sa dajú poslať iba nadol (do nových detských procesov), nikdy nie nahor (do rodičovských procesov), alebo všeobecne do už bežiacich procesov.
Hoci sú formálne nezávislé od shellu, premenné prostredia majú podobnú sémantiku: ich názvy sú alfanumerické reťazce a ich obsah je ľubovoľný text. Aby sme to urobili ešte mätúcejším, shelly sa ku premenným prostredia chovajú rovnako ako ku svojím „interným" premenným (parametrom). Ak je `FOO` premenná prostredia, shell nahradí `$FOO` jej hodnotou, a zavolanie `FOO=bar` ako shellového príkazu zmení jeho hodnotu v hlavnom shellovom procese (a teda vo všetkých jeho budúcich detských procesoch).

Dôležité premenné prostredia

513

- `$PATH` hovorí systému kde nájsť programy
- `$HOME` je domovský adresár aktuálneho užívateľa
- `$EDITOR` a `$VISUAL` nastavujú ktorý textový editor použiť
- `$EMAIL` je e-mailová adresa aktuálneho užívateľa
- `$PWD` je aktuálny pracovný adresár

Podľa konvencie sú všetky premenné prostredia pomenované veľkými písmenami. Existuje niekoľko 'známych' premenných, ktoré ovplyvňujú chovanie rôznych programov: premenná `PATH` obsahuje zoznam adresárov, v ktorých sa hľadajú binárky (pri spúšťaní príkazov v shelli, ale tiež pri zavolaní programov pomocou ich názvu z iných programov). Premenná `HOME` hovorí programom, kde ukladať súbory patriace užívateľovi (dáta aj konfiguračné súbory), a tak ďalej. Niektoré sú nastavené sytémom pri vzniku užívateľskej relácie (`HOME`, `LOGNAME`), ďalšie sú nastavené shellom (`PWD`), niektoré sú typicky nakonfigurované systémovým administrátorom (ale môžu byť zmenené užívateľmi), ako napríklad `PATH`, a ďalšie sú nakonfigurované užívateľom (`EDITOR`, `EMAIL`).

Globbing

514

- vzory pre rýchle vymenovanie niekoľkých súborov
- napr. `ls *.c` vypíše všetky súbory končiace na `.c`
- `*` zastupuje ľubovoľné množstvo znakov
- `?` zastupuje jeden ľubovoľný znak
- funguje to aj s celými cestami – `ls src/*/*.c`

Poďme sa vrátiť nazad k shellu a jeho syntaxi. Keďže sú súbory všadeprítomné a veľa príkazov očakáva názvy súborov ako argumenty, shelly poskytujú špeciálnu funkcionalitu na prácu s nimi. Jednou takouto funkcionalitou je globbing, kde jeden vzor môže nahradiť potencionálne dlhý zoznam názvov súborov (a teda ušetrí množstvo zdĺhavého písania).
Rozbaľovanie globov robí shell samotný, t.j. program dostane jednotlivé názvy súborov ako argumenty, nie glob. Úvodzovky (jednoduché aj dvojité) zabraňujú rozbaľovaniu globov (je to užitočné, ak chceme dodať reťazce, ktoré obsahujú `*` alebo `?` ako argumenty). Reťazce, ktoré nie sú v úvodzovkách a obsahujú ktorýkoľvek z 'meta-znakov' globov sú považované (a rozbalené) za glob, vrátane výsledkov rozbaľovania parametrov (substitúcie).

Podmienkové príkazy

515

- umožňujú podmienené vykonanie príkazov
- `if cond; then cmd1; else cmd2; fi`
- tiež `elif cond2; then cmd3; fi`
- `cond` je tiež príkaz (použije sa návratový kód)

Najzákladnejším prvkom riadenia toku (control flow) je podmienené vykonávanie kódu, kde je príkaz vykonaný alebo preskočený na základe výsledku predchádzajúceho príkazu. Shelly používajú tradičné kľúčové slovo `if`, voliteľne nasledované klauzulami `elif` a `else`.
Na rozdiel od väčšiny programovacích jazykov, `cond` nie je výraz, ale bežný príkaz. Ak príkaz 'uspeje' (skončí s návratovým kódom 0), toto je interpretované ako 'true' (pravda) a potom sa vykoná vetva `then`. inak sú postupne vyhodnotené vetvy `elif` (ak sú prítomné) a ak žiadna neuspeje, vykoná sa vetva `else` (opäť, ak je prítomná).

`test` (vyhodnocovanie booleovských výrazov)

516

- pôvodne externý program, tiež známy ako `[`
  - dnes vstavaný vo väčšine shellov
  - obchádza absenciu výrazov v shelli
- vyhodnotí svoje argumenty a vráti `true` alebo `false`
  - môže byť použitý s príkazmi `if` a `while`

Hoci je podmienka príkazu v príkaze `if` tiež príkaz, často by bolo užitočné byť schopný špecifikovať výrazy, ktoré vzťahujú premenné navzájom, alebo ktoré kontrolujú prítomnosť súborov. Za týmto účelom POSIX špecifikuje špeciálny program nazvaný `test` (v skutočnosti vstavaný vo väčšine shellov).
Príkaz `test` dostáva argumenty ako ktorýkoľvek iný príkaz, vyhodnotí ich, aby získal booleovský výraz a nastaví svoj návratový kód na základe tejto hodnoty tak, aby sa `if test …; then …` choval očakávaným spôsobom.

Príklady `test`

517

- `test file1 -nt file2` → 'nt' = newer than - novší ako
- `test 32 -gt 14` → 'gt' = greater than - väčší ako
- `test foo = bar` → string equality - rovnosť reťazcov
- dá sa skombinovať so substitúciou premenných (`test $y = x`)

`test` poskytuje 3 triedy predikátov:

1. existencia a vlastnosti súborov,
2. porovnanie celých čísel, a
3. porovnanie reťazcov.

Posledné dva napodobňujú to, čo poskytujú 'normálne' programovacie jazyky (hoci so zvláštnou syntaxou). Prvý zjednodušuje a robí pohodlnejším písanie príkazov, ktoré sa majú vykonať, iba ak konkrétny súbor existuje (alebo chýba), čo je veľmi bežnou úlohou v shellovom programovaní.

Cykly

518

- `while cond; do cmd; done`
  - `cond` je príkaz, ako v `if`
- `for i in 1 2 3 4; do cmd; done`
  - dajú sa použiť globy: `for f in *.c; do cmd; done`
  - tiež substitúcia príkazov
  - `for f in $(seq 1 10); do cmd; done`

Po podmienenom vykonávaní kódu sú cykly druhým najzákladnejším prvkom. Opäť, podobne ako u univerzálnych programovacích jazykov, cykly umožňujú shellovým skriptom opakovať sekvenciu príkazov, buď:

1. až kým nejaký konkrétny príkaz nezlyhá (`while` cyklus, spomínaný príkaz často býva `test`, hoci môže ísť samozrejme o akýkoľvek príkaz),
2. raz pre každú hodnotu v zozname, často zozname názvov súborov (ktorý môže byť vytvorený napr. použitím globov).

Ďalšia z častých foriem `for` cyklu používa substitúciu príkazov (command substitution/expansion) na generovanie zoznamu. Často používaným pomocným príkazom (bohužiaľ, neštandardným) v tomto kontexte je utilita `seq`, ktorá generuje sekvenciu čísel. Podobná (a rovnako neštandardná) utilita nazvaná `jot` je dostupná na BSD systémoch.

Príkaz `case`

519

- zvolí príkaz na základe zhody vzoru - pattern matching
- `case $x in *.c) cc $x;; *) ls $x;; esac`
  - áno, `case` skutočne používa nevyvážené zátvorky
  - `;;` indikuje koniec každého prípadu

Trochu pokročilejším prvkom riadenia toku (control flow) je analýza prípadov – príkaz `case`, ktorý dovoľuje použitie zhody vzoru – pattern matching – na spôsob globov na ľubovoľných reťazcoch (t.j. nielen na názvoch súborov). Reťazec, voči ktorému sa má porovnávať je dodaný za `case`, a je zvyčajne výsledkom rozbalenia parametrov alebo príkazov (parameter or command expansion). Všimnite si, že vzory za klauzulou `in` príkazu `case` nie sú nahradené zoznamom názvov súborov (čo by sa stalo, ak by boli interpretované ako globy).

Reťazenie príkazov

520

- `;` (bodkočiarka): spusti dva príkazy za sebou
- `&&` vykonaj druhý príkaz ak prvý uspeje
- `||` vykonaj druhý príkaz ak prvý zlyhal
- napr. skompiluj a spusť: `cc file.c && ./a.out`

Zatiaľ čo je najpriamočiarejší operátor pre reťazenie príkazov `;` (bodkočiarka) asi príliš primitívny na to, aby sa dal nazvať riadením toku programu, existuje niekoľko podobných operátorov, ktoré sú zaujímavejšie. Prvou množinou sú booleovské kombinátory `&&` a `||`, ktoré v podstate fungujú ako skratková syntax pre príkazy `if`. Keďže príkazy skombinované cez `&&` a `||` sú opäť príkazy, tieto môžu vystupovať v podmienke príkazov `if` alebo `while`.
Sú však užitočné aj ako samostatne stojace príkazy, a tiež v interaktívnom móde. Obzvlášť `&&` môže byť použité pre zapísanie sekvencie príkazov, ktorá má skončiť pri prvom zlyhaní, čím výrazne znižuje latenciu spôsobenú interakciou (kedy užívateľ čaká až sa každý príkaz dokončí, a po každom príkaze počítač čaká až užívateľ zadá ďalší príkaz).

Rúry - Pipes

521

- shelly môžu vykonávať príkazy zreťazené rúrami
- `cmd1 | cmd2 | cmd3`
  - všetky príkazy bežia paralelne
  - výstup `cmd1` sa stáva vstupom pre `cmd2`
  - výstup `cmd2` je spracovaný `cmd3`

echo hello world | sed -e s,hello,goodbye,

Pravdepodobne najmocnejšou funkciou shellu sú rúry - pipes, ktoré poskytujú veľmi flexibilný a mocný (aj keď veľmi jednoduchý) spôsob ako skombinovať niekoľko príkazov. Operátor rúry spôsobí, že sa oba príkazy vykonajú paralelne, a všetko, čo prvý program zapíše na svoj štandardný výstup je poslané druhému programu na jeho štandardný vstup. POSIX špecifikuje značné množstvo pomocných programov špeciálne navrhnutých, aby fungovali dobre v takýchto zreťazeniach, a veľa ďalších je dostupných ako rozšírenia od dodávateľov alebo ako externé softvérové balíčky.

Funkcie

522

- tiež v shelli môžete definovať funkcie
- hlavne odľahčená alternatíva voči skriptom
  - nie je nutné `export`-ovať premenné
  - ale nemôžu byť zavolané z programov mimo shell
- funkcie tiež môžu nastavovať premenné

Spomeňte si, že prostredie sa posiela iba nadol, nikdy nie nazad nahor. To znamená, že shellový skript, ktorý nastavuje premennú neovplyvní rodičovský shell. Avšak, vo funkciách (a keď sú skripty zavolané pomocou `.`) môžu byť premenné nastavené a efekt týchto zmien je viditeľný v skripte, ktorý funkciu zavolal.

Časť 9.2: Príkazový riadok

Hoci sú v istom zmysle interaktívne aspekty shellov pre užívateľov výrazne viditeľnejšie, nie sú až tak teoreticky zaujímavé. O interaktívnom shelli sa môžete viac naučiť jednoducho tým, že ho budete používať a objavovať jeho funkcie počas používania a podľa toho, čo vám v danom momente pripadá užitočné. Napriek tomu si urobíme rýchly prehľad základnej funkcionality, ktorú poskytuje väčšina súčasných shellov, aby bolo ich interaktívne použitie pohodlnejšie a efektívnejšie.

Interaktívny shell

524

- shell zobrazí výzvu - prompt a čaká
- užívateľ zadá príkaz a stlačí enter
- príkaz sa okamžite vykoná
- výstup sa vypíše na terminál

Interaktívny mód je charakterizovaný cyklom výzva-odpoveď, kde shell vyzve užívateľa, aby zadal príkaz, užívateľ zadá príkaz, potvrdí ho, a shell ho následne spustí. Štandardný výstup a štandardný chybový výstup (popisovače/deskriptory 1 a 2) a štandardný vstup (deskriptor 0) sú pripojené k terminálu, t.j. k displeju a klávesnici užívateľa.

Doplnenie príkazov (command completion)

525

- väčšina shellov vám dovoľuje použiť TAB pre automatické doplnenie
  - funguje to aspoň na názvy príkazov a názvy súborov
  - ale často je prítomné "chytré doplňanie"
- interaktívna história: stlačte 'hore' pre návrat k predchádzajúcemu príkazu
  - tiež inetraktívne vyhľadávanie v histórii, napr. pomocou `^R` v `bash`i

Počas interaktívneho použitia je významná časť času venovaná zadávaniu príkazov. Preto sa shelly pomerne usilovne snažia zredukovať úsilie, ktoré je potrebné na zadanie týchto príkazov. Jednou z prvých, a veľmi efektívnych, funkcií v tomto smere je 'tab completion' - dopĺňanie príkazov pomocou tabu, kde:

1. užívateľ zadá časť názvu príkazu alebo názvu súboru a stlačí tab,
2. shell vyhľadá všetky možné príkazy alebo názvy súborov s daným prefixom,
3. ak je iba jedna možnosť, doplní názov, inak ponúkne zoznam, cez ktorý užívateľ môže prejsť, alebo doplniť viac písmen, aby bol prefix unikátny.

