KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

OPERAČNÍ SYSTÉMY

ALEŠ KEPRT

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc, 22.12.2007

Abstrakt

Tento text má za cíl posloužit jako studijní text k předmětům o operačních systémech, ve všech jejich variantách. Text je postaven na úrovni teoretické, přičemž praktické doplnění poskytují další samostatné učební texty. Rozsah zde probírané látky odpovídá především potřebám samostudia studentů v kombinované formě, kteří mají operační systémy jako jednosemestrální kurz. Pro studium v prezenční formě, kde operační systémy mají dvojsemestrální podobu, může být především dobrým základem studia, nikoliv však jediným studijním zdrojem.

Cílová skupina

Text je primárně určen pro studenty oboru Aplikovaná informatika uskutečňovaného v kombinované formě na Přírodovědecké fakultě Univerzity Palackého v Olomouci a dále pro studenty oborů Informatika, Aplikovaná informatika a Učitelství výpočetní techniky uskutečňovaných v prezenční formě tamtéž. Pokrývá látku předmětů Operační systémy a Programové vybavení počítačů v jejich různých variantách (kódy XOSY, XOS, XSYS, OS1AI, OS1AA, OS2AI, VP1AW, VP2AW).
K praktickým cvičením zde probírané látky slouží také další dva doplňkové studijní texty, označené v referencích jako [Kep08a] a [Kep08b].

Obsah

1 Úvod

Studijní cíle: Vítejte v úvodní kapitole tohoto učebního textu. Tématem této kapitoly je samotný „počítač“, tedy pojem všeobecně známý. Podíváme se tedy na to, jak tento pojem můžeme definovat, čímž zároveň vysvětlíme, co bude tématem našeho dalšího studia. Uděláme si také krátké okénko do historie počítačů a připomeneme si známý von Neumannův model popisující hardwarovou strukturu počítače. Naučíme se také vidět počítač na různých úrovních abstrakce.

Klíčová slova: počítač, von Neumannův model, abstrakce hardwaru

Potřebný čas: 25 minut.

1.1 Co je to počítač

> **Průvodce studiem**
>
> Pojem „počítač“ je naštěstí všeobecně známý (každý z čtenářů této publikace jistě již počítač viděl), navíc na otázku „Co je to počítač?“ celkem jistě studenti na zkoušce z operačních systémů nenarazí, takže odpověď na ni vlastně ani nechtějí studovat.

Počítač je stroj pro zpracování dat dle instrukcí.

Počítač je obvykle definován jako stroj pro zpracování dat dle daných instrukcí (programu). (V literatuře přitom zřejmě najdete mnoho více či méně podobných definic.) V dnešní době počítače v různých podobách potkáváme doslova na každém kroku, nejobvyklejším typem počítačů jsou přitom jednoznačně počítače zabudované (embedded) – tyto malé počítače nám například slouží v každodenním životě jako ovládací jednotky různých domácích spotřebičů, mobilních telefonů, dopravních prostředků či hraček.

Jak známo, počítače nebyly vždycky tak malé jako dnes. Naopak, první počítače byly mnohem větší, v češtině se pro tyto historické počítače obvykle užívá trefný termín „sálové počítače“ (v angličtině „mainframe“ – čili nejde o doslovný překlad původního termínu). Počítače prvních generací byly nejen daleko větší než ty dnešní, ale také mnohem jednodušší a pomalejší. Jedním z mezníků ve vývoji počítačů bylo použití tranzistoru, který nahradil dříve používanou elektronku a zahájil tím tzv. druhou generaci počítačů na konci 50.let 20.století. Tranzistor umožnil výrobu počítačů levnějších, spolehlivějších, menších i rychlejších zároveň.

> **Průvodce studiem**
>
> Máme-li být zcela korektní, je třeba dodat, že úplně první počítače nebyly ani „sálové“ a dokonce ani binární (tj. nepracovaly na bázi jedniček a nul). Nicméně první reálně používané počítače, které všechny vznikly v souvislosti s druhou světovou válkou a vojenskými účely, skutečně byly velké a „sálové“.

Integrací většího množství maličkých tranzistorů do jediné strojově vyráběné součástky zvané *integrovaný obvod* bylo později umožněno vytvořit počítače ještě menší a opět především podstatně levnější než při manuálním skládání jednotlivých tranzistorů. Tím vznikla třetí generace počítačů.

Tím, jak se počítače zmenšovaly a zlevňovaly, byl samozřejmě také otevřen prostor pro zvětšování jejich výkonu či paměti (a to samozřejmě jejich opětovným zvětšováním). Během tohoto postupného vývoje se začalo ukazovat, že počítače svou složitostí již jistým způsobem přerůstají lidem přes hlavu a zatímco jejich výroba byla stále jednodušší, jejich programování bylo stále složitější. A někde v této době se začaly objevovat první návrhy operačních systémů.

Dnes s odstupem několika desetiletí můžeme již operační systém a jeho smysl velmi přesně definovat: Je to základní program počítače, který odstiňuje ostatní programy od přímé komunikace či spolupráce s hardwarem. Tento základní program umožňuje ostatním programům vyšší úroveň abstrakce, jsou v něm totiž řešeny nejsložitější a často se opakující operace s hardwarem. Ostatním programům operační systém nabízí své funkce v podobě jednoduššího rozhraní.

Průvodce studiem

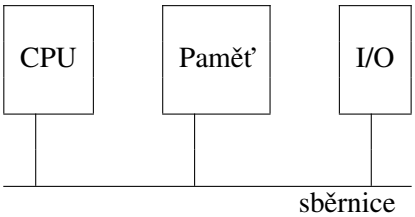
Pro příklad smyslu operačního systému nemusíme chodit daleko. Klasickou funkcí operačního systému je například načtení dat ze souboru. Zatímco z hlediska běžného programu je to otázka zavolání několika málo funkcí (otevři–načti–zavři), operační systém s tím má poměrně hodně práce. Podíváme-li se na věc v kontextu doby, čili s ohledem na to, že v minulosti neexistovaly sofistikované hard disky či CD mechaniky (které dnes mají samy zabudován řídicí počítač), ale jen „hloupá“ pásková zařízení bez jakékoliv inteligence, je vidět, že množství práce, které musel operační systém provádět, aby převedl elektrické impulzy z páskové jednotky na smysluplná data, bylo opravdu enormní.

Význam operačního systému jakožto základní softwarové výbavy počítače samozřejmě ještě dále rostl s tím, jak se dále zvětšovala složitost počítačů. Dalším hlavním mezníkem byl (samozřejmě) přechod na 4.generaci počítačů využívajících mikroprocesory. Složitost počítačů i dnes stále roste a to jak z důvodu jejich zmenšování (menší součástky umožňují rychlejší běh), tak z důvodu paralelizace – zatímco ještě v 80.letech 20.století se běžně používaly počítače s jedním mikroprocesorem, dnešní běžné počítače mají mikroprocesorů tolik, že je těžko spočítáme. Přímá práce s tak složitým systémem inteligentních strojů by dnes už ani nebyla možná, proto se bez operačního systému neobejdeme.

Jakkoliv je dnes v běžném životě možno potkat nepřeberné množství počítačů, u většiny z nich můžeme vidět jisté stejné rysy, jak co do jejich hardwarového návrhu, tak co do architektury jejich operačního systému. A právě tato dvě témata jsou náplní studia Operačních systémů, které pokrývá tento učební text. Naše pozornost bude samozřejmě zaměřena především na tzv. běžné domácí počítače „PC“, s jejichž provozem a programováním se informatici obvykle setkávají v praxi. Méně pak nás budou zajímat počítače zabudované, protože tyto jsou z principu velmi jednoduché (už proto, že musejí být také velmi levné, aby jejich používání mělo smysl), každý z nich má úzkou oblast využití a jejich operační systémy jsou stejně tak velmi jednoduché.

1.2 Architektura počítače, von Neumannův model

Připomeňme si krátce termín „von Neumannova architektura“ (viz obrázek 1). Jde o nejznámější počítačovou architekturu, která je dodnes nejčastěji zmiňována jako jakýsi de facto „základní model počítače“.



Obrázek 1: Schéma počítače – von Neumannův model

Von Neumannova architektura popisuje počítač jako sekvenční stroj, který používá jedinou paměť společnou pro kód i data. Podstatným rysem této architektury tedy je, že kód programu i jeho data jsou uloženy ve společné paměti. Výpočetní model je navíc sekvenční, čili příkazy

v programu (instrukce) jsou zpracovávány v řadě za sebou, pokud není (některou instrukcí) stanoveno jinak.

Průvodce studiem

Tomuto výpočetnímu modelu v podstatě odpovídá i řada známých programovacích jazyků (jako Pascal či C). Máme-li však být přesní, pravá von Neumannova architektura se dnes již téměř nepoužívá. Pro malé zabudované počítače, kterých je nejvíc, je totiž vhodnější harvardský model a „velké“ klasické počítače zase dnes již nepoužívají čistý sekvenční výpočetní model, protože často obsahují více CPU.