Šetrí to čas dvomi rôznymi spôsobmi: za prvé, je často rýchlejšie stlačiť tab než zadať zvyšné písmená, a za druhé, užívateľ nemusí zadať extra príkaz na vymenovanie súborov, alebo na nájdenie presného názvu príkazu.
Okrem názvov príkazov a súborov veľa shellov ponúka 'inteligentné doplňanie', ktoré dokáže doplniť argumenty v závislosti od kontextu, t.j. v závislosti od prefixu príkazu, ktorý je zadávaný. Napríklad, napísanie `ifc^I ^I` (znak TAB je niekedy vyjadrený ako `^I`) môže najprv doplniť príkaz `ifconfig` (pre konfiguráciu sieťových rozhraní) a potom ponúknuť zoznam zariadení, ktoré sú dostupné pre konkrétny počítač.
Ďalšou významnou funkciou, ktorá šetrí písanie, je interaktívna história: keď užívateľ zadá príkaz, tento príkaz je uložený v 'súbore s históriou'. Posledných niekoľko príkazov sa dá jednoducho vyvolať stlačením šípky nahor, ale je tiež možné interaktívne prehľadávať históriu pomocou kľúčových slov. Logika je, že pre dlhšie príkazy môže byť editovanie existujúcich príkazov výrazne rýchlejšie, než opätovné napísanie celého príkazu.

Výzva - prompt

526

- reťazec zobrazený keď shell očakáva príkaz
- riadený premennou prostredia `PS1`
- obvykle zobrazuje vaše užívateľské meno a hostiteľské meno
- alebo pracovný adresár, stav batérie, čas, počasie, ...

Dôležitým nástrojom, ktorý pomáha užívateľovi zorientovať sa je výzva - prompt, ktorý slúži primárne ako indikátor toho, že shell je pripravený prijať príkaz.
Sekundárnou funkciou výzvy je dodať užívateľovi nejaké základné informácie: obvykle je prítomné hostiteľské meno (hostname) počítača (je veľmi jednoduché používať shelly vzdialene), meno užívateľa (login), pod ktorým pracuje, a aktuálny pracovný adresár. To, čo vypisujú, môže byť prispôsobené, a veľa shellov vie spúšťať ľubovoľné príkazy na vypočítanie výzvy (promptu), ktorý sa má zobraziť. V tom prípade môže výzva zahŕňať čokoľvek, čo sa vojde na riadok, vrátane aktuálneho času, stavu batérie, návratového kódu posledného príkazu, aktuálneho počasia, aktívnej `git` vetvy, aktuálneho vyťaženia CPU alebo pamäte, a tak ďalej.

Správa úloh

527

- iba jeden program môže bežať na popredí (terminál)
- ale bežiaci program môže byť pozastavený (`C-z`)
- a obnovený na pozadí (`bg` pre background) alebo na popredí (`fg`)
- použite `&` pre spustenie príkazu na pozadí: `./spambot &`

Hoci nie je správa úloh v moderných systémoch nevyhnutná, môže byť príležitostne užitočná. Pôvodnou motiváciou bolo, že typicky mal užívateľ iba jediný terminál s jednou obrazovkou, a teda mohol naraz spúšťať iba jeden príkaz: shell bol nedostupný, kým program neskončil (keďže štandardné IO programu bolo pripojené k terminálu).
Na zlepšenie tejto situácie umožňujú shelly, aby mohli byť programy vykonávané na pozadí, a pokračujú v interakcii s užívateľom zatiaľ čo program beží. Správa úloh umožňuje užívateľovi privolať príkazy bežiace na pozadí do popredia, pozastaviť program bežiaci v popredí, atď. Dnes už typicky nie je problém mať otvorených toľko terminálov, koľko užívateľ chce.

Terminál

528

- môže zobrazovať text a čítať text z klávesnice
- väčšinou je text pridávaný na poslednom riadku
- text môže obsahovať escape (riadiace) sekvencie
  - pre zobrazenie farebného textu alebo zmazanie obrazovky
  - tiež pre zobrazenie textu na daných súradniciach

Terminál samotný je kľúčovou časťou interakcie, hoci nie je súčasťou shellu samotného. Namiesto toho shell používa terminál pre svoj vstup a výstup, ako akýkoľvek iný textovo-orientovaný program. Hoci zvykli byť terminály hardvérové zariadenia, dnes je výrazne častejšie použitie 'emulátorov terminálov', programov, ktoré sa chovajú ako tradičný hardvérový terminál, ale jednoducho vykreslia obsah obrazovky do okna.
Pri normálnom použití terminálu sa starší text posúva nahor: toto je mód použitý v typickom shelli. Navyše, toto správanie je v termináli automatické. Naopak, celo-obrazovkové terminálové aplikácie automatické posúvanie textu vypínajú a namiesto toho používajú zobrazovanie podľa súradníc: to je dosiahnuté vypísaním špeciálnych 'escape sekvencií' na terminál, ktoré nie sú zobrazené ako doslovný text, ale namiesto toho kódujú inštrukcie pre terminál, napr. posunutie kurzora, alebo zobrazenie farebného textu.

Celo-obrazovkové terminálové aplikácie

529

- aplikácie môžu využívať celú obrazovku terminálu
- knižnica abstrahuje nízkoúrovňové riadiace sekvencie
  - knižnica sa volá `ncurses` pre `new curses`
  - rôzne terminály používajú rôzne riadiace sekvencie

Terminály sú zariadenia 'znakových buniek': obrazovka je rozdelená na neprekrývajúcu sa mriežku buniek a každá bunka môže zobraziť jeden znak alebo symbol. Terminály typicky umožňujú aplikáciám vypnúť automatický posuv textu a potom dať znaky kdekoľvek na obrazovku: tieto schopnosti umožňujú využitie obrazovky spôsobom špecifickým pre aplikáciu.
Napríklad, textový editor môže zobraziť časť súboru, ktorá je práve editovaná, a umožniť užívateľovi pohybovať sa hore a dolu v súbore, ako si praje. Toto použitie zjavne nezapadá do modelu kde je text zobrazený iba na posledný riadok a automaticky sa posúva nahor keď sa riadok zaplní alebo je vypísaný znak nového riadku.
Historicky používali rôzne terminály rôzne escape sekvencie pre rovnaké (alebo súvisiace) funkcie. Tieto funkcie sa tiež mierne líšili medzi dodávateľmi a dokonca aj medzi rôznymi modelmi terminálov. Preto knižnica s názvom `ncurses` prekladá vysokoúrovňové príkazy (na tieto súradnice daj červené 'a') na nízkoúrovňové sekvencie v závislosti od terminálu, ktorý aplikácia aktuálne používa.

UNIX textové editory

530

- `sed` – stream (prúdový) editor, neinteraktívny
- `ed` – práca po riadkoch, interaktívny
- `vi` – vizuálny, práca na celej obrazovke
- `ex` – režim `vi`, ktorý pracuje po riadkoch

Typickým príkladom celo-obrazovkového terminálového programu je textový editor. Nebolo to tak však vždy: prvým bežne používaným 'obrazovkovým' textovým editorom bolo `vi`.[10] Starší editor, `ed`, bol založený na príkazoch, a na zobrazenie časti súboru musel užívateľ zadať príkaz na to určený.

TUI: textové užívateľské rozhranie

531

- existujú špeciálne znaky na vykreslenie rámčekov a oddeľovačov
- program na terminál vykreslí 2D rozhranie
- tieto typy rozhraní môžu byť celkom pohodlné
- často sa programujú jednoduchšie než GUI
- veľmi nízke nároky na priepustnosť pri vzdialenom používaní

Použitím špeciálnej znakovej sady (a špeciálneho fontu) je možné vykresliť jednoduchú grafiku (obdĺžnikové rámčeky) na termináli. Celo-obrazovkové programy, ktoré využívajú takéto funkcie sú na polceste ku GUI, a často ponúkajú menu, formuláre s textovými poľami, zaškrtávacie polia (checkbox) alebo tlačidlá, dialógové okná a ďalšie prvky, ktoré môžeme bežne vídať v grafických programoch.

Časť 9.3: Grafické rozhrania

Samozrejme, moderné operačné systémy[11] ponúkajú grafické užívateľské rozhrania, založené na mriežke miliónov drobných pixelov, namiesto veľkých znakových buniek. Okrem klávesnice na zadávanie textu sú všadeprítomné ukazovacie zariadenia (myši, touchpady, dotykové obrazovky, ...). Zobrazovacie zariadenia založené na pixeloch dokážu zobrazovať ľubovoľné obrázky, ale užívateľské rozhrania tradične zostávajú pri jednoduchých, obdĺžnikových tvaroch.

Systémy založené na oknách (window)

533

- každá aplikácia beží vo svojom vlastnom okne
  - alebo prípadne niekoľkých oknách
- na obrazovke môže byť zobrazených niekoľko aplikácií
- okná môžu byť presúvané, môžeme im zmeniť veľkosť, ap.
  - je to umožnené rámčekmi okolo obsahu okien
  - všeobecne známe ako správa okien (window management)

Ústredným modelom pre staršie GUI systémy bolo okno, ktoré vyvinulo Xerox PARC v 70. rokoch a bolo adoptované do bežne používaných systémov od Apple, Microsoftu a Sun v 80. rokoch.
Systém dokáže zobraziť niekoľko aplikácií naraz, kde každá je obmedzená na svoje okno: obdĺžnikovú oblasť na obrazovke, ktorá sa dá presúvať, ktorej môžete zmeniť veľkosť a ktorá sa môže prekrývať s inými aplikáciami.

Bez-oknové systémy

534

- populárne najmä na menších obrazovkách
- aplikácie zaberajú celú obrazovku
  - okrem rozhrania pre správu a na zobrazovanie stavu
- prepínanie úloh cez dedikovanú obrazovku

[10] Existuje niekoľko obrazovkových editorov, ktoré predchádzali `vi`, avšak žiaden z nich neprežil špecifický hardvér a operačný systém, pre ktorý boli napísané. Na druhú stranu, klony `vi` sú aj v súčasnosti stále bežne používané.

[11] Aspoň tie, ktoré poskytujú nejakú úroveň podpory pre beh na univerzálnych koncových užívateľských zariadeniach, ako sú desktopy, laptopy, tablety alebo smartfóny.

Hoci systémy založené na oknách dominovali počítačovému svetu v 90. rokoch a krátko po roku 2000, toto sa začalo meniť v roku 2007, s príchodom prvého iPhonu. Hoci samozrejme nešlo o prvý smartfón, ani prvé zariadenie s malou obrazovkou, mal veľký vlpyv na svet počítačov. Malá obrazovka iPhonu spôsobila, že okná boli nepraktické a v podstate vzkriesila režim fungovania 'jedna aplikácia naraz'.
Samozrejme, podkladový operačný systém bol plne schopný súbežného vykonávania viacerých úloh, a počítač reálne vykonával niekoľko úloh na pozadí. Užívateľské rozhranie poskytuje možnosti interakcie s týmito úlohami (napr. upozornenia), ktoré majú súčasne aspekty jedno- aj viac-úlohových prostredí. Toto paradigma je dnes bežne používané na tabletových počítačoch a smartfónoch všetkých významných výrobcov.

GUI vrstva

535

- ovládač grafickej karty, nastavenie režimu
- vykresľovanie (obvykle hardvérovo akcelerované)
- multiplexovanie (napr. použitím okien)
- ovládacie prvky (widgets): tlačidlá, štítky, zoznamy, ...
- rozloženie (layout): čo má ísť kde na obrazovku

Vykreslenie mriežky znakových buniek je pomerne jednoduché: každé písmeno a symbol, ktoré môžu byť zobrazené dostane bitmapu s veľkosťou bunky mriežky. Obrázok na obrazovke je zlepený z neprekrývajúcej sa mriežky týchto malých obdĺžnikových bitmáp.
Na druhú stranu, grafická vrstva je výrazne zložitejšia. Hoci je pravdepodobne podkladový koncept: malé farebné obdĺžniky (pixely) zjavne jednoduchší než bunky znakov, proces vyskladania užitočných obrázkov z nich je výrazne zložitejší.

Známe GUI

536

- Windows
- macOS, iOS
- X11
- Wayland
- Android

Prenositeľnosť

537

- GUI 'toolkity' uľahčujú prenositeľnosť
  - Qt, GTK, Swing, HTML5+CSS, ...
  - veľa z nich beží na všetkých významných platformách
- prenositeľnosť kódu nie je jediným problémom
  - GUI majú pravidlá pre vzhľad a chovanie programov
  - prenositeľné aplikácie nemusia zapadnúť

Rôzne GUI vrstvy poskytujú rôzne API, rôzne abstrakcie a rôzne schopnosti. Keďže je prenositeľnosť softvéru žiaduca aj u GUI aplikácií, programátori často používajú toolkit, ktorý sa nachádza nad GUI a vytvára jednotnú abstrakciu. Vďaka tomu môže aplikácia bežať na rôznych GUI vrstvách jednoducho pomocou opätovného prekladu (rebuild). Niečo to však stojí: toolkity môžu byť veľmi komplikované (stovky tisíc riadkov kódu, v prípade webu ide o rádovo milióny).