Výpočetní model, o kterém je zde řeč, publikoval v roce 1945 John von Neumann, Američan uherského původu. Později se však zjistilo, že již v roce 1936 si jej patentoval Němec Konrad Zuse, který je tak vlastně skutečným objevitelem tohoto vynálezu. (Zuseho také některé zdroje uvádějí jako sestrojitele prvního univerzálního (tj. Turing–kompletního) počítače v roce 1941.) Základní strukturu von Neumannova modelu tvoří CPU s aritmeticko–logickou jednotkou, řídicí jednotka (řadič), paměť a vstupně–výstupní jednotka, které jsou všechny propojeny jednou společnou sadou sběrnic (a ta se skládá se tří sběrnic: řídicí, adresové a datové). Ještě jednodušeji lze model popsat jako „CPU + paměť + sběrnice“, případně další externí zařízení připojená ke sběrnici. CPU pak bývá obvykle implementováno pomocí mikroprocesoru, proto dnes pojmy CPU a mikroprocesor či procesor splývají.

CPU je centrální procesní jednotka.

Zastavme se ještě krátce u výše zmíněné aritmeticko–logické jednotky (ALU). Je to obvod provádějící aritmetické a logické operace a je pochopitelně základní součástí prakticky každého CPU. V moderních mikroprocesorech, ať už se používají jako CPU nebo v jiných aplikacích, najdeme takových ALU hned několik, protože stále je to hlavní výpočetní součást mikroprocesoru.

Mnoho českých knih uvádí von Neumannův model jako protiklad k harvardskému modelu, který je podobný, ale používá dvě nezávislé paměti pro kód a data. Za hlavní vlastnost však musíme brát fakt, že oba modely používají paměť pro uložení programu. Alternativou k tomuto přístupu by bylo uložení programu přímo v procesoru, což se dnes u malých zabudovaných počítačů někdy používá, či naopak uložení programu úplně externě, což je nejméně efektivní a používalo se jen u nejstarších počítačů (z technických důvodů).

1.3 Abstrakce hardwaru a architektury

Jak již bylo řečeno a ukázáno, počítače jsou stroje pro člověka poměrně hodně složité. Zaměříme-li se na jejich programování, pak složitost počítače se projevuje zejména tím, že CPU zná jen velmi jednoduché instrukce, jejichž význam je poměrně dosti vzdálen od toho, v jakých pojmech přemýšlí a vymýšlí programy člověk–programátor. Proto bylo od začátku existence počítačů snahou jejich tvůrců umožnit co nejvyšší úroveň abstrakce, čili umožnit psát programy ve člověku srozumitelné formě. Ke každému programovacímu jazyku pak musí existovat překladač či interpret, který převede program z jeho zápisu do nějakého programovacího jazyka na nižší úrovni abstrakce. Na nejnižší úrovni abstrakce je pak samotný jazyk počítače – strojový kód, další ještě nižší úrovně abstrakce jsou pak již hardwarové a programátorů se netýkají.

Operační systém je dalším nástrojem abstrakce. Zatímco samotný (vyšší) programovací jazyk umožňuje vytvářet především algoritmickou část programů v člověku přijatelnější formě, operační systém tvoří vrstvu abstrakce především pro práci s hardwarovými zařízeními, která jsou mimo CPU (včetně abstrakce paměti). Výše uvedený příklad s načítáním souborů z vnějšího média hovoří za vše. Navíc operační systém umožní například abstrahovat fyzický formát média, protože ať už je zdrojem dat děrný štítek, magnetická páska či internet, program operuje

s pojmem soubor a třemi jednoduchými příkazy otevři–načti–zavři. A právě nebýt operačního systému, pojem soubor by neexistoval.

Samotný operační systém samozřejmě může mít mnoho podob, v závislosti na konkrétním počítači. V následujících kapitolách se nejprve krátce seznámíme se strukturou počítače z pohledu programátora, pak ještě stručněji nahlédneme do struktury programů a zbývající kapitoly se budou věnovat jednotlivým základním skupinám funkcí operačního systému. Operační systém je přitom nutno chápat jako software, který má dvě různá rozhraní – jedním komunikuje s hardwarovými zařízeními (tj. zbytkem počítače), druhé pak nabízí ostatním programům jako prostředek k daleko jednodušší komunikaci mezi sebou navzájem a zbytkem počítače. A právě toto druhé rozhraní nás v kurzu Operační systémy bude zajímat především.

Shrnutí

V této krátké úvodní kapitole jsme se pokusili formálně definovat, co je to počítač, z čeho se skládá a na jakém principu funguje. De facto jsme tedy jen připomněli či formálně popsali všeobecně známá fakta, včetně malého okénka do historie počítačů a zmínky o von Neumannovu modelu, který popisuje hardwarovou strukturu počítače. Zároveň jsme se však seznámili s tím, že počítač není jen hardwarem, ale i softwarem; klíčovým prvkem na cestě k pochopení fungování počítače jako celku je naučit se tento stroj vnímat na různých úrovních abstrakce. V tomto duchu budeme ve studiu pokračovat i v následujících kapitolách.

Pojmy k zapamatování

- první, druhá, třetí a čtvrtá generace počítačů
- operační systém
- von Neumannův model počítače
- abstrakce hardwaru
- CPU (centrální procesní jednotka)
- operační paměť
- sběrnice
- I/O zařízení

Kontrolní otázky

1. *Popište von Neumannův model počítače.*
2. *Vysvětlete, proč je v souvislosti s počítačem pojem abstrakce tak důležitý.*

2 Základní prvky počítače

Studijní cíle: Dle von Neumannova modelu jsou základními prvky počítače CPU, operační paměť, I/O zařízení (a ještě sběrnice, která je propojuje). V této kapitole tyto tři části podrobněji prozkoumáme a uděláme si tak základ pro následné studium operačních systémů.

Klíčová slova: CPU, instrukce, operand, operační paměť, unicode, adresování, I/O zařízení

Potřebný čas: 150 minut.

2.1 CPU

2.1.1 Procesor a instrukce

Centrální procesní jednotku, neboli CPU, intuitivně chápeme jako nejdůležitější součást počítače. Obvykle ji neformálně nazýváme „procesor“ nebo „čip“, což je nepřesné, ale velmi zažité, takže se toho budeme držet.

Průvodce studiem

V současných počítačích je CPU implementována mikroprocesorem, proto se často můžeme setkat se záměnou či splýváním těchto dvou pojmů. Nejběžněji užívaný je pak termín „procesor“, kterým vyjadřujeme, že mluvíme o CPU implementované pomocí mikroprocesoru. Má-li počítač mikroprocesorů víc, pojmem „procesor“ se obvykle označuje pouze CPU, ačkoliv ostatní mikroprocesory jsou z technického hlediska ekvivalentní a rovněž mají právo na termín procesor. Stejně tak populární, leč nepřesný je termín „čip“, který se rovněž nesprávně používá jako ekvivalent CPU, ačkoliv čipem je ve skutečnosti každý integrovaný obvod.

Úlohou CPU je vykonávat instrukce.

Procesor má v počítači přesně definovanou úlohu: Vykonává instrukce uložené v operační paměti (pokud se omezíme na von Neumannův model). Každý procesor obsahuje především aritmeticko–logickou jednotku (ALU, vykonává aritmetické a logické operace), řídicí jednotku (řídí chod procesoru) a registry (paměťové buňky umístěné uvnitř procesoru). Registrů je v procesoru obvykle poměrně málo, protože jejich počet má přímý vliv na složitost a cenu celého čipu. Některé registry (na jednodušších počítačích pouze jeden jediný) používá ALU pro výpočty, ostatní pak slouží jako řídicí. Např. v registru IP (instruction pointer) je uchována aktuální pozice vykonávání programu. Říkáme proto, že IP ukazuje na aktuální pozici vykonávání programu. Tato hodnota postupně roste, jak běží program, a může se i náhle změnit v okamžiku, kdy procesor vykoná některou řídicí instrukci (skok, volání podprogramu, či návrat z něj). Dalším řídicím registrem je IR (instruction register), ve kterém má procesor uchovánu právě prováděnou instrukci. Jako třetí příklad uveďme registr SP (stack pointer), který neustále ukazuje na vrchol programového zásobníku. (Zde uváděná jména registrů jsou jen orientační; jednotliví výrobci procesorů ve skutečnosti používají pro stejné registry různá jména.)

Každý procesor má svou „instrukční sadu“, což je sada instrukcí, kterým rozumí. Každá instrukce má nějaký číselný kód (takto je program uložen v paměti) a také název (určen pro lidi). Instrukční kódy i jejich jména jsou v kompetenci jednotlivých výrobců procesorů, proto mezi nimi najdeme poměrně velké rozdíly. Krátká ukázka kódu zapsaného v instrukcích procesoru typu x86 [Bra94, IA32-1] je na obrázku 2.

Operand je parametr instrukce.

Instrukce je příkaz pro procesor.

Jak je vidět na ukázce, instrukce se obvykle zapisují na samostatné řádky. Každá instrukce může mít jeden nebo více operandů (parametrů), dle smyslu dané instrukce. Terminologie je zde trošku matoucí, protože instrukcí nazýváme jak celý řádek, tak jeho první slovo. Instrukce je tedy jak celý příkaz pro procesor včetně jeho parametrů, tak pouze název tohoto příkazu.

```
mov eax, pole
mov ebx, index
mov eax, [eax+ebx*4]
```

Obrázek 2: Ukázka kódu v Assembleru x86

Dle typu procesoru a instrukce se můžeme setkat s různými kombinacemi počtu vstupních a výstupních operandů. Zatímco nejčastěji používané instrukce obvykle mají dvojoperandovou podobu, kde druhý či oba operandy jsou použity jako vstupní a první operand je výstupní (pokud je nějaký výstup). Potřebuje-li instrukce dva vstupy a ještě někam uložit výsledek, první operand je vstupní i výstupní současně. Více než dva operandy mají instrukce jen velmi zřídka, řada instrukcí naopak ani dva operandy nepotřebuje, takže má jen jeden nebo žádný. Některé hodně složité či málo používané instrukce fungují pouze s určitými registry pro vstup či výstup, takže tyto pak rovněž mají méně operandů, než bychom mohli čekat, nebo dokonce žádný. Zdá-li se vám tento systém poněkud nepřehledný, skutečně to tak je. Důvodem je, že instrukce jsou přímo vykonávány hardwarovým čipem, což je technicky velmi složité, a proto struktura instrukční sady a podporované operandy jsou na většině procesorů nějak omezené.

Výsledek instrukce je v 1. operandu.

2.1.2 Vykonání instrukce

Jedním ze sledovaných parametrů každého procesoru je jeho takt. (Ten byl kdysi udáván v kHz, později v MHz a dnes už v GHz.) Takt je udáván hodinovým krystalem, což je součástka mimo procesor, která posílá elektrické impulzy v pravidelných intervalech a tím popohání procesor kupředu. Sám takt však není jediným faktorem ovlivňujícím rychlost procesoru, protože vykonání jedné instrukce může trvat více než jeden tik hodin.

Průvodce studiem

Nejrozšířenější řada procesorů typu x86 je známá tím, že jedna instrukce skutečně není provedena v jednom tiku hodin (1T). Extrémním příkladem jsou třeba instrukce pro násobení a dělení, které na původním procesoru Intel 8086 trvaly až 190T, plus další čas pro dekódování adresy operandu.

Zpracování a vykonání instrukce probíhá v těchto krocích:

1. Přesun na další instrukci
2. Načtení instrukce do CPU – načte se kód instrukce z paměti
3. Dekódování instrukce – zjistí se, co to je za instrukci
4. Výpočet adres operandů
5. Přesun operandů do CPU
6. Vlastní provedení operace
7. Uložení výsledku (a potom zpět k bodu 1)

Tento seznam 7 kroků je obecný a netýká se žádného konkrétního procesoru, navíc jednotlivé kroky mohou být vynechány v případě, že daná konkrétní instrukce je nepotřebuje. Proto také i vykonání stejné instrukce může trvat různě dlouhou dobu podle toho, jaké má tato instrukce operandy.

Tento model vykonávání instrukce v sedmi krocích tedy vlastně říká, že v CPU je 7 samostatných částí, z nichž každá má zcela jinou úlohu. Přitom vždy jedna část CPU pracuje a zbylých 6 čeká.

Takový model by byl v praxi samozřejmě značně neefektivní, takže moderní procesory jsou navrženy tak, aby co nejvíc z těchto kroků bylo prováděno paralelně (tj. současně). Například instrukce na Pentiu 4 trvají běžně 1–3T (dle operandů), jelikož kroky 1–4 jsou prováděny v předstihu a trvají 0T (v ideálním případě, tj. „když se to stihne“) a zbylé tři kroky trvají maximálně 1T. Moderní procesory mají navíc několik ALU, které mohou pracovat současně (pro programátora je to bez práce, neboť řídicí jednotka sama určuje, kterou instrukci která ALU zpracuje, registry jsou společné pro celý procesor).

2.1.3 Paralelní vykonávání instrukcí

Zatímco například kdysi populární 8bitový procesor Zilog Z80 (nejprodávanější 8bitový čip všech dob) používal jednoduchou posloupnost zpracování instrukcí ve 4 krocích, takže každá instrukce trvala 4T nebo násobek 4T, u dnešních procesorů je takové mrhání časem těžko představitelné. Instrukce jsou prováděny paralelně, výrobci čipů k tomu používají celou řadu technik. Skutečnou rychlost provádění celých programů tak často více než takt procesoru ovlivňuje rychlost paměťového čipu či způsob, jakým překladač programovacího jazyka dokáže seskládat instrukce do řady za sebou, aby tyto skutečně mohly být vykonány paralelně.

Pipelining je základní formou paralelizmu.

Základním modelem paralelního zpracování instrukcí je *pipelining*. Ten funguje tak, že každá ze 7 částí (našeho modelového) CPU po vykonání své práce na instrukci okamžitě přechází na další instrukci. Pipelining v praxi nemusí být úplný, tj. ne každá část CPU toto musí podporovat. Pokud úplný je, dostaneme se teoreticky až k situaci, že v každém tiku hodin se začne zpracovávat další instrukce. Toto je samozřejmě omezeno tím, že vlastní vykonání instrukce nemusí trvat pouze 1T (viz příklad násobení a dělení výše).

Dalším omezujícím faktorem pipeliningu jsou situace, kdy jedna instrukce závisí na druhé. Základním příkladem je, když vstupním parametrem instrukce je výsledek z přímo předchozí či některé blízké předchozí instrukce. V takovém případě je nutno zajistit, aby instrukce „nepředbíhaly“, tj. aby pozdější instrukce nezačala dříve, než všechny ty, na kterých závisí, byly hotové. Procesory x86 mají tuto situaci ošetřenou hardwarově, navenek se tedy vše chová tak, jakoby pipelining zaveden nebyl (k „předbíhání“ instrukcí a chybám při výpočtu tedy nedochází). Jiné procesory ale mohou mít třeba jiný paměťový model (např. Intel Itanium 2), který se projevuje právě chybným chováním při pipeliningu. Praktické zkušenosti s těmito procesory ukázaly, že ošetření předbíhání instrukcí v překladačích programovacích jazyků je téměř neřešitelný problém; tento model se tedy v praxi příliš neosvědčil.

I když procesor sám ošetří, aby nedošlo k předbíhání instrukcí, jejich vzájemná závislost způsobí, že pipelining nepřinese žádné zrychlení výpočtu. Extrémem jsou zde instrukce podmíněných skoků – velmi často používané instrukce, pomocí nichž je obvykle implementováno větvení programu (příkaz **if** apod.). Tyto instrukce mohou změnit adresu vykonávání kódu podle toho, jaký je výsledek některé předchozí instrukce. Všechny instrukce následující za instrukcí podmíněného skoku na ní tedy přímo závisí, protože dokud není skoková podmínka vyhodnocena, nevíme vlastně, zda následující instrukce se vůbec mají vykonávat. Skokové instrukce tedy způsobují úplné zastavení pipeliningu a jsou nejméně efektivními instrukcemi z hlediska možného paralelního běhu procesoru.

Průvodce studiem

Prvním procesorem řady x86, který používal pipelining, byl i486dx. Tento procesor zpracovával instrukce v 5 krocích, které mohly běžet současně. Navíc jednotky dekódování instrukce měl hned dvě. I přes poměrně černě vylíčené chování u instrukcí podmíněných skoků dokázal tento procesor díky pipeliningu na 25MHz pracovat rychleji než Intel 386dx na 40MHz.

Dalším modelem umožňujícím paralelní vykonávání instrukcí v jednom procesoru je *superskalární architektura*. Ta se v běžných počítačích objevila s procesorem Intel Pentium. Tyto procesory (a také všechny následující modely řady x86) obsahují více než jednu ALU. Procesor tak může zpracovávat dvě nebo více instrukcí současně, podobně jako při pipeliningu. Obsažení dvou ALU také umožňuje sofistikovanější chování u podmíněných skoků – každá ALU jde jinou cestou (protože podmínka zcela jistě buď platí, nebo neplatí). Každá z ALU jednotek tak vlastně vykonává dopředu nějaký kód, aniž tuší, zda to nedělá zbytečně. V okamžiku vyhodnocení podmínky je jedna z ALU vynulována, zatímco druhá mezitím využila čas produktivně a vykonala část programu dopředu. Tím je procesor ještě rychlejší než při pipeliningu.