Vykresľovanie textu

538

- prekvapivo zložitá úloha
- na rozdiel od terminálov, GUI používajú fonty s premenlivou šírkou
  - prináša to problémy ako kerning
  - ťažké predvídať šírku riadka v pixeloch
- zlé interagovanie s tlačou (→ WYSIWIG)

Bitmapové fonty

539

- znaky sú reprezentované ako polia pixelov
  - obvykle iba čierno-biele
- tradične ručné vykresľovanie po pixeloch
  - veľmi časovo náročné (veľa písmen, veľkostí, verzií)
- výsledok je ostrý ale zúbkovaný (nie je hladký)

Obrysové fonty

540

- Type1, TrueType – založené na splajnoch
- môžu byť škálované na ľubovoľnú veľkosť v pixeloch
- rovnaký font môže byť použitý pre obrazovky a pre tlač
- rasterizácia sa obvykle robí softvérovo

Hinting, Anti-Aliasing - vyhladzovanie

541

- obrazovky sú zariadenia s nízkym rozlíšením
  - typické HD displeje majú DPI okolo 100
  - laserové tlačiarne majú DPI 300 a viac
- hinting: deformuje obrysy, aby písmo lepšie sedelo do pixelovej mriežky
- anti-aliasing: vyhladí obrysy pomocou odtieňov šedej

X11 (oknový systém X)

542

- tradičný UNIX-ový oknový systém
- poskytuje C API (`xlib`)
- vstavaná sieťová transparentnosť (založená na soketoch)
- core protocol verzia 11 z roku 1987

Architektúra X11

543

- X server poskytuje grafiku a vstup
- X klient je aplikácia, ktorá používa X
- správca okien je (špeciálny) klient
- kompozítor je ďalší špeciálny klient

Vzdialené displeje

544

- aplikácia beží na počítači A
- displej nie je konzola A
  - môže ísť o dedikovaný grafický terminál
  - môže ísť o ďalší počítač na LAN
  - alebo dokonca cez internet

Protokoly pre vzdialené displeje

545

- jedným z prístupov je posielanie pixelov
  - VNC (Virtual Network Computing)
- X11 používa vlastný protokol na vykresľovanie
- ďalší používajú vysokoúrovňové abstrakcie
  - NeWS (založený na PostScripte)
  - HTML5 + JavaScript

VNC (Virtual Network Computing)

546

- po drôte posiela komprimované pixelové obrázky
  - môže využiť pravidelnosti v pixelových dátach
  - môže posielať inkrementálne aktualizácie
- a vstupné udalosti opačným smerom
- žiadna podpora pre periférie alebo synchronizáciu súborov

V podstate jedinou výhodou VNC je jednoduchosť. Na bezpečnosť sa začalo myslieť až neskôr a nie je úplne kompatibilná cez rôzne implementácie. Je navrhnuté najmä pre siete s nízkou priepustnosťou a vysokou latenciou (napr. internet).

RDP (Remote Desktop Protocol)

547

- viac sofistikovaný než VNC (ale proprietárny)
- tiež vie posielať príkazy na vykresľovanie po drôte
  - ako X11, ale používa na vykresľovanie DirectX
  - tiež umožňuje vzdialené OpenGL
- podpora pre audio, vzdialené USB ap.

RDP je založené primárne na modeli pretláčania pixelov, ale existuje množstvo rozšírení, ktoré umožňujú posielanie vysokoúrovňových vykresľovacích príkazov pre lokálne, hardvérovo akcelerované spracovanie. U niektorých nastavení to zahŕňa vzdialené akcelerované OpenGL a/alebo Direct3D.

SPICE

548

- Simple Protocol for Independent Computing Env.
- otvorený protokol niekde medzi VNC a RDP
- dokáže posielať OpenGL (ale iba cez lokálny soket)
- dvoj-cestné audio, USB, integrácia schránky
- ešte stále založené najmä na posielaní pixelov

SPICE = "Jednoduchý protokol pre nezávislé výpočetné prostredie"

Bezpečnosť vzdialenej obrazovky

549

- užívateľ musí byť autentizovaný po sieti
  - heslá sú jednoduché, biometrické dáta o niečo menej
- dátový prúd by mal byť šifrovaný
  - nie je súčasť protokolov X11 alebo NeWS
  - u HTTP nie je nutne zapnuté (používa sa pre HTML5/JS)

Napríklad, RDP na Windows 10 nepodporuje prihlasovanie cez odtlačky prstov (bolo to podporované u predchádzajúcich verzií, ale bolo to deaktivované kvôli bezpečnostným problémom.

Review Questions

550

33.What is a shell?
34.What does variable substitution mean?
35.What is an environment variable?
36.What belongs into the GUI stack?

Časť 10: Kontrola prístupu

Táto prednáška sa zameriava na základné bezpečnostné aspekty v operačnom systéme, špeciálne u súborových systémov, ktoré sú typicky najviditeľnejšou inštanciou kontroly prístupu (access control) v OS.

Obsah prednášky

552

1. Viac-užívateľské systémy
2. Súborové systémy
3. Granularita na úrovni menšej než užívateľ

Najprv sa pozrieme na motiváciu a implementáciu užívateľov, ako základnej jednotky vlastníctva a kontroly prístupu v operačnom systéme. Tiež sa pozrieme na niektoré dôsledky a možnosti použitia viac-užívateľských systémov, a rozoberieme si ako je kontrola prístupu implementovaná a vynucovaná. V druhej časti sa zameriame na kanonický prípad použitia kontroly prístupu: súborové systémy. Konečne, v poslednej časti sa oboznámime s tým, čo sa stane keď je kontrola prístupu na úrovni užívateľa nedostatočná a potrebujeme jemnejší systém prístupových práv.

Časť 10.1: Viac-užívateľské systémy

Viac-užívateľské systémy boli normou až do vzostupu osobných počítačov v cca polovici 80. rokov: počítače boli predtým príliš drahé a nes-kladné, než aby mohli byť alokované pre jednu osobu. Namiesto toho systémy predtým používali nejakú formu zdieľania (multi-tenancy), či už bola implementovaná administratívne (dávkové systémy) alebo operačným systémom (interaktívne počítače na princípe terminálov).

Užívatelia

554

- pôvodne zástupná definícia pre ľudia
- v súčasnosti ide o viac všeobecnú abstrakciu
- užívateľ je jednotkou vlastníctva
- veľa práv (permissions) je sústredených okolo užívateľa

Koncept užívateľa sa vyvinul z potreby udržiavať oddelené účty pre rôznych ľudí (eponymických užívateľov systému). V moderných systémoch užívateľ stále zosobňuje abstrakciu, ktorá zahŕňa účty pre jednotlivých ľudí, ale tiež pokrýva ďalšie potreby. Užívateľ je v podstate jednotkou vlastníctva a jednotkou kontroly prístupu.

Zdieľanie počítača

555

- počítač je (často drahý) zdroj
- efektivita využitia je dôležitá
  - jeden užívateľ zriedka plne využíva počítač
- zdieľanie dát spôsobuje, že kontrola prístupu je nevyhnutná

Hoci je efektívne využitie zdrojov to, čo motivovalo zdieľané využitie počítačových systémov, nutnosť vzniku kontroly prístupu spôsobil globálny zdieľaný súborový systém: užívatelia si nutne neprajú dôverovať všetkým ostatným užívateľom systému s prístupom k ich súborom.

Vlastníctvo

556

- rôzne objekty v OS môžu byť vlastnené (owned)
  - najmä súbory a procesy
- vlastníkom je typicky ten, čo objekt vytvoril
  - hoci vlastníctvo môže byť prevedené
  - obvykle sa na to vzťahujú obmedzenia

Štandardný model kontroly prístupu v operačnom systéme súvisí s vlastníctvom objektov (object ownership). Všeobecne vzaté, vlastníctvo objektu udeľuje práva (manipulovať s objektom) ale aj záväzky (vlastnené objekty sa započítavajú do kvóty). V závislosti od okolností môže byť vlastníctvo objektu prípadne prevedené, buď pôvodným vlastníkom, alebo administrátormi systému.

Vlastníctvo procesu

557

- každý proces patrí nejakému užívateľovi
- proces vykonáva úlohy v zastúpení užívateľa
  - proces dostane rovnaké privilégiá ako jeho vlastník
  - to obmedzuje proces a zároveň mu dáva práva
- procesy sú aktívnymi účastníkmi

Asi najdôležitejším vlastníckym vzťahom je ten medzi užívateľmi a ich procesmi. Je to preto, že procesy vykonávajú kód v zastúpení užívateľa, a všetky akcie, ktoré užívateľ vykoná v systéme sú sprostredkované nejakým procesom. V tomto zmysle procesy konajú v zastúpení svojho vlastníka a akcie, ktoré vykonávajú, podliehajú akýmkoľvek obmedzeniam, ktoré sa na daného užívateľa vzťahujú.

Vlastníctvo súborov

558

- každý súbor tiež patrí nejakému užívateľovi
- dáva to užívateľom práva (alebo skôr ich procesom)
  - môžu čítať a zapisovať do súboru
  - môžu zmeniť práva alebo vlastníctvo
- súbory sú pasívnymi účastníkmi

Podobne ako procesy, súbory sú objekty, ktoré podliehajú vlastníctvu. Avšak, na rozdiel od procesov, súbory sú pasívne: nevykonávajú žiadne akcie. Teda v tomto prípade vlastníctvo iba dáva užívateľovi určité práva vykonávať akcie na súbore (čo je najdôležitejšie, môže zmeniť práva kontroly prístupu, ktoré sa vzťahujú na daný súbor).

Modely riadenia prístupu

559

- obvykle o tom, kto môže pristupovať k ich objektom, rozhodujú vlastníci
  - toto je známe ako tzv. discretionary access control – voliteľný
- v prostrediach s vysokou mierou bezpečnosti to nie je dovolené
  - známe ako mandatory access control – povinný
  - centrálna autorita rozhoduje o politike prístupu

Existujú dva hlavné postoje čo sa týka kontroly prístupu: rozšírený discretionary model (voliteľný, v zmysle vlastník má voľbu toho, komu objekt sprístupní), kde sa vlastníci rozhodujú, kto môže interagovať s ich súbormi (alebo inými objektami, podľa toho, čo je relevantné) a tzv. mandatory model – povinný, v ktorom sa užívateľom nedôveruje s bezpečnostnými záležitosťami a rozhodnutia o kontrole prístupu sú vložené do rúk centrálnej autority.
V oboch prípadoch operačný systém udeľuje (alebo zamieta) prístup k objektu, v závislosti od politiky kontroly prístupu: avšak iba v druhom prípade sa dá táto politika chápať ako jednoliaty, sebestačný dokument (v porovnaní s kolekciou pravidiel, o ktorých rozhoduje množstvo nekoordinovaných užívateľov).

(Virtuálni) systémoví užívatelia

560

- užívatelia sú užitočnou abstrakciou vlastníctva
- rôzne systémové služby majú vlastných ‘falošných’ užívateľov
- to im umožňuje vlastniť súbory a procesy
- a tiež obmedziť ich prístup ku zvyšku OS

Ukázalo sa, že užívatelia sú veľmi užitočnou abstrakciou. Je bežnou praxou, že služby (či už systémové alebo aplikačné) bežia pod vlastnými špeciálnymi užívateľmi. To znamená, že tieto služby môžu vlastniť súbory a ďalšie zdroje, a spúšťať procesy pod svojou vlastnou identitou. Ďalej to znamená, že tieto služby môžu byť obmedzené použitím rovnakých mechanizmov, ktoré sa vzťahujú na ‘normálnych’ užívateľov.

Princíp najmenších privilégií

561

- entity by mali mať najmenšie možné požadované privilégiá
  - vzťahuje sa to na softvérové komponenty
  - ale tiež na ľudských používateľov systému
- obmedzuje to rozsah chýb
  - a tiež bezpečnostných narušení systému

Princíp najmenších privilégií je dôležitou zásadou pre návrh bezpečných systémov: hovorí nám, že bez ohľadu na kombináciu subjektu a objektu by mali byť práva pridelené iba ak existuje skutočná potreba subjektu manipulovať s konkrétnym objektom. Odôvodnením tohto princípu je, že chyby sa stávajú a keď k nim dôjde, chceme obmedziť ich rozsah (a teda škody): chyby nemôžu ohroziť objekty, ktoré sú neprístupné páchateľovi.

Oddelenie privilégií

562

- rôzne časti systému potrebujú rôzne privilégiá
- princíp najmenších privilégií vyžaduje rozdelenie systému
  - komponenty sú od seba navzájom izolované
  - sú im dané iba práva, ktoré potrebujú
- komponenty komunikujú prostredníctvom veľmi jednoduchého IPC

Dôležitým dôsledkom princípu najmenších privilégií je návrhový vzor známy ako oddelenie privilégií (privilege separation). Systémy, ktoré sa ním riadia, sú rozdelené do množstva nezávislých komponent, kde každá vykonáva malú, dobre definovanú a bezpečnostne oddeliteľnú funkciu. Každý z týchto modulov môže potom byť izolovaný do svojho vlastného malého sandboxu a komunikovať so zvyškom systému cez starostlivo definované rozhrania (obvykle postavené na nejakej forme medzi-procesovej komunikácie).

Oddelenie procesov

563

- spomeňte si, že každý proces beží vo svojom vlastnom adresnom priestore
  - zdieľaná pamäť musí byť explicitne vyžiadaná
- každý užívateľ vidí na súborový systém
  - štandardne je veľa vecí v súborovom systéme zdieľaných
  - obzvlášť menný priestor (adresárová hierarchia)

V pamäti nie je veľká potreba pre kontrolu prístupu: každý proces má svoju vlastnú a nemôže vidieť pamäť žiadneho iného procesu (s malými, kontrolovanými výnimkami, ktoré vznikajú vzájomnou dohodou dvoch procesov).
Súborový systém je však veľmi odlišný: má globálny, zdieľaný menný priestor, ktorý je viditeľný pre všetkých užívateľov a pre všetky procesy. Navyše, veľa objektov (súborov) má byť zdieľaných, pomerne ad-hoc spôsobom, buď cez 'známe' cesty – paths (čo je prípad mnohých súborových systémov) alebo cez posielanie si ciest. Čo je dôležité, cesty nie sú žiadnou formou tokenu, ktorý dovoľuje prístup, a v takmer všetkých prípadoch platí, že zatajenie cesty k súboru nijak nebráni prístupu k objektu (cesty sa dajú jednoducho zistiť).

Politika kontroly prístupu

564

- sú prítomné 3 časti informácie
  - subjekt (užívateľ)
  - akcia/sloveso (čo sa má urobiť)
  - objekt (súbor alebo iný zdroj)
- existuje veľa spôsobov ako zakódovať tieto informácie

Ako sme už spomenuli, súbor pravidiel, ktoré rozhodujú o tom, ktoré akcie sú dovolené, a ktoré nie sú dovolené, je známy ako politika kontroly prístupu (access control policy). Všeobecne vzaté, ide o súbor pravidiel, ktorý odpovedá na otázky vo forme 'Má (subjekt) dovolené vykonať (akciu) na (objekte)?' Zjavne existuje veľa rôznych možností, ako môže byť táto 'kniha pravidiel' zakódovaná: na niektoré z najbežnejších stratégií sa neskôr pozrieme.