Průvodce studiem

Superskalární architektura dnešních procesorů způsobuje, že u některých programů je skutečná rychlost vykonávání instrukcí vyšší než takt procesoru. Průměrný čas potřebný pro vykonání jedné instrukce je tedy méně než 1T, často i docela výrazně.

Má-li procesor více než dvě ALU, pak rozdělení na dvě alternativní větve u podmíněného skoku není zrovna optimálním využitím prostředků procesoru. Proto jsou využívány i jiné techniky k optimalizaci skoků. Předvídání větví (branch prediction) je technika předvídání, zda daná podmínka bude, či nebude platit. Některé procesory toto dělají staticky, čili u každé instrukce podmíněného skoku je dáno, kterou variantu (platnost/neplatnost podmínky) upřednostňuje. Překladače vyšších jazyků s tím pak mohou počítat a skládat podle toho instrukce. Dokonalejší procesory umějí i dynamické předvídání, které je velmi vhodné u cyklů (konstrukce typu for, while apod.). Takový procesor si totiž u několika posledních podmíněných skoků pamatuje, zda jejich podmínky byly či nebyly splněny, a pokud se dostane na stejné místo programu (dříve než tuto zkušenostní informaci zapomene), chová se dle předchozí zkušenosti. Právě pro běžné konstrukce cyklů je toto velmi výhodné (splete se jen jednou nebo dvakrát – možná při prvním a určitě při posledním průchodu cyklem).

2.1.4 Instrukční sada

Z hlediska instrukční sady rozlišujeme dva základní typy procesorů:

CISC procesor má úplnou sadu instrukcí.

CISC (complex instruction set computer) obsahuje tzv. úplnou sadu instrukcí. Jedná se o složité procesory nabízející velké množství instrukcí a běžné operace umějí také provádět přímo v operační paměti. Instrukce v těchto procesorech často přímo podporují operace prováděné vyššími programovacími jazyky (práce se zásobníkem, lokálními proměnnými, funkcemi apod.). Vykonání jedné instrukce obvykle trvá déle než 1T. Příkladem CISC procesoru je řada x86, x64 nebo také výše zmíněný Z80.

RISC procesor má redukovanou sadu instrukcí.

RISC (reduced instruction set computer) vychází z teze, že stejného výsledku lze dosáhnout také pomocí posloupnosti jednoduchých instrukcí. Tyto procesory mají jen základní sadu instrukcí, jejich design a výroba je tedy jednodušší. Vykonání jedné instrukce trvá 1T. Příkladem RISC procesoru je PowerPC (používají např. starší modely Macintosh a řada herních konzolí).

Klasické počítače PC používají procesory řady x86, tedy typu CISC. Obrovská složitost moderních procesorů řady x86 však přiměla jejich výrobce, aby přešli na RISC technologii. Všechny dnešní x86 procesory jsou tedy ve skutečnosti RISC procesory, které simulují složitější instrukce pomocí zabudovaných mikroprogramů, čímž se navenek tváří jako CISC.

Starší procesory bývaly programovány přímo, tedy pomocí procesorových instrukcí ve strojovém kódu či assembleru. Aby programování bylo jednodušší, výrobci procesorů se tehdy snažili implementovat co největší funkcionalitu přímo do jediné instrukce. Tím se taky šetřila tehdy velmi drahá a pomalá paměť – když byly instrukce chytřejší, programy mohly být kratší.

Dnes je však situace zcela odlišná. Paměťové čipy jsou poměrně levné a rychlé, takže na velikosti programů až tak nezáleží, a implementace některých moderních technik paralelního zpracování instrukcí u CISC procesorů je technicky velmi obtížná z důvodu jejich složitosti. Navíc se dnes používají téměř výhradně vyšší programovací jazyky, jejichž překladače si obvykle vystačí s několika málo instrukcemi. Řada CISC instrukcí je dnes v procesorech tedy vlastně zbytečně. Z toho důvodu dnes jednoznačně „vítězí“ RISC model.

2.2 Operační paměť

Druhou významnou součástí počítačů je paměť. Význam paměti je zcela jasný: Počítač si v ní pamatuje data, často navíc i program.

Paměti bývá zvykem dělit na vnitřní a vnější. Operační paměť je pamětí vnitřní, zatímco například hard disk je pamětí vnější. Nejoblíbenější jsou pak obecně paměti, které procesoru umožňují data zapisovat i číst, protože využitelnost pamětí jen pro čtení je samozřejmě omezená. Zařízení, která naopak data umějí jen zapisovat (tiskárny apod.) obvykle za paměti nepovažujeme.

Podobně jako u procesorů, i u paměti se velmi často používá terminologie, která je velmi zažitá, leč chybná. Správně bychom totiž měli vždy hovořit o operační paměti, protože existují i jiné druhy pamětí, v praxi se však při vyslovení jednoduchého „paměť“ obvykle myslí právě paměť operační. Zůstaneme tedy u tohoto jednoduchého termínu, budeme přitom vždy mít na mysli operační paměť z von Neumannova modelu počítače.

2.2.1 Uložení čísel v počítači

Čísla jsou základním typem informace, který v počítači uchováváme. Ke zpracování čísel ostatně byly počítače původně vytvořeny. Jelikož paměť má podobu řady bitů, je zřejmé, že uložení úplně libovolného čísla do takové formy není možné. Ať už chceme ukládat pouze čísla celá, nebo i racionální či reálná, jsme vždy nějak omezeni. Stejná omezení pak mají dopad i na aritmetické operace – je-li totiž samotné uložení čísel omezeno, pak i operace s těmito čísly mohou být těmito omezeními postiženy.

Nejjednodušší situace je u prvně zmíněných celých čísel. Paměť dokáže uchovat libovolné celé číslo, kladné i záporné, jehož velikost nepřesahuje určitou mez. Víme, že do 1 bajtu se vejdou čísla v rozsahu 0 až 255 (či -128 až +127), atd. Naprostá většina procesorů používá *doplňkový kód*, díky kterému řada aritmetických operací může být prováděna bez ohledu na to, zda je číslo kladné či záporné, a na to, zda samotný datový typ je znaménkový nebo neznaménkový. Doplňkový kód již znáte z dřívějšího studia, připomeňme jen, že změnu znaménka u čísla provedeme tak, že převrátíme hodnoty všech jeho bitů a přičteme k němu jedničku. (Pro procvičení si dokažte!)

Doplňkový kód umožňuje snadnou práci se zápornými čísly.

Dvě mezní situace, které mohou nastat u celých čísel, jsou *přetečení* a *podtečení*. Přetečení nastává, je-li výsledek větší než maximální hodnota uložitelná do paměti. V takovém případě hodnota v paměti takzvaně „přeteče“ a následující hodnotou za největší možnou je nejmenší možná. U sčítání může při jednom sečtení dvou čísel přetéct hodnota jen jednou, zatímco u násobení může přetéct hodnota mnohokrát. (Např. 129*128 = 16512, ale v bajtu bude hodnota 128.) Podtečení je opačná situace, kdy výsledek je menší než nejmenší možná hodnota. Chování přetečení a podtečení je zcela stejné.

Průvodce studiem

Matematicky funguje přetečení a podtečení tak, že ze skutečného výsledku se zachová jen tolik nejnižších bitů, kolik se vejde do paměti. Výše uvedený příklad 129*128 můžeme pro

ukázku zapsat v šestnáctkové soustavě, kde každému bajtu odpovídají přesně dvě cifry: 81*80 = 4080, do jednoho bajtu se tedy vejde šestnáctkové číslo 80, což je v desítkové soustavě právě 128. Tímto způsobem funguje přetečení i podtečení a to pro kladná i záporná čísla.

Přetečení a podtečení je jednou z nejdůležitějších událostí, která může při běhu nastat, proto je pro jejich zpracování vyhrazen speciální bit v jednom registru. Tento registr se na x86 jmenuje eflags, samotný bit je pak na většině procesorů označen jako carry, či zkratkou CY, CF nebo C. Při úplně každé aritmetické operaci se CY nastaví na nulu, pokud k přetečení/podtečení nedošlo, či na jedničku, pokud k přetečení/podtečení došlo. Tato hodnota zůstává v CY tak dlouho, než ji jiná aritmetická operace přepíše.

U vícebajtových čísel (tj. těch, které v paměti zabírají více než jeden bajt) existuje několik variant, jak je do paměti uložit. Každý procesor bohužel používá jiný přístup. Je v zásadě několik možných variant:

Big endian ukládá čísla od nejvyšších řádů po nejnižší.

Big endian systém ukládá čísla od nejvyšších řádů po nejnižší. Tento systém tedy odpovídá našem způsobu zápisu čísla (de facto pozpátku, protože číslo „roste“ doleva). Např. číslo ABCD v šestnáctkové soustavě je uloženo jako AB CD, číslo 1 je uloženo ve 4bajtovém zápisu jako 00 00 00 01, číslo 65536 je čtyřbajtově uloženo jako 00 01 00 00. Tento způsob používají např. procesory Motorola 68k.

Little endian ukládá čísla od nejnižších řádů po nejvyšší.

Little endian systém ukládá čísla od nejnižšího řádu po nejvyšší. Výhodou je, že při takovém zápisu můžeme číslo zvětšovat bez změny adresy. Např. číslo ABCD v šestnáctkové soustavě je uloženo jako CD AB, číslo 1 je uloženo ve 4bajtovém zápisu jako 01 00 00 00, číslo 65536 je čtyřbajtově uloženo jako 00 00 01 00. Tento způsob používají procesory řady x86, Z80 aj.

Bi–endian je označení pro procesory podporující oba výše uvedené systémy. Příkladem je Intel řady IA64, PowerPC aj.

Middle endian je označení pro procesory používající jiné zvláštní řazení bajtů u čísel.

Průvodce studiem

Názvy little a big endian pocházejí z románu Guliverovy cesty, kde dva národy vedou nesmiřitelný boj o to, na které špičce se má natloukat vajíčko. Stejně jako u vajíčka, i u procesorů je ve skutečnosti samozřejmě zcela jedno, který systém použijeme, objektivně jsou oba stejně dobré. Domluva na systému zápisu celých čísel je však velmi důležitá při přenosu dat (komunikaci) mezi různými systémy (např. v internetu).

Průvodce studiem

Z našeho lidského hlediska je u „endianů“ vidět jeden paradox. Aniž bychom si to nějak hlouběji uvědomovali, jelikož používáme arabská čísla a písmo píšeme zleva doprava, přirozený formát záznamu čísel je pro nás big endian. Jenže Arabové píší zprava doleva a pochopitelně používají stejný zápis čísel, tj. tentýž zápis je pro ně little endian. Tento paradox vznikl tím, že jsme převzali arabská čísla a naučili se je psát pozpátku, přestože to je poněkud méně komfortní.

Dalším typem čísel, se kterými procesor umí pracovat, jsou tzv. *BCD čísla* (binary coded decimal). Jedná se o málo používanou formu čísel, která je pozůstatkem z dávných dob, kdy mikroprocesory byly určeny především pro matematické kalkulátory (kalkulačky). Každé 4 bity

zde uchovávají jednu desítkovou cifru. V bajtu jsou tedy dvě desítkové cifry – tento zápis je dosti neefektivní, ale odpovídá způsobu, jakým fungují kalkulačky. Vyšší programovací jazyky BCD nepoužívají vůbec.

Třetím způsobem uložení čísel v počítači jsou *čísla v plovoucí řádové čárce* (FP, floating point numbers). Obvykle se jim chybně říká reálná, i když jde pouze o čísla racionální. (Onen správný termín se neujal především z toho důvodu, že je přece jen poněkud dlouhý.) FP–čísla jsou opět uložena v několika bitech, v praxi nejčastěji zabírají 8 bajtů, případně 4 bajty. Do těchto 8 či 4 bajtů jsou zakódovány 3 části čísla: znaménko (1 bit), mantisa (52 či 23 bitů) a exponent (zbytek, čili 11 či 8 bitů). Toto kódování je naštěstí unifikováno (je jednotné pro různé typy procesorů). Skutečná hodnota čísla se vypočte takto:

FP čísla jsou na všech počítačích uložena stejně.

$$hodnota = znaménko * mantisa * základ^{exponent}$$

kde exponent může být kladný i záporný a základem bývá obvykle 2, výjimečně i 10.

Tento způsob reprezentace čísel je poměrně praktický, neboť bez ohledu na velikost čísla je vždy zaručena přesnost na určitý počet řádů. Hardwarově nejefektivnější je, aby základem byla dvojka (řada operací se pak hardwarově jednoduše implementuje); nevýhodou tohoto řešení však je, že některá běžná čísla nelze uložit zcela přesně (např. číslo 0.1). Základ 10 se používá jen výjimečně a není podporován hardwarově.

U FP čísel má také jiný význam podtečení – hovoříme o něm tehdy, když je hodnota (bez ohledu na znaménko) mezi nulou a nejmenším zobrazitelným nenulovým číslem. Přetečení je situace, kdy je číslo (bez ohledu na znaménko) naopak větší než největší zobrazitelné číslo.

2.2.2 Uložení znaků v paměti

Uložení znaků v počítači je poněkud jednodušší než v případě čísel. Historicky se nejvíce rozšířil model používající 7bitové kódování ASCII. Do jednoho bajtu se tak vejde právě jeden znak (americké abecedy). Tento model vznikl původně pro telegrafy, kde nejvyšší bit byl využit pro kontrolní součet (CRC). Na počítačích však byl tento prostor využit pro národní abecedy (např. naše české znaky se do 7 bitů nevejdou), nehovoříme pak však již o ASCII.

ASCII je 7bitový kód pro americkou angličtinu.

Osmibitových kódování znaků existuje celá řada, dokonce i pro češtinu jich vzniklo poměrně hodně. Tato nejednotnost je samozřejmě nešťastná, dokonce ani Windows, Linux a Macintosh nedokázaly použít jednotné kódování. (Windows a Linux se dokonce liší jen u tří písmen, což je až humorné.) Zatímco MS-DOS dlouhá léta používal kódování CP852 (Latin 2), řada českých programů dávala přednost kódování KEYBCS2 (CP895), které obsahovalo kromě češtiny i znaky pro rámečky. Windows pak přišlo s kódem CP1250, který je vzhledem k rozšíření Windows dnes nejrozšířenější, ale zpět do DOSu se nikdy neprosadil. Linux pak používá ISO-8859-2, což je de jure standard, ovšem vnucený nám z USA. A Macintosh má další jiné kódování. (Podobně jsou na tom i ostatní počítače – každý pes, jiná ves.)

Nevýhodu těchto 8bitových kódování řeší *Unicode* – jednotný světový kód společný pro všechny národní abecedy. Každý znak v Unicode měl původně 2 bajty, později se přešlo na 4 bajty. Znaky všech národních abeced se však vešly již do oněch prvních dvou bajtů, proto se v praxi některé systémy omezují na 2bajtové znaky Unicode. Windows či Java používají kódování UTF-16, kde většina znaků má 2 bajty (první dva bajty Unicode) a pouze v nutných případech jsou znaky delší. Prvních 256 znaků je zde stejných jako v kódu ISO-8859-1 (západní latinka). Do souborů se Unicode obvykle ukládá v kódu UTF-8, kde znaky ASCII mají 1 bajt a ostatní znaky mají 2 až 4 bajty (pro češtinu vždy dva bajty). Existuje ještě několik dalších kódování pro Unicode (např. 7bitové UTF-7), ty se ale téměř nepoužívají.

Unicode je jednotný kód pro všechny světové jazyky.

Průvodce studiem

Zajímavostí Unicode je, že některé znaky jsou v tomto kódu vícekrát. Chceme-li pak třeba zjistit, zda jsou dva řetězce stejné, je třeba nejprve provést tzv. *normalizaci*, kdy se všechny výskyty těchto znaků sjednotí na stejné číslo. Pak je možno provést porovnání klasickým způsobem.

2.2.3 Adresování paměti

Paměť je jako jedno velké pole bajtů.

Operační paměť je lineární struktura s pevnou délkou a náhodným přístupem (čili je to totéž jako „pole“ ve vyšších programovacích jazycích). Pojem *náhodný přístup* znamená, že procesor může číst a zapisovat libovolné paměťové buňky. Ačkoliv fyzicky mívají různé typy pamětí buňky rozličných velikostí, z hlediska softwarového se ujal jednotný model používající „bajt“ – každá paměťová buňka tedy má velikost jednoho bajtu (8 bitů). Určení, se kterou buňkou chce procesor pracovat, říkáme *adresace paměti*. Každá buňka paměti má tedy svou adresu, což je číslo v rozsahu 0 až velikost paměti–1.

Tento jednoduchý a tradiční způsob organizace paměti nazýváme *lineární* (l. adresování, l. adresa, atp.). Je však nevhodný, neboť neumožňuje jednoduchý přenos dat či kódu programu na jiné místo v paměti – při přemístění se totiž mění adresy. Ať už přemístíme kód, či data, musíme softwarově přepočítat všechny adresy.

Většina současných procesorů umožňuje i nějaký jiný adresovací režim než lineární. Stačí přitom rozdělit lineární adresu na dvě složky: *bázi* a *posunutí*. Báze určuje začátek programu či dat, posunutí pak relativní adresu vzhledem k bázi. Procesory řady x86 mají tento princip implementován pomocí tzv. *segmentového adresování*, kde se adresa vypočítá jako *segment* + *offset*.[1]

Na 16bitových procesorech má každá z těchto dvou složek 16 bitů, celá adresa má tedy 32 bitů. (Takové 32bitové adrese se říká *far pointer* (vzdálený pointer). Opakem je 16bitový *near pointer*, což je pouze offset bez určení segmentu.) Převod na lineární adresu je proveden jednoduchým vzorcem

$$lineární\ adresa = (segment << 4) + offset$$

Segment je tedy posunut o 4 bity a sečten s offsetem. Celkově tedy můžeme adresovat (jen) 20 bitů, čili 1MB paměti. (Tento způsob adresace používá systém MS-DOS. Část paměti využívá systém, proto je velikost reálně využitelné paměti obvykle méně než 640KB.)