Prístupové práva: Subjekty

565

- v typickom OS ide o (prípadne virtuálnych) užívateľov
  - sú možné jednotky menšie než užívateľ (napr. programy)
  - roly a skupiny môžu tiež byť subjekty
- subjekt musí byť pomenovaný (názvy/mená, identifikátory)
  - jednoduché pri jednom systéme, zložité na sieti

Najčastejším subjektom kontroly prístupu (aspoň pokiaľ ide o špecifikáciu politiky prístupu), sú, ako už bolo naznačené, užívatelia, či už 'reálni' (takí, ktorí zastupujú ľudí) alebo virtuálni (ktorí zastupujú služby).
Vo väčšine prípadov musí byť možné pomenovať subjekty, aby sme na nich mohli referovať v pravidlách. Niekedy však môžu byť pravidlá priamo pripojené k subjektom, v tom prípade tieto subjekty nepotrebujú mať so sebou spätý stabilný identifikátor.

Prístupové práva: Akcie (slovesá)

566

- dostupné 'slovesá' (akcie) závisia na type objektu
- typickým objektom je súbor
  - súbory môžu byť čítané, zapisované, spúšťané
  - adresáre môžu byť prehľadávané alebo vypísané alebo zmenené
- sieťové spojenia môžu byť ustanovené ap.

Konkrétne voľby akcie závisia na type objektu: každý takýto typ má fixný zoznam akcií, ktoré odpovedajú operáciám, alebo typom operácií, ktoré operačný systém ponúka skrze svoje rozhrania.
Akcie môžu byť ovplyvnené politikou priamo alebo nepriamo – napríklad, právo čítania na súbore nie je vynucované v čase keď sa zavolá `read`: namiesto toho je skontrolované v čase volania `open` (pri otváraní súboru), s predpokladom, že `read` sa môže použiť na popisovačoch súboru (file descriptors), iba ak sú otvorené na čítanie. To znamená, že program musí uviesť, v čase vykonania `open`, či si praje čítať zo súboru.

Prístupové práva: Objekty

567

- všetko, čo môže byť manipulované programami
  - hoci nie všetko podlieha kontrole prístupu
- môže ísť o súbory, adresáre, sokety, zdieľanú pamäť, ...
- názvy objektov závisia od ich typu
  - cesty k súborom, číslo i-uzla, IP adresy, ...

Podobne ako subjekty, aj objekty musia mať mená, pokiaľ nie sú časti politiky, ktoré sa na ne vzťahujú priamo pripojené k samotným objektom. V prípade objektov je však toto priame pripojenie výrazne častejšie: je typické, že i-uzol uchováva informácie o právach.

Subjekty v POSIX-e

568

- existujú 2 typy subjektov: užívatelia a skupiny
- každý užívateľ môže patriť do viacerých skupín
- užívatelia sa delia na normálnych užívateľov a `root`
  - `root` je tiež známy ako super-užívateľ (super-user)

V POSIX-ových systémoch sú prítomné dva základné typy subjektov, ktoré sa môžu objaviť v politike kontroly prístupu: užívatelia a skupiny. Keďže POSIX obsahuje kontrolu prístupu iba pre súborový systém, objekty nepotrebujú byť pomenované: ich práva sú pripojené k i-uzlu. Špeciálny užívateľ, známy ako `root` reprezentuje administrátora sys-

tému (tiež známy ako super-užívateľ/super-user). Tento účet nepodlieha kontrole práv. Existuje niekoľko akcií (obvykle nie sú späté s konkrétnym objektom), ktoré môže vykonávať iba root (napr. rebootovať počítač).

Identifikátory užívateľov a skupín

569

- užívatelia a skupiny sú reprezentovaní ako čísla
  - zlepšuje to efektivitu mnohých operácií
  - tieto čísla sa nazývajú uid a gid
- tieto čísla sú validné iba na jednom počítači
  - alebo nanajvýš na lokálnej sieti

V politike kontroly prístupu sú užívatelia a skupiny identifikovaní číslami (každý užívateľ a každá skupina dostane malé, lokálne unikátne celé číslo). Keďže majú tieto identifikátory fixnú veľkosť, môžu byť veľmi kompaktne uložené v i-uzloch, a tiež môžu byť veľmi efektívne porovnávané, pričom obe z týchto kritérií boli historicky dôležité. Okrem efektivity číselné identifikátory tiež zjednodušujú schému dátových štruktúr, ktoré ich nesú, čím zmenšujú priestor pre chyby.

Správa užívateľov

570

- systém potrebuje databázu užívateľov
- na sieti musia byť často identity užívateľov zdieľané
- môže to byť aj obyčajný textový súbor
  - /etc/passwd a /etc/group na UNIX-ových systémoch
- ale môže to byť aj také zložité ako distribuovaná databáza

Databáza užívateľov má dve základné roly: hovorí systému ktorí užívatelia sú oprávnení pristupovať k systému (o tomto viac neskôr), a mapuje ľudsky-čitateľné užívateľské mená na číselné identifikátory, ktoré systém interne používa.
V lokálnych sieťach je často žiaduce, aby všetky počítače mali rovnakú predstavu o tom, kto sú užívatelia, a aby používali rovnaké mapovanie medzi ich menami a číselnými identifikátormi. LDAP a Active Directory sú populárne voľby pre centralizovanú databázu užívateľov na lokálnej sieti.

Zmena vlastníctva

571

- každý proces patrí nejakému užívateľovi
- vlastníctvo sa dedí skrze fork()
- procesy super-užívateľa môžu používať setuid()
- exec() môže niekedy zmeniť vlastníka procesu

Spomeňte si, že všetky procesy sú vytvorené pomocou systémového volania fork, s výnimkou init. Keď proces vykoná fork, detský proces zdedí vlastníka rodiča, to znamená, že patrí rovnakému užívateľovi ako rodičovský proces (ktorého vlastníctvo nie je nijak ovplyvnené volaním fork).
Keď je však proces vlastnený super-užívateľom, môže zmeniť svojho vlastníka pomocou systémového volania setuid. Navyše, exec môže tiež niekedy zmeniť vlastníka procesu, cez takzvaný setuid bit (ktorý si netreba zamieňať so systémovým volaním s rovnakým názvom). Proces init je vlastnený super-užívateľom.

Prihlásenie (login)

572

- proces super-užívateľa spravuje prihlásenia užívateľov
- užívateľ zadá svoje meno a heslo
  - login program autentizuje užívateľa
  - potom zavolá setuid() aby zmenil vlastníka procesu
  - a použije exec() aby spustil shell pre užívateľa

Možno si spomeniete, že na konci procesu bootovania sa spustí login proces, aby umožnil užívateľovi autentizovať sa vytvoriť sedenie (reláciu). V tradičnej implementácii sa login najprv spýta užívateľa na jeho užívateľské meno a heslo, ktoré overí v databáze užívateľov. Ak údaje súhlasia, program login nastaví základné prostredie, zmení vlastníka procesu na užívateľa, ktorý sa práve autentizoval a spustí jeho preferovaný shell (podľa konfigurácie v databáze užívateľov).

Autentizácia užívateľov

573

- užívateľ sa musí autentizovať
- najčastejšie používanou metódou sú heslá
  - systém musí rozpoznať správne heslo
  - užívateľ by si mal byť schopný zmeniť heslo
- biometrické metódy sú tiež dosť populárne

Výrazne najčastejšou metódou autentizácie užívateľov (t.j. potvrdenie, že sú tí, za ktorých sa vydávajú) je opýtať sa ich na nejaké tajomstvo – heslo zložené zo slova alebo frázy. Myšlienka je, že iba legitímny vlastník konkrétneho účtu pozná toto tajomstvo.
V ideálnom prípade systém neukladá heslo samotné (v prípade, že by došlo ku krádeži databázy), ale namiesto toho ukladá informácie, ktoré sa dajú použiť na overenie, že heslo, ktoré užívateľ zadal je správne. Obvykle sa toto robí pomocou kryptografických hashovacích funkcií s pridanou tzv. soľou.
Okrem hesiel existujú ďalšie spôsoby autentizácie, napríklad kryptografické tokeny a biometrika.

Vzdialené prihlasovanie

574

- autentizácia po sieti je komplikovanejšia
- heslá sú najjednoduchšie, ale nie jednoduché
  - na bezpečný prenos hesiel je potrebné šifrovanie
  - spolu s autentizáciou počítača
- populárnym vylepšením je 2-faktorová autentizácia

Hoci je heslo jednoducho krátky reťazec, ktorý sa dá pomerne jednoducho poslať po sieti, má to isté problémy. Za prvé, sieť samotná často nie je bezpečná, a heslo by mohlo byť odpočuté útočníkom. To znamená, že na prenos hesla musíme použiť kryptografiu, alebo jeho znalosť overiť inak.
Ďalším problémom je, že v prípade, že pošleme šifrované heslo, počítač na druhom konci nemusí byť ten, ktorý očakávame (t.j. môže patriť útočníkovi).
Keďže užívateľ nemusí byť fyzicky prítomný, aby sa pokúsil autentizovať, výrazne to zvyšuje riziko útokov, takže je o to dôležitejšie používať silné heslá. Okrem silných hesiel môže byť bezpečnosť zlepšená 2-faktorovou autentizáciou (o tomto si povieme za chvíľu).

Autentizácia počítačov

575

- ako zabezpečiť, že posielame heslo správnej strane?
  - útočník sa môže vydávať za náš vzdialený počítač
- zvyčajne pomocou asymetrickej kryptografie
  - súkromný kľúč môže byť použitý na podpisovanie správ
  - server podpíše výzvu, aby potvrdil svoju identitu

Pri interakcii so vzdialeným počítačom (cez sieť) je pomerne dôležité zabezpečiť, že komunikujeme s počítačom, s ktorým sme zamýšľali. Hoci je bezprostredným problémom posielanie hesiel, nie je to, samozrejme, jediný problém: rovnako zlé by bolo omylom poslať tajné dáta nesprávnemu počítaču, ak nie horšie.
Bežný prístup je, že každý počítač dostane unikátny súkromný kľúč, zatiaľ čo jeho verejný náprotivok (alebo aspoň jeho odtlačok-- fingerprint) je distribuovaný na ostatné počítače. Pri pripájaní môže klient vygenerovať náhodnú výzvu, a požiadať vzdialený počítač, aby ju podpísal použitím súkromného kľúča, ktorý je spätý s počítačom, ktorý sme chceli kontaktovať, aby potvrdil svoju identitu. Za predpokladu, že cieľový počítač samotný nebol kompromitovaný, útočník nebude schopný vytvoriť validný podpis a jeho útok bude zmarený.

2-faktorová autentizácia

576

- 2 rôzne typy autentizácie
  - ťažšie úspešne zaútočiť na oba naraz
- možno zvoliť z niekoľkých faktorov
  - niečo, čo užívateľ vie (heslo)
  - niečo, čo užívateľ má (kľúče, tokeny)
  - niečo, čo užívateľ je (biometrika)

Dvoj-faktorová (alebo viac-faktorová) autentizácia je populárna u vzdialenej autentizácie (ako sme už skôr spomenuli), lebo po sieti sú útoky výrazne lacnejšie a častejšie. V tomto prípade je prvým faktorom obvykle heslo, a druhým faktorom je kryptografický token – malé zariadenie (často vo forme kľúčenky), ktoré generuje unikátnu sekvenciu kódov, z ktorých jeden užívateľ prepíše do systému, aby potvrdil vlastníctvo tokenu. Vzdialená autentizácia cez biometriku je trochu menej praktická (aj keď nie nemožná).
Dvoj-faktorová autentizácia môže samozrejme byť použitá aj lokálne, v tom prípade sa stáva biometrika výrazne atraktívnejšou. Kryptografické tokeny alebo smart karty sú však tiež časté, hoci v lokálnom prípade obvykle komunikujú s počítačom priamo, než aby sa spoliehali na to, že užívateľ skopíruje kód.

Vynucovanie: hardvér

577

- všetko vynucovanie začína hardvérom
  - CPU poskytuje privilegovaný režim kernelu
  - DMA pamäť a IO inštrukcie sú chránené
- MMU umožňuje kernelu izolovať procesy
  - a chrániť jeho vlastnú integritu

Keď už máme zavedenú politiku kontroly prístupu a overili sme identitu užívateľa, potrebujeme vyriešiť ešte jednu vec, a tou je vynucovanie tejto politiky. Politika kontroly prístupu je samozrejme zbytočná, keď sa dá obísť.
Schopnosť operačného systému vynucovať bezpečnosť vyvstáva zo schopností hardvéru: softvér samotný nemôže dostatočne obmedziť ďalší softvér bežiaci na rovnakom počítači. Hlavným nástrojom, ktorý umožňuje kernelu vynucovať svoju bezpečnostnú politiku je MMU (a fakt, že iba kernel ju môže programovať) a kontrola nad obsluhou prerušení.

Vynucovanie: kernel

578

- kernel využíva na implementáciu bezpečnosti funkcie hardvéru
  - nachádza sa medzi zdrojmi a procesmi
  - prístup je sprostredkovaný pomocou systémových volaní
- súborové systémy sú súčasťou kernelu
- abstrakcie užívateľov a skupín sú súčasťou kernelu

Hardvérové zdroje sú spravované kernelom: pamäť prostredníctvom MMU, procesory cez prerušenia časovača, pamäťovo-mapované periférie opäť cez MMU a cez tabuľku obslúh prerušení. Keďže užívateľské programy nemôžu pristupovať k fyzickým zdrojom priamo, všetky interakcie s nimi musia ísť cez kernel (pomocou systémových volaní), čo poskytuje príležitosť pre kernel, aby mohol skontrolovať požadované akcie voči politike.

Vynucovanie: systémové volania

579

- kernel sa chová ako arbiter
- proces je uväznený vo svojom vlastnom adresnom priestore
- procesy pre prístup ku zdrojom využívajú systémové volania
  - kernel sa môže rozhodnúť čo dovolí
  - v závislosti od svojho modelu kontroly prístupu a politiky

Keď sa zavolá systémové volanie, kernel pozná vlastníka tohto procesu, a tiež všetky objekty, ktorých sa systémové volanie týka. S pomocou týchto znalostí môže jednoducho konzultovať politiku kontroly prístupu, aby sa rozhodol, či je požadovaná akcia dovolená, a ak nie je, vrátil procesu chybu, namiesto vykonania danej akcie.