Deskriptor popisuje segment paměti (jeho začátek, délku atd.).

Novější procesory x86 pak umožňují ještě další adresovací režimy. K podrobnějšímu popisu adresování se dostaneme později, v kapitole o správě paměti v operačním systému. Nyní se proto jen seznámíme se základním principem. Adresování ve 32bitovém režimu procesorů x86 používá opět dvě složky: *selektor* + *offset*. Selektor má 16 bitů a je to ukazatel do tabulky *deskriptorů*. Každý deskriptor je záznam v tabulce, který popisuje nějaký segment. (Deskriptor popisuje segment. Selektor jej vybírá.) Tímto způsobem je umožněno, aby programy používaly stále jen 16 bitů k popisu segmentů i za hranicí 1MB. Po přepočtu adresy pomocí selektoru se ještě přičte offset; ten má samozřejmě 32 bitů. Celý far pointer má tedy 48 bitů; tato neobvyklá velikost může být matoucí, nicméně v praxi se většinou používají jen near pointery, neboť 32bitový offset adresuje až 4GB paměti. U každého programu tedy stačí při startu pomocí deskriptoru a selektoru nastavit jeho bázi (tj. určit, kde v paměti začíná), a pak už se pracuje jen s 32bitovým offsetem. Programy tedy vlastně pracují jakoby s lineární adresou, což je dosti výhodné pro různé výpočty s adresami apod.

GDT je globální tabulka deskriptorů.

Tabulky deskriptorů spravuje pouze operační systém, samotné programy je nemohou ovliv-

[1] Offset je anglický termín pro posunutí.

ňovat. Sám procesor má jednu globální tabulku GDT (global descriptor table), ukazuje na ni registr GDTR. Deskriptory v této tabulce mohou odkazovat na další tabulky deskriptorů, z nichž právě jedna je vždy aktivní jako LDT (local descriptor table). Každý program tedy obvykle má svou LDT (což je pochopitelné), na kterou ukazuje registr LDTR, a navíc může používat přímo i deskriptory v GDT. Deskriptory, které program používá, pak již jsou samotné segmenty pro kód či data programu. Každý deskriptorový záznam obsahuje především údaje o počátku, délce a typu segmentu, povolení čtení/zápisu/spouštění, atp.

Každý program má svou vlastní tabulku deskriptorů LDT.

16bitové segmentové registry, které původně ukazovaly někam do prvního 1MB paměti, nyní ukazují do tabulky LDT nebo GDT. Registry jsou označovány jako selektory segmentů a jejich 16bitová hodnota se skládá z indexu do tabulky deskriptorů (13 bitů – maximálně tedy může být 8192 deskriptorů v jedné tabulce), bitu rozlišujícího GDT a LDT a 2 bitů oprávnění (bude probráno v kapitole 11.2 na straně 115). Registr LDTR je sám segmentovým registrem ukazujícím do GDT; GDT už nemá kam ukazovat, takže obsahuje přímo lineární adresu tabulky deskriptorů a její velikost (obojí s přesností na bajt).

Procesory x86 také používají stránkování, což je další stupeň abstrakce. Při stránkování používá procesor další tabulky, pomocí kterých se mapují 4KB bloky fyzické paměti (rámce) do jiného pořadí. Vznikají tak 4KB rámce. Výhodou stránkování je, že umožňuje, aby více programů bylo současně v paměti, každý používal různé „kousky“ fyzické paměti, a přitom z pohledu tohoto programu se zdálo, že má pro sebe jeden lineární nekouskovaný prostor. (Opět totiž platí, že stránkování zajišťuje operační systém a samotné programy o něm nevědí.) Stránkování také umožňuje odkládat část vnitřní paměti na disk. Programy tedy mohou využívat více operační paměti, než kolik máme v počítači vnitřní paměti (RAM).

2.2.4 Typy adres v operandech

Procesory CISC u většiny instrukcí umožňují, aby tyto ukazovaly přímo na paměť. Vykonání operace nad nějakou proměnnou tedy lze provést bez toho, abychom hodnotu nejprve načetli do registru, pak provedli operaci, a pak výsledek uložili zpět do paměti.

Procesory x86 umožňují určit adresu dvěma způsoby:

Přímá adresa ukazuje na pevné místo v paměti, čili jedná se o offset (viz definici výše). Tento způsob je efektivní z hlediska běhu programu, ale není flexibilní. V praxi je použitelný jen pro určení adres globálních proměnných a podprogramů, které jsou pochopitelně vždy na stejném místě v paměti (vzhledem k bázi).

Nepřímá adresa ukazuje na místo v paměti nepřímo. Adresa se tentokrát vypočítá z hodnot registrů, případně přičtením nějakého dalšího posunutí. Tento způsob adresování je méně efektivní (pomalejší), ale je naprosto flexibilní.

16bitové procesory x86 mají nepřímé adresování možné ve formě

$$adresa = posunutí + si/di + bx$$

Jednotlivé složky (sčítance) jsou nepovinné. Posunutí je přímá hodnota (offset). Si, di a bx jsou registry. (Z prvních dvou je možno použít jen jeden: tedy buď si, nebo di.)

32bitové procesory x86 mají nepřímé adresování dokonalejší. Adresu lze určit pomocí až dvou registrů a posunutí takto:

$$adresa = posunutí + báze + index * faktor$$

Jednotlivé složky (sčítance) jsou opět nepovinné. Báze a index jsou registry, je možno použít téměř libovolné všeobecné registry. Posunutí je přímá hodnota a faktor je číslo 1, 2, 4 nebo 8.

Jednotlivé součásti adresy samozřejmě nejsou povinné a používáme je dle potřeby. Například pro přístup do globálního pole použijeme formát adresy $posunutí + index * faktor$, kde

posunutí je adresa pole, index je číslo buňky v poli a faktor je velikost jedné buňky. Druhý příklad: Pro přístup do lokálního skaláru (jednoduché proměnné) použijeme posunutí + báze, kde báze je adresa rámce aktuální funkce na zásobníku a posunutí je pozice naší proměnné v rámci funkce.

2.2.5 Cache paměť

Cache je malá rychlá paměť.

Cache neboli vyrovnávací paměť je zvláštní pomocná paměť, která je malá, ale velmi rychlá. Můžeme ji nalézt v procesorech, ale i v jiných hardwarových i softwarových prvcích. Smyslem cache je zrychlit opakované čtení paměti tím, že to, co bylo nedávno čteno, je dočasně uchováno také v cache a při opakovaném čtení téhož se data čtou z cache namísto opravdové paměti. Moderní procesory mají různé typy cache pro zrychlení různých specifických operací. Cache obvykle pomáhá i se zápisem do paměti tím, že při současném čtení i zápisu paměti se data dočasně zapisují jen do cache a přednost má čtení paměti. Teprve později, až je procesor méně vytížen, se data z cache zapisují do skutečné paměti. Fungování cache paměti je přitom dosti složité, naštěstí nás nemusí až tak trápit, neboť pro programy je její existence zcela transparentní (čili nevědí o ní).

Přínos cache je vyšší, pokud programy dodržují tzv. *časově–prostorovou lokalitu*, tj. v určitém krátkém čase přistupují pokud možno jen do jedné lokality paměti. Proto jsou při běhu programy používající globální proměnné nesystematicky rozseté po paměti pomalejší než objektově orientované programy, kde je většina dat jen v lokálních proměnných funkce a ve „svém“ objektu (v řadě jazyků na adrese označené jako „this“).

2.3 I/O zařízení

Třetí částí počítače jsou dle von Neumannova schématu *I/O zařízení* (neboli *periferie*). Sem spadá vše, co je mimo jádro počítače tvořené procesorem a vnitřní pamětí. Zastavíme se u nich jen stručně, neboť jich existuje velké množství a jejich jednotlivé funkce nás až tak nezajímají. Tím, jak si se zařízeními poradí operační systém, se navíc budeme zabývat až v pozdějších kapitolách. Nyní si proto jen představme několik základních principů.

V dnešních počítačích jsou I/O zařízení poměrně inteligentní, původní von Neumannův model však počítal s tím, že to budou spíše jednoduchá zařízení umožňující jen základní formu komunikace s procesorem na bázi příkazů, které dává procesor zařízení, a odpovědi, kterou dává zařízení procesoru. Zařízení tedy např. nemá mít přímý přístup do operační paměti (i když v současné praxi je tomu obvykle naopak).

Obecně platí, že spolupráce se zařízeními patří ke spíše složitějším postupům, protože je nutno velmi pečlivě sledovat, v jakém stavu zařízení je, zda vůbec očekává příkazy od procesoru, zda je neposíláme moc rychle atd. Při komunikaci je navíc nutno počítat s tím, že mezi vysláním povelu a obdržením odpovědi ze zařízení je vždy nějaká prodleva. Situace bývá řešena jedním ze dvou způsobů:

Aktivní čekání – Procesor vše sám přímo řídí a sleduje. Během čekání na odpověď od zařízení může průběžně kontrolovat situaci, nemůže však vykonávat žádný jiný program. Tento způsob komunikace se používal především v minulosti, dnes se mu vyhýbáme, neboť způsobuje „zamrznutí“ počítače. (Výjimkou jsou některé těžko pochopitelné chyby v návrhu systému, např. zamrznutí Windows při otevírání CD z mechaniky apod.)

Využití přerušení – Procesor jen zařízení vyšle povel a dál pokračuje ve vykonávání jiného programu. Jakmile je připraven výsledek, zařízení vyvolá přerušení, čímž procesor přejde do podprogramu obsluhy přerušení. Tam vykomunikuje se zařízením potřebné detaily a ukončí tento podprogram. Tím se dostane zpět do místa, kde byl přerušen, a běh programu pokračuje.

Využití přerušení má obrovskou výhodu v tom, že je efektivně využit výpočetní výkon procesoru, proto má tato metoda komunikace se zařízeními jednoznačně přednost. Nevýhodou je podstatně složitější návrh jak hardwaru počítače, tak operačního systému a programů, které s přerušením nějak pracují. Proto se v minulosti, kdy počítače i programy byly jednodušší, používalo přerušení méně než dnes.

Výpočetně nejnáročnější operací je čtení či zápis většího množství dat. Například načtení dat z disku začne tím, že procesor dá diskovému zařízení povel ke čtení. Pak čeká, až jsou data připravena. Potom je však třeba data načíst, což znamená absolvovat kolečko požadavek–odpověď pro každý bajt dat. Během tohoto kolečka musí procesor pečlivě sledovat, zda data nežádá moc rychle, protože procesor je obvykle mnohem rychlejší než I/O zařízení. Jednotlivé bajty dat zařízení posílá po jednom po datové sběrnici. Procesor si je průběžně skládá někam do paměti, zatímco samo zařízení operační paměť přímo nevidí.

Alternativou k tomuto postupu je využití modernější techniky zvané DMA (Direct Memory Access – přímý přístup do paměti), která umožňuje I/O zařízením přímo přistupovat do paměti. DMA lze použít jen tehdy, podporuje-li ho počítač, operační systém i zařízení. Na počítačích PC je funkce DMA zajištěna obvodem DMAC (DMA Controller). Jedná se o inteligentní zařízení připojené na sběrnici, které umí komunikovat jak s I/O zařízeními, tak s operační pamětí. Čtení z disku pak může proběhnout tak, že procesor naprogramuje DMAC k přenosu dat ze zařízení do paměti a pak dá povel zařízení, aby zahájilo čtení disku. Procesor pak může provádět jiný program, zatímco DMAC postupně odebírá data přicházející ze zařízení a ukládá je na určené místo do paměti. Jedno omezení zde však zůstává: Data do operační paměti stále proudí po téže sběrnici, kterou procesor používá pro čtení svého programu a dat. DMAC tak vlastně „krade“ sběrnici, neboť se chová jako druhý inteligentní procesor na sběrnici. Na sběrnici pak může docházet ke kolizím, dalším limitujícím faktorem je samotná rychlost paměťových čipů. Pokud totiž CPU i DMAC chtějí pracovat současně (což je jejich cílem), zákonitě na omezenou rychlost paměťového čipu narazí.

Problémem zde popsaného původního DMAC je především dnes již nedostačující rychlost a také nemožnost adresovat více než 16MB paměti. Dnešní zařízení však tento obvod již nepotřebují, neboť sběrnice PCI (a novější) umožňují zařízením přímý přístup do operační paměti. Zatímco starší zařízení byla napojena na sběrnici ISA, která byla oddělena od sběrnice jádra počítače pomocí ISA řadiče poskytujícím pouze základní funkcionalitu na frekvenci 8MHz, novější sběrnice (např. PCI) jsou jak rychlejší, tak chytřejší. Sběrnice AGP určená pro grafické karty je dokonce přímo spřažena se sběrnicí, na které je připojen procesor, takže AGP zařízení má opravdu přímý přístup i bez dalšího řadiče. Takový přístup je tedy na rozdíl od PCI nejen jednoduchý, ale i rychlý; obráceně i procesor má přímý a rychlý přístup do paměti na AGP grafické kartě.

Moderní zařízení používají sériové sběrnice.

Průvodce studiem

Jmenované sběrnice PCI a AGP jsou samozřejmě jen dvěma příklady z mnoha. Důvody, proč zařízení v počítačích PC nejsou přímo připojena na stejnou sběrnici jako procesor, jsou především v rovině elektrické. Délka vedení by byla několik desítek centimetrů a na takovou vzdálenost není technicky možné zajistit dostatečně rychlou a spolehlivou komunikaci. Každé zařízení navíc sběrnici dále zatěžuje, protože jejím používáním z jednotlivých linek odebírá proud. Společnou sběrnici proto mívají jen pomalejší a hlavně pouze malé počítače, a to doslova.

Dalším způsobem, jak zlepšit fungování sběrnice, je přechod z paralelního na sériový model. Zatímco u počítačů hovoříme, že jsou např. 32bitové, když mají 32bitovou sběrnici, nová sběrnice PCI Express je 1bitová. Díky tomu je její realizace elektricky mnohem jednodušší a její provoz spolehlivější. Sériová sběrnice však musí běžet mnohem rychleji než paralelní, např. 32× rychleji než 32bitová PCI, aby dosáhla její komunikační propustnosti. Proto se k sériovému typu sběrnic přistoupilo teprve v okamžiku, kdy byly k dispozici

dostatečně rychlé a levné řadiče. (Dalšími příklady sériových sběrnic jsou USB či SATA.)

Alternativou k DMA přenosům je použití kanálů. Jde o ještě inteligentnější programovatelné zařízení sloužící ke stejnému účelu. Kanály používaly např. sálové počítače IBM System/360. Na počítačích PC se kanály nepoužívají.

Shrnutí

Tato kapitola přinesla podrobnější popis základních součástí počítače, tj. procesoru, operační paměti a I/O zařízení. (Z hlubších diskuzí jsme vynechali sběrnici, kterážto je čistě hardwarovou záležitostí a studia operačních systémů se nijak netýká.)
V první části kapitoly jsme se věnovali CPU neboli procesoru. Dozvěděli jsme se, co je to instrukce a jak se vykonává, seznámili jsme se také s paralelním vykonáváním instrukcí a rozdílem mezi procesory s úplnou a redukovanou instrukční sadou. V další části kapitoly jsme se věnovali operační paměti, zejména pak způsobu uložení informací textových a číselných v ní a také obvyklými způsoby adresace paměti. Více o instrukcích procesoru a adresování paměti se studenti mohou dovědět na cvičeních a/nebo v publikaci [Kep08a], která slouží jako vodítko (učební text) pro cvičení.
Závěr kapitoly byl věnován představení obvyklých způsobů práce s I/O zařízením, zejména se zaměřením na problematiku přerušení a s ním souvisejících témat.

Pojmy k zapamatování

- procesor
- instrukce
- ALU (aritmeticko–logická jednotka)
- vykonání instrukce
- pipelining
- CISC (úplná instrukční sada)
- RISC (redukovaná instrukční sada)
- doplňkový kód
- přetečení, podtečení
- big endian, little endian
- bi–endian, middle endian
- BCD (binary coded decimal)
- adresování paměti
- segmentové adresování
- přímá a nepřímá adresa
- cache paměť
- časově–prostorová lokalita
- aktivní čekání
- přerušení
- DMA (direct memory access)
- sběrnice ISA, PCI, AGP, PCI Express

Kontrolní otázky

1. *Vysvětlete postup vykonání instrukce procesorem.*
2. *Popište, jak funguje pipelining.*
3. *Jaké jsou základní rozdíly mezi CISC a RISC procesory?*

4. *Dokažte, že změnu číselného znaménka v doplňkovém kódu provedeme převrácením hodnoty všech bitů a přičtením jedničky.*
5. *Jaký je rozdíl mezi kódováním čísel big endian a little endian? Která varianta je lepší a proč?*
6. *Popište způsob adresování paměti pomocí segmentů a offsetů.*
7. *Vysvětlete pojem časově–prostorové lokality přístupu do paměti.*
8. *Jaké jsou nevýhody aktivního čekání? Vidíte v něm i nějaké výhody? Popište je.*
9. *Jaký má při práci s I/O zařízeními význam přerušení? Byl význam přerušení v minulosti větší, či menší než dnes? Proveďte diskuzi.*
10. *Popište, k čemu slouží DMA a jak funguje.*
11. *Vysvětlete, proč LDTR je segmentový registr, ale GDTR není segmentový registr.*

3 Řízení výpočtu

Studijní cíle: V předchozí kapitole byla řeč mj. o způsobu zpracování jedné instrukce procesorem. Nyní na to navážeme a seznámíme se s globální organizací práce procesoru, tj. s pravidly pořadí vykonávání instrukcí, se skoky a podprogramy. Řeč bude také o zásobníku a přerušení, což jsou dvě velmi důležité součásti vykonávání kódu programů a řízení výpočtu.