Vynucovanie: API služieb

580

- užívateľské procesy môžu vynucovať kontrolu prístupu
  - obvykle systémové služby, ktoré poskytujú IPC API
- napr. cez systémové volanie `getpeereid()`
  - povie volajúcemu ktorý užívateľ je pripojený k soketu
  - kontrola prístupu na užívateľskej úrovni sa spolieha na funkcie kernelu

Rovnako ako kernel rozhoduje o prostriedkoch, ku ktorým nemôžu užívateľské programy pristupovať priamo, rovnaký princíp možno aplikovať na programy v užívateľskom priestore, najmä služby.
Pravdepodobne najviac názorným príkladom je relačná databáza: databázový engine beží pod dedikovaným (virtuálnym) užívateľom a ukladá svoje dáta do kolekcie súborov. Práva na týchto súboroch sú nastavené tak, že iba vlastník ich môže čítať a zapisovať do nich – kernel nedovolí žiadnemu inému procesu priamo interagovať s týmito súbormi.
Databázový systém však môže selektívne dovoliť ďalším programom interagovať s dátami, ktoré sú v ňom uložené: programy sa pripájajú k databázovému serveru pomocou UNIX-ového soketu. V tomto bode môže databáza požiadať operačný systém, aby jej poskytol id užívateľa, pod ktorým klient beží (použitím `getpeereid`).
Keďže server môže priamo pristupovať k súborom, v ktorých sú dáta uložené, môže teda v mene klienta spúšťať dotazy (databázové – queries) a vracať výsledky. Tiež však môže zamietnuť niektoré databázové

dotazy na základe svojej vlastnej politiky kontroly prístupu a užívateľského id klienta.

Časť 10.2: File Systems

File Access Rights

582

- file systems are a case study in access control
- all modern file systems maintain permissions
  - the only extant exception is FAT (USB sticks)
- different systems adopt different representation

Representation

583

- file systems are usually object-centric
  - permissions are attached to individual objects
  - easily answers "who can access this file"?
- there is a fixed set of verbs
  - those may be different for files and directories
  - different systems allow different verbs

The UNIX Model

584

- each file and directory has a single owner
- plus a single owning group
  - not limited to those the owner belongs to
- ownership and permissions are attached to i-nodes

Access vs Ownership

585

- POSIX ties ownership and access rights
- only 3 subjects can be named on a file
  - the owner (user)
  - the owning group
  - anyone else

Access Verbs in POSIX File Systems

586

- read: read a file, list a directory
- write: write a file, link/unlink i-nodes to a directory
- execute: exec a program, enter the directory
- execute as owner (group): `setuid`/`setgid`

Permission Bits

587

- basic UNIX permissions can be encoded in 9 bits
- 3 bits per 3 subject designations
  - first comes the owner, then group, then others
  - written as e.g. `rwxr-x--` or `0750`
- plus two numbers for the owner/group identifiers

Changing File Ownership

588

- the owner and `root` can change file owners
- `chown` and `chgrp` system utilities
- or via the C API
  - `chown()`, `fchown()`, `fchownat()`, `lchown()`
  - same set for `chgrp`

Changing File Permissions

589

- again available to the owner and to `root`
- `chmod` is the user space utility
  - either numeric argument: `chmod 644 file.txt`
  - or symbolic: `chmod +x script.sh`
- and the corresponding system call (numeric-only)

`setuid` and `setgid`

590

- special permissions on executable files
- they allow `exec` to also change the process owner
- often used for granting extra privileges
  - e.g. the `mount` command runs as the super-user

Sticky Directories

591

- file creation and deletion is a directory permission
  - this is problematic for shared directories
  - in particular the system `/tmp` directory
- in a sticky directory, different rules apply
  - new files can be created as usual
  - only the owner can unlink a file from the directory

Access Control Lists

592

- ACL is a list of ACE's (access control elements)
  - each ACE is a subject + verb pair
  - it can name an arbitrary user
- ACL is attached to an object (file, directory)
- more flexible than the traditional UNIX system

ACLs and POSIX

593

- part of POSIX.1e (security extensions)
- most POSIX systems implement ACLs
  - this does not supersede UNIX permission bits
  - instead, they are interpreted as part of the ACL
- file system support is not universal (but widespread)

Device Files

594

- UNIX represents devices as special i-nodes
  - this makes them subject to normal access control
- the particular device is described in the i-node
  - only a super-user can create device nodes
  - users could otherwise gain access to any device

Sockets and Pipes

595

- named sockets and pipes are just i-nodes
  - also subject to standard file permissions
- especially useful with sockets
  - a service sets up a named socket in the file system
  - file permissions decide who can talk to the service

Special Attributes

596

- flags that allow additional restrictions on file use
  - e.g. immutable files (cannot be changed by anyone)
  - append-only files (for logfile integrity protection)
  - compression, copy-on-write controls
- non-standard (Linux `chattr`, BSD `chflags`)

Network File System

597

- NFS 3.0 simply transmits numeric `uid` and `gid`
  - the numbering needs to be synchronised
  - can be done via a central user database
- NFS 4.0 uses per-user authentication
  - the user authenticates to the server directly
  - filesystem `uid` and `gid` values are mapped

File System Quotas

598

- storage space is limited, shared by users
  - files take up storage space
  - file ownership is also a liability
- quotas set up limits space use by users
  - exhausted quota can lead to denial of access

Removable Media

599

- access control at file system level makes no sense
  - other computers may choose to ignore permissions
  - user names or id's would not make sense anyway
- option 1: encryption (for denying reads)
- option 2: hardware-level controls
  - usually read-only vs read-write on the entire medium

The `chroot` System Call

600

- each process in UNIX has its own root directory
  - for most, this coincides with the system root
- the root directory can be changed using `chroot()`
- can be useful to limit file system access
  - e.g. in privilege separation scenarios

Uses of `chroot`

601

- `chroot` alone is not a security mechanism
  - a super-user process can get out easily
  - but not easy for a normal user process
- also useful for diagnostic purposes
- and as lightweight alternative to virtualisation

Časť 10.3: Granularita na úrovni menšej než užívateľ

V tejto sekcii sa pozrieme na niekoľko prípadov, kedy je potrebný (alebo aspoň užitočný) precíznejšie definovaný subjekt (v kontexte kontroly prístupu).

Užívatelia nestačia

603

- užívatelia nie sú vždy správnou abstrakciou
  - vytváranie užívateľov je pomerne drahé
  - iba super-užívateľ môže vytvoriť nových užívateľov
- môžete chcieť pridať možnosť programov ako subjektov
  - alebo skôr kombináciu užívateľ + program

Jednou z hlavných nevýhod bezpečnostného návrhu, ktorý sa sústredí na užívateľov je, že je ťažkopádny a vyžaduje privilégiá super-užívateľa. Navyše, normálni užívatelia nemôžu jednoducho vymedziť procesy aby bežali pod pomocnými užívateľmi (iba cez pomocný program s nastaveným `setuid` bitom, ktorý opäť musí byť nakonfigurovaný užívateľom `root`).
Prirodzeným rozšírením konceptu subjektu v kontrole prístupu je zahrnúť aktuálne bežiaci program do popisu – čo umožní, aby politika mohla obsahovať veci ako: ku `/home/xuser/mail` môže pristupovať thunderbird (mailový klient) bežiaci pod účtom `xuser`, ale nie firefox (webový prehliadač) bežiaci pod tým istým účtom.

Pomenovanie programov

604

- užívatelia majú užívateľské mená, ale čo programy?
- možnosť 1: kryptografické podpisy
  - prenositeľné cez rôzne počítače, ale zložité
  - ustanoví identitu na základe samotného programu
- možnosť 2: i-uzol spustiteľného súboru
  - jednoduché, lokálne, identita založená na umiestnení

Nanešťastie, priraďovať pravidlá politiky programom je výrazne ťažšie, než súborom alebo užívateľom, keďže ich identita je pomerne nejednoznačná. Môže existovať ľubovoľné množstvo programov, ktoré sa volajú thunderbird, z ktorých niektoré môžu byť iné verzie alebo zostavenia (build) rovnakého softvéru, ale niektoré môžu iba tvrdiť, že sú thunderbird, aby sa dostali do vášho emailu.
Relatívne dobrým, hoci komplikovaným riešením je zakomponovať

do spustiteľných súborov kryptografický podpis, ktorý zhruba hovorí ‘tento program je Firefox, podpísaný Mozillou’. Za predpokladu, že dôverujeme Mozille (asi dôverujeme, keďže používame ich softvér), môžeme v našej politike kontroly prístupu odkazovať na ‘Firefox od Mozilly’. Variácia tohto prístupu je použitá v mobilných operačných systémoch, ako Android a iOS.
Ďalšou možnosťou, výrazne jednoduchšou, je pridať poznámku typu ‘tento program je Firefox’ do i-uzla spustiteľného súboru. Tento prístup sa používa v systémoch ako SELinux (kde je táto poznámka realizovaná ako bezpečnostný štítok – security label).

Program ako subjekt

605

- program: pasívny (súbor) vs aktívny (procesy)
  - iba proces môže byť subjekt
  - ale identita programu patrí k súboru
- práva procesu závisia na jeho programe
  - `exec()` zmení privilégiá

Keď sme si už zvládli vymedziť čo je to program a ako ho identifikovať, nastáva nový problém: v oboch prípadoch sme priradili identitu súboru, ale v skutočnosti patrí procesu. Keďže sú však procesy výrazne dynamickejšie ako súbory, priraďovať im identifikátory je ešte menej praktické. V tomto prípade môžeme použiť rovnaký trik, aký sa používal pre `setuid` programy: systémové volanie `exec` môže skontrolovať binárku a podľa toho upraviť privilégiá procesu.

Mandatory Access Control – povinná kontrola prístupu

606

- deleguje kontrolu práv centrálnej autorite
- často sa používa spolu s bezpečnostnými štítkami
  - klasifikuje subjekty (užívateľov, procesy)
  - a tiež objekty (súbory, sokety, programy)
- vlastník nemôže zmeniť práva objektu

Bezpečnostné štítky (security labels) sú v istom zmysle zovšeobecnením užívateľských skupín. Môžu byť pripojené k objektom aj subjektom, a `exec` aktualizuje štítky pripojené k procesu podľa štítkov, ktoré sú pripojené k spustiteľnému súboru.
Pri povinnej kontrole prístupu (MAC) užívatelia nemajú dovolené meniť práva na objektoch. V praxi sú však oba módy v systémoch zvyčajne kombinované: voliteľné práva (discretionary permissions) sú pripojené k súborom ako obvykle, a aplikujú sa na akciu v prípade, že by to povinné (mandatory) pravidlá samotné boli dovolili.

Schopnosti – capabilities

607

- nie všetky slovesá (akcie) potrebujú brať objekty
- napr. vypnutie počítača (máme iba jeden)
- pripájanie (mounting) súborových systémov (nemôžu byť vždy pomenované)
- počúvanie na portoch s číslom menším než 1024

Termín ‘schopnosti’ (capabilities) sa často používa v jednom z dvoch významov formy pravidiel pre politiku kontroly prístupu:

1. keď je objekt tzv. singleton (jedináčik), t.j. existuje iba jediný objekt pre danú akciu, alebo
2. keď je nepraktické pomenovať objekty alebo pripojiť k nim informácie o právach.

Analýza užívateľa `root`

608

- tradičný užívateľ `root` je všemohúci
  - “všetko alebo nič” je často neuspokojivé
  - porušuje to princíp najmenších privilégií
- veľa špeciálnych vlastností užívateľa `root` sú schopnosti – capabilities
  - `root` sa stáva užívateľom so všetkými schopnosťami
  - ostatní užívatelia môžu dostať selektívne privilégiá

V mnohých prípadoch je jednoduché rozdelenie na `root`-a a normálnych užívateľov (čo mimochodom odzrkadľuje rozdelenie na kernel a užívateľské programy) neadekvátne. Existujú tri základné spôsoby ako to riešiť:

1. `setuid` programy môžu poskytnúť niektoré zo špeciálnych privilégií, ktoré sú výhradne pre `root` aj normálnym užívateľom (napr. `mount`, `passwd`),
2. systém schopností (capabilities) pridáva možnosť umožniť niektorým užívateľom vykonávať niektoré z neprístupných operácií,
3. prístup na úrovni užívateľov, ktorý sa spomínal na konci sekcie 1, kde služba beží pod `root`-om (napr. PolicyKit).

Bezpečnosť a vykonávanie programov

609

- bezpečnosť záleží na tom čo sa smie vykonávať
- spúšťanie ľubovoľného kódu patrí medzi najhoršie exploity
  - umožňuje neautorizované spúšťanie kódu
  - rovnaký efekt ako vydávanie sa za užívateľa
  - skoro také zlé ako ukradnuté prístupové údaje

Kontrola nad tým, ktorý kód sa môže spúšťať (a s akými privilégiami) je v ústredí všetkých obmedzení kontroly prístupu. Ak môže byť program oklamaný, aby spúšťal kód dodaný útočníkom, všetky privilégiá, ktoré program mal sú automaticky dostupné aj útočníkovi.

Nedôveryhodný vstup

610

- programy často spracúvajú dáta z pochybných zdrojov
  - vezmite si napr. programy na zobrazovanie obrázkov, audio & video prehrávače
  - rozbaľovanie archívov, vykresľovanie fontu, ...
- chyby v programoch môžu byť zneužité
  - program môže byť oklamaný za účelom vykonania kódu

Najčastejšou formou ako môžu byť programy unesené týmto spôsobom je cez nevhodné spracovanie nedôveryhodných vstupov, t.j. obsahu pochádzajúceho z nedôveryhodných zdrojov. Ak dokážu neočakávané vstupné dáta zmeniť priebeh vykonávania programu, otvára to dvere pre útočníka na prevzatie kontroly nad programom.
Samotný payload (kód, ktorý útočník chce spustiť) je obvykle dodaný ako súčasť vstupu, a teda sa k nemu typicky program chová ako ku dátam. Ak je však prítomná nejaká chyba, program môže byť oklamaný, aby tieto dáta spustil (alebo interpretoval) ako kód.

Proces ako subjekt

611

- niektoré privilégiá môžu byť späté s konkrétnym procesom
  - tieto sa uplatňujú iba počas života procesu
  - často ide skôr o obmedzenia než o privilégiá
  - týmto spôsobom sa robí tzv. privilege dropping – odoberanie privilégií
- obmedzenia sa dedia cez `fork()`

Programy (alebo časti programov bežiace v samostatnom procese) môžu požiadať operačný systém, aby im odobral niektoré z ich privilégií (napríklad prístup k súborovému systému, prístup k sieti, atď.). Je veľa spôsobov ako to urobiť, ale nie sú veľmi prenositeľné (t.j. závisia od funkcií konkrétneho operačného systému, ktoré nespadajú pod POSIX, napr. užívateľské menné priestory na Linuxe, seccomp, FreeBSD Capsicum, OpenBSD `pledge` a `unveil`, atď).
Jedným z mála prenositeľných prístupov je odoberanie privilégií, a ide v podstate o podmnožinu oddelenia prvilégií (privilege separation): je vytvorený špeciálny užívateľ pre konkrétny proces a tento proces, po tom, čo dokončí potrebné privilegované inicializačné operácie, použije `setuid` a prípadne `chroot` aby sa zamkol.

Pieskovisko – sandboxing

612

- snaží sa obmedziť škody spôsobené exploitmi, ktoré spúšťajú kód
- program odoberie všetky privilégiá, ktoré môže
  - toto urobí ešte predtým než siahne na akýkoľvek vstup
  - útočník sa musí vysporiadať s obmedzenými privilégiami
  - často to môže predísť úspešnému útoku

Pieskovisko je kolekcia techník (vrátane niektorých vyššie), ktorá sa snaží minimalizovať dopad úspešného exploit útoku na program. Pieskovisko môže byť dobrovoľné (program si založí vlastný sandbox) alebo nedobrovoľné (tiež pozri ďalší slide).

Nedôveryhodný kód

613

- tradične by ste spúšťali iba dôveryhodný kód
  - často na základe reputácie alebo iných externých faktorov
  - toto neškáluje na obrovské množstvo dodávateľov
- je bežné spúšťať nedôveryhodný, dokonca pochybný kód
  - môže to byť v poriadku s dostatočne silným sandboxom

Spúšťanie kódu z pochybných zdrojov je vždy riskantné, ale je v podstate garantované, že dôjde k prielomu ak nie sú uplatnené preventívne opatrenia. Keďže je však moderný web plný spustiteľného kódu, uchyľujeme sa k jeho čo najväčšej izolácii, a dúfame, že to dobre dopadne.

Kontrola prístupu na úrovni API

614

- systém schopností (capabilities) pre zdroje na užívateľskej úrovni
  - veci ako zoznamy kontaktov, kalendáre, záložky
  - objekty, ktoré nie sú poskytované priamo kernelom
- vynucovanie napr. cez virtuálny stroj
  - nevzťahuje sa na spúšťanie natívneho kódu
  - alternatíva: API založené na IPC

Selektívne udeľovanie práv programom cez systémy práv na užívateľskej úrovni je možné aj pre bežných užívateľov (nie root). Existujú dve bežne používané metódy:

1. virtuálny stroj (na úrovni programov), ako JVM alebo javascript virtuálne stroje, ktoré sú vstavané do webových prehliadačov, ktoré vynucujú, aby programy komunikovali so systémom iba cez obmedzené API,
2. prísny sandbox, ktorého jediný prístup k systému je sprostredkovávaný démonom, ktorý beží mimo sandbox (napr. snap a flatpak, do istej miery).

Oba prístupy môžu byť skombinované, kde bežná technika je zamknutie VM použitím sandboxu na úrovni OS, ako ochrana pred bezpečnostnými chybami v samotnom VM.

Android/iOS práva

615

- aplikácie v obchode (store) sú polo-dôveryhodné
- typicky ide o jedno-užívateľské počítače/zariadenia
- práva sú pripojené k aplikáciám namiesto užívateľov
- čiastočne virtuálni užívatelia, čiastočne na úrovni API

Napríklad na Androide dostane každá aplikácia svojho vlastného virtuálneho užívateľa s veľmi obmedzenými právami a interakcia so systémom prebieha takmer výhradne cez vysokoúrovňové API rozhrania. Tieto API potom vykonávajú kontrolu práv, a prípadne si vyžiadajú od užívateľa potvrdenie, ak je to potrebné.

Review Questions

616

37. What is a user?
38. What is the principle of least privilege?
39. What is an access control object?
40. What is a sandbox?

Časť 11: Virtualizácia & kontajnery

Táto prednáška sa sústredí na prevádzkovanie niekoľkých operačných systémov na rovnakom fyzickom počítači. Doteraz sme vždy predpokladali, že operačný systém (konkrétne kernel) má priamu kontrolu nad fyzickými zdrojmi. Tento týždeň si ukážeme, že to nemusí byž vždy tak (v skutočnosti je to u produkčných systémov čím ďalej vzácnejšie). Namiesto toho uvidíme, že niekoľko operačných systémov môže zdieľať jeden počítač spôsobom podobným tomu, ako niekoľko aplikácií (procesov) koexistuje v rámci operačného systému.
Tiež preskúmame prístup, ktorý je kompromisom, známy ako kontajnery, kde sú duplikované a navzájom izolované iba časti operačného systému, ktoré patria užívateľskému priestoru, zatiaľ čo kernel zostáva zdieľaný a zachováva si priamu kontrolu nad strojom.

Obsah prednášky

618

1. Hypervízory
2. Kontajnery
3. Správa

Prednáška je rozdelená na 3 časti: prvá časť predstaví plnú virtualizáciu a koncept hypervízora, zatiaľ čo druhá časť bude pojednávať o kontajneroch. Konečne, pozrieme sa na niekoľko tém, ktoré majú oba systémy spoločné, a v istom zmysle sú tiež relevantné pri správe sietí fyzických počítačov.

Časť 11.1: Hypervízory

V oblasti hardvérovo-akcelerovanej virtualizácie je hypervízor súčasťou virtualizačného softvéru, zhruba zodpovedajúcou kernelu operačného systému.

Čo je to hypervízor

620

- tiež známy ako monitor virtuálneho stroja (VMM)
- umožňuje spustenie viacerých operačných systémov
- niečo ako kernel, ktorý spúšťa kernely
- zlepšuje využitie hardvéru

Hoci sa hypervízor samotný správa trochu ako kernel, keďže leží medzi hardvérom a virtualizovaným operačným systémom, virtualizované operačné systémy, ktoré bežia nad ním sú, istým spôsobom, ako procesy (vrátane ich kernelov). Konkrétne, ich fyzické pamäte sú izolované (buď použitím normálneho MMU a trochu softvérovej mágie, alebo použitím MMU, ktorá je schopná dvojúrovňového prekladu) a využívajú časové zdieľanie dostupných procesorov.

Motivácia

621

- zdieľanie na úrovni OS je zložité
  - izolácia na úrovni užívateľov je často nedostačujúca
  - iba `root` môže inštalovať softvér
- rozhranie hypervízor/OS je jednoduché
  - v porovnaní s rozhraním medzi OS a aplikáciou

Virtualizované operačné systémy umožňujú úroveň autonómie, ktorá zvyčajne nie je možná keď viacero užívateľov zdieľa jeden operačný systém. Čiastočne je to spôsobené jednoduchosťou rozhrania medzi hypervízorom a operačným systémom: nie sú tam žiadne súborové systémy, vlastne žiadna komunikácia medzi operačnými systémami (okrem komunikácie cez štandardné sieťové rozhranie), žiadna správa užívateľov, a tak ďalej. Virtuálny stroj iba vyhradí nejaké zdroje a sprístupní ich operačnému systému.

Virtualizácia všeobecne

622

- veľa zdrojov je "virtualizovaných"
  - fyzická pamäť cez MMU
  - periférie operačným systémom
- zjednodušuje to správu zdrojov
- umožňuje to izoláciu komponent

Operačné systémy (alebo, ak preferujete, počítače) samozrejme nie sú jedinou vecou, ktorá môže byť (alebo je) virtualizovaná. Ak sa nad tým zamyslíte, veľká časť operačného systému samotného je postavená na nejakej forme virtualizácie: virtuálna pamäť, súborové systémy, sieťová vrstva, ovládače zariadení – tieto všetky v istom zmysle virtualizujú hardvérové zdroje. Toto následne umožňuje aby niekoľko programov, a niekoľko užívateľov, zdieľalo tieto zdroje bezpečne a spravodlivo.

Typy hypervízorov

623

- typ 1: „bare metal"
  - samo-stojaci, na spôsob mikrokernelu
- type 2: hosťovaný
  - beží nad normálnym OS
  - obvykle potrebuje podporu kernelu

Existujú dva základné typy hypervízorov, v závislosti od toho ako je systém ako celok vrstvený. U typu 1 je hypervízor na spodku zásobníka (tesne nad hardvérom), a je zodpovedný za správu základných zdrojov (niečo ako jednoduchý mikrokernel): procesor a RAM (plánovanie a správa pamäti, v tomto poradí).
Na druhú stranu, hypervízory typu 2 bežia nad operačným systémom a využívajú jeho plánovač a správu pamäti: virtuálne stroje sa objavujú ako skutočné procesy hosťujúceho systému.

Typ 1 (Bare Metal)

624

- IBM z/VM
- (Citrix) Xen
- Microsoft Hyper-V
- VMWare ESX

Typ 2 (Hosťované)

625

- VMWare (Workstation, Player)
- Oracle VirtualBox
- Linux KVM
- FreeBSD bhyve
- OpenBSD vmm

História

626

- začalo to s tzv. mainframe počítačmi
- IBM CP/CMS: 1968
- IBM VM/370: 1972
- IBM z/VM: 2000

Prvýkrát bola súbežná prevádzka niekoľkých operačných systémov na rovnakom hardvéri uvedená firmou IBM v neskorých 60. rokoch a krátko na to sa na veľkých mainframových počítačoch stala pomerne štandardnou funkciou.

Virtualizácia na PC

627

- x86 hardvéru chýba privilegovaný režim virtualizácie
- výhradne softvérové riešenia dostupné od 90. rokov
  - Bochs: 1994
  - VMWare Workstation: 1999
  - QEMU: 2003

Malé (osobné) počítače dlhú dobu neponúkali žiadne schopnosti virtualizácie. Výkon PC procesorov začal dostačovať na emuláciu PC na PC v polovici 90. rokov, ale pokles výkonu bol spočiatku obrovský a bol vhodný iba na spúšťanie starého softvéru (ktorý bol navrhnutý pre výrazne pomalší hardvér).

Paravirtualizácia

628

- predstavená ako VMI v roku 2005 firmou VMWare
- alternatívny prístup v projekte Xen v roku 2006
- spolieha sa na modifikáciu hosťovského (guest) OS
- takmer natívna rýchlosť bez podpory HW

O dekádu neskôr urobilo VMWare prelom v softvérovo-založenej virtualizačnej technológii, vynájdením paravirtualizácie: vyžadovalo to modifikácie hosťovského (guest) operačného systému, ale tou dobou sa začínali rozmáhať open-source operačné systémy – a portovať open-source systémy na paravirtualizujúci hypervízor nebolo tak ťažké.

Virtuálna revolúcia x86

629

- 2005: rozšírenia pre virtualizáciu na x86
- 2008: MMU virtualizácia
- nemodifikovaný host (guest) v takmer natívnej rýchlosti
- väčšina výhradne softvérových riešení sa stala zastaralou

Zhruba v rovnakom čase začali výrobcovia desktopových CPU zapracovávať virtualizačné rozšírenia, ktoré následne spôsobili, že už nebolo nutné modifikovať hosťovský (guest) operačný systém (aspoň teoreticky). Od roku 2008 už bežné desktopové procesory ponúkali MMU virtualizáciu, čím ďalej zjednodušili návrh x86 hypervízora (a tiež ho urobili efektívnejším).

Paravirtuálne zariadenia

630

- špeciálne ovládače pre virtualizované zariadenia
  - blokové úložisko, sieť, konzola
  - generátor náhodných čísel
- rýchlejšie a jednoduchšie než emulácia
  - nezávislé od CPU/MMU virtualizácie

Paravirtualizácia však mala rýchly a dramatický návrat: hoci bola virtualizácia CPU a pamäte, z väčšej časti, obsluhovaná najmä samotným hardvérom, hardvérovo-založený prístup je pre virtualizáciu periférií pomerne nehospodárny.
Naviac, paravirtualizované periférie nepotrebujú zmeny v hosťovskom (guest) operačnom systéme: všetko, čo potrebujú, je v zásade obyčajný ovládač zariadení, ktorý realizuje daný protokol. Virtuálne periférie poskytované hostiteľským (host) systémom sa potom vďaka vhodnému ovládaču javia hosťovskému (guest) systému ako bežné zariadenia.

Virtuálne počítače

631

- obvykle známe ako virtuálne stroje (virtual machines)
- všetko v počítači je virtuálne
  - buď prostredníctvom hardvéru (VT-x, EPT)
  - alebo softvéru (QEMU, `virtio`, ...)
- výrazne jednoduchšie na správu než skutočný hardvér

Celý systém bežiaci pod virtualizovaným operačným systémom je známy ako virtuálny stroj (virtual machine; alebo niekedy tiež ako virtuálny počítač), nezamieňajme ich ale s virtuálnymi strojmi (tiež VM) na úrovni programov (napr. Java Virtual Machine).

Základné zdroje

632

- CPU a RAM
- perzistentné (blokové) úložisko
- sieťové pripojenie
- konzola

Typický virtuálny stroj poskytuje aspoň procesor, pamäť, blokové úložisko (na ktoré operačný systém uloží súborový systém), sieťové pripojenie a konzolu pre správu. Hoci sú ďalšie periférie možné, nie sú veľmi bežné, aspoň nie u serverov.

Zdieľanie CPU

633

- rovnaký princíp ako u normálnych procesov
- v hypervízore je plánovač
  - jednoduchší, s inými kompromismi
- privilegované inštrukcie vyvolajú výnimku (trap)

Väčšina inštrukcií (obzvlášť tie dostupné užívateľským programom) sú jednoducho vykonané bez dodatočnej réžie pre hostiteľské (host) CPU, bez priamej účasti hypervízora. Hypervízor však spravuje virtualizovanú MMU. Čo je však rovnako dôležité, keď CPU narazí na niektoré typy privilegovaných inštrukcií, vyvolá hypervízor, aby vykonal (emuloval) požadované akcie softvérovo.

Zdieľanie RAM

634

- veľmi podobné štandardnému stránkovaniu
- softvérové (tieňové stránkovanie – shadow paging)
- alebo hardvérové (dvojúrovňový preklad)
- fixné množstvo RAM pre každú VM

Podobne ako virtualizácia CPU, zdieľanie pamäte je postavené na rovnakých základných princípoch, ktoré štandardné operačné systémy používajú pre vzájomnú izoláciu procesov. Pamäť je rozdelená na stránky a väčšinu práce (preklad adries) realizuje MMU.

Tieňové tabuľky stránok

635

- hosťovský (guest) systém nemôže pristupovať k MMU
- nastaví sa tieňová tabuľka, neviditeľná pre hosťa (guest)
- tabuľky stránok hosťa synchronizuje VMM do sPT
- gPT je označené výhradne na čítanie → zápis = výnimka

Výnimka (trap) potom môže synchronizovať tabuľku stránok hosťa s tieňovými tabuľkami stránok, ktoré sú jej preloženou verziou. „Fyzické“ adresy uložené v hosťovej tabuľke stránok (gPT) sú virtuálne adresy hypervízora. Tieňová tabuľka stránok (sPT) ukladá reálne fyzické adresy, keďže je používaná reálnou MMU.

Dvojúrovňový preklad adries

636

- hardvérovo-podporovaná virtualizácia MMU
- pridáva preklad hosťovských (guest) na fyzické adresy
- výrazne zjednodušuje VMM
- tiež je výrazne rýchlejšia než tieňové tabuľky stránok

Tieňové tabuľky stránok spôsobujú veľa réžie, spôsobia totiž prepnutie do hypervízora pri každej zmene tabuľky stránok hosťa (guest). Nane-

šťastie, tabuľky stránok sú hosťovským (guest) operačným systémom menené pomerne často (na skutočnom hardvéri je toto pomerne lacné). Moderné procesory však ponúkajú ďalšiu úroveň prekladu, ktorá nie je dostupná hosťovskému (guest) operačnému systému. Keďže je si MMU vedomá virtualizácie, hosť (guest) môže priamo modifikovať svoje tabuľky stránok, bez narušenia vzájomnej izolácie virtuálnych strojov (a izolácie VM od hypervízora).

Zdieľanie siete

637

- obvykle paravirtualizovaná NIC
  - prenáša rámce medzi hosťom (guest) a hostiteľom (host)
  - obvykle pripojené ku SW mostu hostiteľa
  - alternatívy: smerovanie, NAT
- všetci používajú jednu fyzickú sieťovú kartu (NIC)

V súčasných riešeniach virtualizácie sa pre sieťovú vrstvu používa paravirtuálna sieťová karta (NIC – network interface card), ktorá je pripojená k pseudo-zariadeniu, ktoré predstavuje Ethernetový tunel na hostiteľskom (host) systéme (v podstate ide o virtuálnu sieťovú kartu, ktorá manipuluje s Ethernetovými rámcami). Rámce odoslané na paravirtuálne zariadenie sa objavia na virtuálnej NIC hostiteľa (host) a naopak. Pseudo-zariadenie je potom buď softvérovo premostené (bridged) na hardvérovú sieťovú kartu (a teda na vonkajší ethernet), alebo alternatívne je nastavené smerovanie (vrstva 3) medzi pseudo-zariadením a hardvérovou sieťovou kartou.

Virtuálne blokové zariadenia

638

- obvykle tiež paravirtualizované
- často podložené normálnymi súbormi
  - môžu byť v špeciálnom formáte
  - napr. založené na copy-on-write
- ale môže to byť aj skutočné blokové zariadenie

Podobne ako sieťová vrstva, blokové úložisko je typicky založené na paravirtualizácii. V tomto prípade je hostiteľská (host) strana zariadenia buď zaistená bežným súborom v súborovom systéme hostiteľa, alebo niekedy aj skutočným blokovým zariadením tamtiež (často virtualizovaným, napr. cez LVM/device mapper alebo podobnú technológiu, ale niekedy je tiež priamo zaistená hardvérovým blokovým zariadením).

Špeciálne zdroje

639

- užitočné najmä na desktopových systémoch
- GPU / grafický hardvér
- audio zariadenia
- tlačiarne, skenery, ...

Keď už sme pokryli nevyhnutné časti, poďme sa letmo pozrieť na ďalšie triedy hardvéru. Avšak, s prípadnou výnimkou výpočtových GPU sú periférie užitočné iba na desktopových systémoch, ktoré tvoria veľmi malý trh v porovnaní so serverovou virtualizáciou.

PCI Passthrough

640

- „anti-virtualizačná“ technológia
- založené na IO-MMU (VT-d, AMD-Vi)
- virtuálny OS môže pristupovať k reálnemu hardvéru
  - samozrejme iba jeden OS naraz

Najprv spomeňme veľmi generickú „proti-virtualizujúcu“ metódu, ktorou môže virtuálny stroj dostať prístup hardvéru: PCI zariadenie je sprístupnené hosťovskému (guest) operačnému systému priamo, cez IO-MMU-mapovanú pamäť. Použitie IO-MMU je nevyhnutné, pretože inak by hosťovský (guest) OS mohol inštruovať hardvér, aby prepísal fyzickú pamäť, ktorá patrí hostiteľovi (host), alebo inej VM bežiacej na rovnakom systéme. Keď je však tomuto predídené, nič nebráni hostiteľskému (host) systému, aby dal kontrolu nad špecifickými PCI koncovými bodmi hosťovi - guest (samozrejme, hostiteľský (host) systém sa nesmie pokúsiť komunikovať s týmito zariadeniami cez svoje vlastné ovládače, inak by nastal chaos).

GPU a virtualizácia

641

- môžu byť priradené (cez VT-d) jednému OS
- alebo časovo zdieľané použitím natívnych ovládačov (GVT-g)
- paravirtualizované
- zdieľané inými spôsobmi (X11, SPICE, RDP)

Samozrejme, keďže je GPU pripojené cez PCI, môže byť zdieľané pomocou IO-MMU (VT-d) prístupu, ktorý je popísaný vyššie. Moderné GPU však všetky podporujú časové zdieľanie (time-sharing; t.j. umožňujú, aby kontexty mohli byť pozastavené a obnovené, rovnako ako vlákna a procesy na CPU). Aby to mohlo fungovať, hypervízor (alebo hostiteľský OS - host) musí mať k dispozícii pre dané GPU ovládače, ktoré mu umožnia sprostredkovávať prístup jednotlivým virtuálnym strojom (VM).
Ďalším riešením je paravirtualizácia: hosť (guest) používa protokol nezávislý na výrobcovi na posielanie prúdu príkazov ovládaču bežiacemu v hypervízore, ktorý následne vykonáva multiplexovanie. Hosťovský (guest) systém ešte stále potrebuje časť GPU ovládača, ktorá patrí do užívateľského priestoru, aby vygeneroval prúd príkazov a preložil shadery.
Samozrejme je možné medzi hosťom (guest) a hostiteľom (host) použiť existujúce sieťové protokoly pre grafiku, hoci nikdy nie sú tak efektívne ako jedna z vyššie uvedených špecializovaných možností.

Periférie

642

- užitočné buď cez passthrough
  - audio, webkamery, ...
- alebo štandardné technológie zdieľania
  - sieťové tlačiarne & skenery
  - sieťové audio servery

Existuje veľké množstvo periférií, ktoré môžu byť pripojené k PC. Niektoré z nich, ako tlačiarne a skenery, a v niektorých prípadoch (alebo skôr v niektorých operačných systémoch) audio hardvér, môžu byť zdieľané cez štandardné siete, a teda tiež medzi hosťami (guests) a hostiteľom (host) cez virtuálnu sieť. Pri tomto type periférií buď nedochádza ku strate výkonu (tlačiarne, skenery), prípadne dochádza k malému nárastu latencie (toto ovplyvňuje najmä audio zariadenia).

Passthrough periférií

643

- virtuálne PCI, USB alebo SATA zbernica
- preposielanie (forwarding) na skutočné zariadenie
  - napr. jeden USB kľúč
  - alebo jeden SATA disk

Samozrejme, zdieľanie po sieti nie je vždy praktické. Našťastie sa väčšina periférií pripája k hostiteľskému (host) systému cez niekoľko štandardných zberníc, ktoré nie je zložité využiť ani na techniku passth-

rough, ani na paravirtualizáciu. Zariadenia sa potom zobrazujú ako koncové body na virtuálnej zbernici predpísaného typu, ktorý je sprístupnený hosťovskému (guest) operačnému systému.

Pozastavenie & obnovenie

644

- VM môže byť pomerne jednoducho zastavený
- RAM zastaveného VM môže byť skopírovaná
  - napr. do súboru v hostiteľskom (host) súborovom systéme
  - spolu s registrami a ďalším stavom
- a tiež neskôr načítaný a obnovený (môže pokračovať)

Dôležitou funkciou, ktorá je dostupná vo väčšine virtualizačných riešení je schopnosť pozastaviť (suspend) beh VM a uložiť jej stav do súboru (t.j. vytvoriť obraz bežiaceho virtualizovaného OS). Toto je samozrejme užitočné iba ak sa tento obraz dá neskôr načítať a môže byť obnovený „ako keby sa nič nestalo".
Navonok to vyzerá ako to, čo sa deje keď sa zavrie notebook: počítač sa zastaví (v tomto prípade aby šetril energiou) a keď je znovu otvorený, pokračuje tam, kde prestal. Dôležitý rozdiel je, že u VM nemusí hosťovský (guest) operačný systém spolupracovať, alebo dokonca ani si uvedomovať, že prebieha operácia pozastavenia/obnovenia (suspend/resume).

Základy migrácie

645

- uložený stav môže byť poslaný cez sieť
- a obnovený na inom hostiteľovi (host)
- za predpokladu, že virtuálne prostredie je rovnaké
- toto je známe ako paused migration – pozastavená migrácia

Samozrejme, ak obraz môže byť uložený v súbore, môže rovnako byť poslaný cez sieť. Obnovenie obrazu na inom hostiteľovi (host) sa nazýva „pozastavená" (paused) migrácia, keďže VM je pozastavená po dobu sieťového prenosu: v závislosti od veľkosti obrazu, toto môže trvať dosť dlho na to, aby došlo k uplynutiu časového limitu (time out) TCP spojení alebo protokolov na aplikačnej úrovni. Samozrejme, aj keď toto nenastane, systém bude mať pri interaktívnom použití zreteľné oneskorenie (lag).
Zjavným predpokladom tejto operácie je požiadavka, že podporné prostredie mimo VM je dostatočne kompatibilné medzi hostiteľmi: konkrétne, podkladové úložisko pre virtualizované blokové zariadenia, a virtuálna sieťová infraštruktúra sa musia zhodovať.

Živá migrácia

646

- používa asynchrónne pamäťové snímky
- hostiteľ (host) kopíruje stránky a označuje ich ako výhradne na čítanie
- snímok je odosielaný počas toho ako je vytváraný
- zmenené stránky sú poslané nakoniec

Živá migrácia je zlepšením oproti pozastavenej migrácii (popísanej vyššie) v tom, že nespôsobuje zreteľné oneskorenie (lag) a neohrozuje TCP alebo iné stavové protokoly, ktoré používajú časové limity na detekciu rozbitých spojení.
Hlavnou myšlienkou, ktorá umožňuje živú migráciu je, že VM môže pokračovať v behu ako normálne, zatiaľ čo jeho pamäť je kopírovaná, s podmienkou, že všetky ďalšie zápisy musia byť zaznamenávané hypervízorom: to sa dosahuje cez štandardný trik „copy-on-write", kde sú stránky označované ako výhradne na čítanie (read-only) tesne predtým než sú skopírované, a hypervízor chytá výnimky. Umožňuje, aby zápisy naďalej prebiehali, ale tiež označuje stránky ako „špinavé" (dirty). Keď je prvý priechod dokončený, urobí sa ďalší priechod, ale tentokrát iba cez špinavé stránky, ktoré sú zároveň označované za čisté.

Odovzdávanie pri živej migrácii

647

- VM je následne pozastavená
- sú poslané registre a niekoľko posledných stránok
- VM je obnovená na vzdialenom konci
- obvykle do niekoľkých milisekúnd

Keď je počet špinavých stránok na konci iterácie dostatočne malý, VM je pozastavená, zvyšné špinavé stránky a CPU kontext sú prekopírované a VM je okamžite obnovená. Keďže má posledný prenos len niekoľko sto kilobajtov, latencia prepnutia je takmer zanedbateľná.

Memory Ballooning

648

- ako dealokovať "fyzickú" pamäť?
  - t.j. vrátiť ju hypervízoru
- toto je často pri virtualizácii žiaduce
- vyžaduje špeciálne rozhranie hostiteľ/hosť (host/guest)

Nakoniec uvážme, že hypervízor alokuje pamäť hosťovským (guest) VM na požiadanie, ale operačné systémy typicky nemajú koncept 'dealokácie' fyzickej pamäte, ktorú aktívne nepoužívajú. Za týchto okolností, ak dôjde na VM k špičke vo využití pamäte, táto pamäť bude natrvalo držaná touto VM, aj keď pre ňu nemá žiadne využitie.
Často používaným riešením je takzvaný „memory ballooning" ovládač, ktorý beží na strane hosťa (guest) a vracia nenamapovanú „fyzickú" (z pohľadu hosťa) pamäť hostiteľskému (host) operačnému systému. Zruší sa mapovanie pamäte na strane hostiteľa (t.j. obsah pamäte je pre hosťa stratený) a neskôr opäť namapovaný ak vznikne dopyt.

Časť 11.2: Kontajnery

Hoci je hardvérovo-akcelerovaná virtualizácia pomerne efektívna čo sa týka CPU réžie, sú s ňou spojené ďalšie náklady. Niektoré sa dajú redukovať rôznymi trikmi (ako memory ballooning, TRIM, copy-on-write obrazy disku, atď.) ale ďalšie môže byť problém eliminovať. Keď je požiadavkou maximálne využitie zdrojov, kontajnery môžu často prekonať plnú virtualizáciu, bez výrazného zhoršenia v iných oblastiach, ako udržiavateľnosť, izolácia, alebo bezpečnosť.

Čo sú kontajnery?

650

- virtualizácia na úrovni OS
  - napr. virtualizovaná sieťová vrstva
  - alebo obmedzený prístup k súborovému systému
- nie celý virtuálny počítač
- implementované ako rozšírenie procesov

Kontajnery používajú virtualizáciu (v širokom zmysle slova), ktorá je už vstavaná v operačnom systéme, založená najmä na procesoch. To je rozšírené o dodatočnú separáciu, kde skupiny procesov môžu zdieľať, napríklad, sieťovú vrstvu (sieťový zásobník), ktorá je oddelená od sieťovej vrstvy, ktorá je dostupná inej množine procesov. Hoci obe sieťové vrstvy používajú rovnaký hardvér, majú oddelené IP adresy, oddelené routovacie tabuľky, a tak ďalej. Podobne, každý kontajner má obvykle prístup iba k časti súborového systému (môže byť oddelený napr. pomocou `chroot`), mapovanie užívateľov je oddelené, a rovnako sú oddelené aj tabuľky procesov.

Prečo kontajnery

651

- u virtuálnych strojov chvíľu trvá než nabootujú
- každá VM potrebuje svoj vlastný kernel
  - réžia sa nasčíta, keď potrebujete veľa VM
- jednoduchšie efektívne zdieľať pamäť
- jednoduchšie zmenšiť obraz OS

Kontajnery majú dve základné črty, ktoré ich robia atraktívnymi:

1. rýchlosť vytvorenia (ustanovenia, provisioning) – čas, ktorý trvá dostať sa od 'chcem nový systém' do jeho nabootovania,
2. efektívnejšie využitie zdrojov.

Oba sú z veľkej časti umožnené zdieľaním kernelu medzi kontajnermi: v prvom prípade nie je nutné inicializovať (bootovať) nový kernel, čo šetrí nezanedbateľné množstvo času. U druhého bodu je toto ešte dôležitejšie: pri jednom kerneli môžu kontajnery zdieľať súbory (napr. cez spoločné prípojné body – mounts) a procesy z rôznych kontajnerov môžu zdieľať pamäť – najmä spustiteľné obrazy a zdieľané knižnice, ktoré sú zaistené spoločnými súbormi. Dosiahnutie rovnakého efektu s virtuálnymi strojmi je celkom nemožné.

Zdieľanie kernelu

652

- niekoľko kontajnerov zdieľa jeden kernel
- ale nie tabuľky užívateľov, tabuľky procesov, ...
- kernel to musí explicitne podporovať
- ďalšia úroveň izolácie (proces, užívateľ, kontajner)

Samozrejme, keďže jeden kernel obsluhuje viacero kontajnerov, daný kernel musí podporovať dodatočnú úroveň izolácie (nad rámec procesov a užívateľov), kde oddelené kontajnery majú samostatné tabuľky procesov a tak ďalej.

Čas potrebný na bootovanie

653

- odľahčenému virtuálnemu stroju to trvá sekundu alebo dve
- kontajner to môže stihnúť pod 50ms
- ale VM môžu byť pozastavené a obnovené
- ale uspané VM zaberajú výrazne viac miesta

Aj keď nebudeme brať do úvahy veci ako prípravu obrazov disku, už na samotnom čase bootovania môže byť kontajner 20 krát rýchlejší než tradičný virtuálny stroj (ak nepočítame exokernely a podobné malé operačné systémy).

`chroot`

654

- zárodok kontajnerových systémov
- nie je veľmi sofistikovaný ani bezpečný
- ale umožňuje niekoľko obrazov OS pod 1 kernelom
- všetko ostatné je zdieľané

Systémové volanie `chroot` môže byť použité na beh niekoľkých obrazov OS (presnejšie ich častí, ktoré tvoria užívateľský priestor) pod jedným kernelom. Keďže je však všetko okrem súborového systému plne zdieľané, nemôžeme sa zatiaľ úplne baviť o kontajneroch.

'Kontajnery' založené na `chroot`

655

- tabuľky procesov, sieť, atď. sú zdieľané
- superužívateľ musí byť tiež zdieľaný
- kontajnery majú vlastný pohľad na súborový systém
  - vrátane systémových knižníc a pomocných programov

Keďže sú tabuľky procesov, sieťová vrstva a ďalšie dôležité služby zdieľané medzi obrazmi, spôsobuje to veľa interferencie. Napríklad, nie je možné spustiť dva nezávislé webové servery z dvoch rôznych `chroot` pseudo-kontajnerov, pretože iba jeden sa dokáže namapovať na (zdieľaný) port 80 (alebo 443 ak sa cítite moderne).
Ďalším dôsledkom je, že rola superužívateľa v kontajneri nie je obmedzená: `root` vnútri sa jednoducho môže stať `root`-om vonku.

BSD väzenia - jails

656

- rozšírenie kontajnerov založených na `chroot`
- pridáva oddelenie užívateľa a tabuľky procesov
- a virtualizovanú sieťovú vrstvu
  - každé väzenie môže dostať svoju vlastnú IP adresu
- `root` má vo väzení obmedzené právomoci

Mechanizmus väzení vo FreeBSD je vývojom `chroot`, ktorý pridáva to, čo chýba: oddelenie užívateľov, tabuliek procesov a sieťových vrstiev. Väzenie tiež limituje čo môže robiť 'vnútorný' `root` (a bráni tomu, aby získali privilégiá mimo väzenia). Je jedným z najstarších open-source riešení kontajnerizácie.

Linux VServer

657

- ako BSD väzenia ale na Linuxe
  - FreeBSD jail 2000, VServer 2001
- nie je súčasťou hlavnej vývojovej vetvy kernelu
- `root` užívateľ je vo väzení čiastočne izolovaný

Podobná vec bola urobená na Linux-ovom kerneli o rok neskôr, ale nebola prijatá do oficiálnej verzie kernelu a dlho bola distribuovaná ako sada externých záplat (patchov).

Menné priestory - namespaces

658

- oddiely určujúce viditeľnosť v Linux kerneli
- virtualizujú bežné OS zdroje
  - hierarchia súborového systému (vrátane prípojných bodov - mounts)
  - tabuľky procesov
  - siete (IP adresa)

Riešenie, ktoré bolo nakoniec pridané do oficiálnych Linuxových kernelov je založené na menných priestoroch, ktoré spravujú každý aspekt kontajnerizácie osobitne: keď je nový proces vytvorený (cez systémové volanie podobné `fork`-u, ktoré sa volá `clone`), rodič môže špecifikovať, ktoré aspekty majú byť zdieľané s rodičom a ktoré majú byť oddelené.

cgroups

659

- spravuje alokáciu HW zdrojov v Linuxe
- CPU skupina je spravodlivá jednotka plánovania
- pamäťová skupina nastavuje limity na využitie pamäte
- prakticky nezávislé na menných priestoroch

Ďalším dôležitým komponentom u Linuxových kontajnerov sú 'kontrolné skupiny', ktoré limitujú využitie zdrojov celého podstromu procesov (ktorý sa môže zhodovať s podstromom procesov, ktoré patria jednému kontajneru).To umožňuje kontajnerom, aby boli izolované nielen vzhľadom na svoj prístup k objektom na úrovni OS, ale aj čo sa týka spotreby zdrojov.

LXC

660

- kanonický Linuxový spôsob vytvárania kontajnerov
- založený na menných priestoroch a cgroups
- relatívne nový (2008, 7 rokov po vserver)
- funkcionalita podobná ako u VServer, OpenVZ ap.

LXC je sada užívateľských nástrojov na správu kontajnerov založená na Linuxových menných priestoroch a kontrolných skupinách. Od verzie 1.0 (circa 2014) LXC tiež umožňuje oddelenie vnútro-kontajnerového superužívateľa, a tiež existujú neprivilegované kontajnery, ktoré môžu vytvárať a spravovať bežní užívatelia (s určitými obmedzeniami).

Linux v užívateľskom režime (User-mode Linux)

661

- na polceste medzi kontajnerom a virtuálnym strojom
- starší, plne paravirtualizovaný systém
- Linux-ový kernel beží ako proces na inom Linuxe
- integrovaný do Linuxu 2.6 v roku 2003

Porty kernelov 'na seba samých': režim, kde kernel beží ako bežný užívateľský proces nad inou konfiguráciou rovnakého kernelu, sú niekde medzi kontajnermi a plnými virtuálnymi strojmi. V dosť veľkej miere sa spoliehajú na techniky paravirtualizácie, hoci pomerne netypickým spôsobom: keďže je kernel štandardný proces, môže priamo pristupovať k POSIX API hostiteľského (host) operačného systému, a napríklad priamo zdieľať hostiteľský súborový systém.

DragonFlyBSD virtuálne kernely

662

- veľmi podobné Linuxu v užívateľskom režime
- súčasť DFlyBSD od 2007
- používa štandardnú libc, na rozdiel od UML
- paravirtuálny ethernet, úložisko a konzola

Ďalší príklad rovnakého prístupu je známy ako 'virtuálne kernely' v DragonFlyBSD. V tomto prípade užívateľský port kernelu dokonca používa štandardnú libc, ako akýkoľvek iný program. Nanešťastie nie je možný žiaden priamy prístup k hostiteľskému (host) súborovému systému, vďaka čomu má tento prístup bližšie k štandardným VM.

Kernely v užívateľskom režime (User Mode Kernels)

663

- je jednoduchšie ich bezpečne dodatočne pridať
  - používajú existujúce bezpečnostné mechanizmy
  - pre hostiteľa (host) ide obvykle o štandardný proces
- kernel však potrebuje byť portovaný
  - analogické k novej hardvérovej platforme

Pokiaľ ide o zložitosť implementácie, kernely v užívateľskom režime sú jednoduchšie než kontajnery a ponúkajú lepšiu bezpečnosť na strane hostiteľa (host), keďže vystupujú ako bežné procesy, bez špeciálneho postavenia.

Migrácia

664

- nie je bežne podporovaná, na rozdiel od hypervízorov
- stav procesu sa výrazne ťažšie serializuje
  - popisovače súborov, sieťové pripojenia, ap.
- je to trochu zmiernené rýchlym vypnutím/bootovaním

Jednou veľkou nevýhodou kontajnerov aj kernelov v užívateľskom režime je nedostatok podpory pre pozastavenie a obnovenie, a teda pre migráciu. V oboch prípadoch je to spôsobené výrazne zložitejším stavom procesu v porovnaní s virtuálnym strojom, hoci tento problém je výrazne závažnejší u kontajnerov (kernel v užívateľskom režime je často iba jeden proces na hostiteľovi (host), zatiaľ čo procesy v kontajneroch sú vlastne reálne procesy na strane hostiteľa).

Časť 11.3: Management

Disk Images

666

- disk image is the embodiment of the VM
- the virtual OS needs to be installed
- the image can be a simple file
- or a dedicated block device on the host

Snapshots

667

- making a copy of the image = snapshot
- can be done more efficiently: copy on write
- alternative to OS installation
  - make copies of the freshly installed image
  - and run updates after cloning the image

Duplication

668

- each image will have a copy of the system
- copy-on-write snapshots can help
  - most of the base system will not change
  - regression as images are updated separately
- block-level de-duplication is expensive

File Systems

669

- disk images contain entire file systems
- the virtual disk is of (apparently) fixed size
- sparse images: unwritten area is not stored
- initially only filesystem metadata is allocated

Overcommit

670

- the host can allocate more resources than it has
- this works as long as not many VMs reach limits
- enabled by sparse images and CoW snapshots
- also applies to available RAM

Thin Provisioning

671

- the act of obtaining resources on demand
- the host system can be extended as needed
  - to keep pace with growing guest demands
- alternatively, VMs can be migrated out
- improves resource utilisation

Configuration

672

- each OS has its own configuration files
- same methods apply as for physical networks
  - software configuration management
- bundled services are deployed to VMs

Bundling vs Sharing

673

- bundling makes deployment easier
- the bundled components have known behaviour
- but updates are much trickier
- this also prevents resource sharing

Security

674

- hypervisors have a decent track record
  - security here means protection of host from guest
  - breaking out is still possible sometimes
- containers are more of a mixed bag
  - many hooks are needed into the kernel

Updates

675

- each system needs to be updated separately
  - this also applies to containers
- blocks coming from a common ancestor are shared
  - but updating images means loss of sharing

Container vs VM Updates

676

- de-duplication may be easier in containers
  - shared file system – e.g. link farming
- kernel updates: containers and type 2 hypervisors
  - can be mitigated by live migration
- type 1 hypervisors need less downtime

Docker

677

- automated container image management
- mainly a service deployment tool
- containers share a single Linux kernel
  - the kernel itself can run in a VM
- rides on a wave of bundling resurgence

The Cloud

678

- public virtualisation infrastructure
- “someone else’s computer”
- the guests are not secure against the host
  - entire memory is exposed, including secret keys
  - host compromise is fatal
- the host is mostly secure from the guests

Review Questions

679

41What is a hypervisor?
42What is paravirtualisation?
43How are VMs suspended and migrated?
44What is a container?