Klíčová slova: skok, volání, podprogram, zásobník, přerušení

Potřebný čas: 90 minut.

3.1 Pořadí vykonávání instrukcí

Jak víme, instrukce jsou prováděny sekvenčně, tj. „jedna za druhou“ tak, jak jsou v paměti. Toto je samozřejmě naprosto intuitivní a dobře pochopitelné.

Procesory mohou mít instrukční sadu postavenou tak, že každá instrukce je stejně dlouhá. To umožní snadný postup programu vpřed. V praxi se však častěji používá model proměnlivé délky instrukcí – každá instrukce je pak tak dlouhá, jak je potřeba (jeden bajt, či více). Například nám známá instrukce `mov` může pracovat buď pouze s registry, nebo také s pamětí. Pracuje-li s pamětí, je pochopitelně delší, než když pracuje jen s registry. (Instrukce je obvykle delší o 4 bajty, kde je adresa paměti.) Dále, jelikož v jednom bajtu máme jen 256 hodnot, všechny instrukce se tam nevejdou, takže některé instrukce už z principu mají více než jeden bajt, přestože s pamětí ani nepracují. Procesor tedy musí u každé konkrétní instrukce vědět či nějak spočítat, o kolik bajtů dále v paměti je instrukce následující. Způsob, jakým je to zajištěno, nás naštěstí nemusí trápit.

Na procesorech x86 je v registru `eip` adresa právě vykonávané instrukce. Běžně se tedy hodnota tohoto registru postupně zvyšuje, jak program běží. Pořadí vykonávání instrukcí však můžeme několika způsoby změnit – v zásadě jsou dva typy situací, které mohou nastat:

Skok je situace, kdy instrukce velí procesoru přejít s vykonáváním na jinou adresu. Rozlišujeme *nepodmíněný skok* (instrukce `jmp`), kdy program „skočí“ na novou adresu vždy, a *podmíněný skok*, kdy program přejde na novou adresu jen při splnění určité podmínky. Které konkrétní podmínky lze testovat, to záleží na konkrétním procesoru. Např. řada x86 umí skákat dle řady hodnot v registru příznaků, nebo dle hodnoty registru ecx.

Instrukce jmp provede nepodmíněný skok.

Přerušení či volání podprogramu je situace obdobná nepodmíněnému skoku, jenže tentokrát si procesor před skokem na novou adresu nejprve na zásobník uchová aktuální adresu. Tu později použije pro *návrat z podprogramu*. Pojem „přerušení“ popisuje zvláštní případ volání podprogramu, které sice proběhne obvykle velmi podobně klasickému volání, ale je možno jej vyvolat i z vnějšího zařízení. Konkrétní možnosti přerušení jsou samozřejmě závislé na konkrétním procesoru.

Přerušení je tedy nástroj, jak program „přerušit“. Po vykonání nějakého kódu, kterému říkáme *obsluha přerušení*, program pokračuje tam, kde byl přerušen. Jelikož volání podprogramu je de facto totéž, hovoříme o něm na stejném místě. Rozdíly mezi těmito dvěma nástroji si popíšeme v následujících odstavcích. Skoky oproti tomu neumějí provést „návrat“ na původní místo – jakmile jednou někam takto skočíme, program se již zpět nevrátí.

Průvodce studiem

Samozřejmě je možné udělat návrat na původní místo i pomocí skoku – stačí na konec podprogramu dát opět instrukci skoku směřující zpět na původní místo programu. Rozdíl takového řešení oproti přerušení či volání podprogramu je, že při něm musíme předem vědět, kam se vlastně chceme vrátit, a napsat to k oné poslední instrukci skoku. Jenže

v praxi se podprogramy volají z různých míst, takže neexistuje jediné místo návratu. Proto se při volání či přerušení uloží aktuální hodnota eip na zásobník a při návratu se z něj opět vyzvedne.

Pomůcka: Instrukce skoku `jmp <adresa>` skočí na danou adresu. Adresu obvykle označíme pomocí návěští (jméno s dvojtečkou, uvedeno na začátku řádku). Instrukce volání `call <podprogram>` volá podprogram. Tato instrukce provede jakoby dvě věci:

Instrukce call volá podprogram.

```
push eip
jmp <podprogram>
```

Návrat z podprogramu provedeme jednoduchou instrukcí `ret`, která provede jakoby `pop eip`.

Instrukce ret provede návrat z podprogramu.

Průvodce studiem

Ačkoliv soudobé procesory používají pro volání podprogramů zásobník, existují i jiné modely volání. Některé procesory prostě zásobník nemají; adresu návratu pak musejí ukládat jinam, např. sálové počítače řady IBM System/360 ukládají adresu návratu do speciálního registru. Zatímco běžné řešení se zásobníkem umožňuje také rekurzi, při ukládání adresy návratu do registru není rekurze tak jednoduše možná. A jelikož rekurze je dnes jedním z klíčových prvků programování, zásobník dnes najdeme skutečně všude.

3.2 Volání podprogramu

Podprogram je obecný technický název pro procedury, funkce a metody, které najdeme ve většině programovacích jazyků. Jejich volání je jedním z nejčastějších úkolů, které program po procesoru požaduje, takže nepřekvapí, že je zajištěno hardwarově. Úroveň hardwarové podpory se samozřejmě na různých procesorech liší, my se opět budeme věnovat především řadě x86.

Od volání obvykle očekáváme:

- Předání parametrů do podprogramu
- Předání návratové hodnoty z podprogramu
- Umožnění reentrance (čili možnost rekurze)

Předávání parametrů

Procesory x86 umožňují předávat parametry několika způsoby. Nejrozšířenější jsou ty, které používají překladače nejrozšířenějších vyšších jazyků (jako je C/C++ či Turbo Pascal). Uveďme si tři nejběžnější konvence volání lišící se místem uložení parametrů, jejich pořadím a odpovědností za úklid parametrů po volání.

Předávání pomocí registrů – Stanovíme si, ve kterém registru budeme předávat který parametr. Toto řešení může být efektivní z hlediska rychlosti, protože práce s registry je vždy rychlá. Nevýhodou však může být, že registry nemůžeme používat pro běžné výpočty, pokud jsou obsazeny hodnotami parametrů. Po návratu není třeba provádět úklid.

V praxi se tento způsob předávání parametrů používá v několika variantách. Např. Visual C++ a GCC používají fastcall model, kdy první dva parametry jsou v registrech ecx a edx, ostatní pak jsou na zásobníku zprava doleva. Zásobník musí uklidit volaný.

Konvence C – Parametry předáváme přes zásobník. Před zavoláním (call) podprogramu uložíme pomocí push instrukce jednotlivé hodnoty parametrů na zásobník v opačném pořadí (tj. zprava doleva). Ukládáme pozpátku, zásobník ale roste dolů, takže zavolaná funkce má parametry pěkně za sebou zleva doprava. Funkce může libovolně používat registry eax, ecx, edx. Po návratu volající provede úklid parametrů (co uložil na zásobník, to z něj zase vybere), právě díky tomu lze tímto způsobem předávat proměnlivý počet parametrů (což používá např. funkce printf).

C ukládá parametry na zásobník zprava doleva.

Pascal – Parametry předáváme přes zásobník jako v předchozím případě, tentokrát je však ukládáme zleva doprava. Podprogram má parametry tedy pozpátku a volání s proměnlivým počtem parametrů tedy není možné (neboť by se posouvala adresa prvního parametru, což nelze). Úklid zásobníku provede volaný – jelikož počet parametrů není proměnlivý, podprogram ví, kolik parametrů má, takže je může uklidit sám. Po návratu tedy není žádný další kód potřeba.

Pascal ukládá parametry na zásobník zleva doprava.

Návratová hodnota se na x86 předává vždy v registru `eax`. Potřebujeme-li 64bitovoou hodnotu, pak použijeme dvojici registrů `edx:eax`.

Průvodce studiem

Kvalitní překladače jazyků C/C++ umožňují používat libovolné způsoby předávání parametrů. Pomocí direktiv cdecl, stdcall a fastcall si můžeme u každé funkce nastavit, jakou konvenci chceme právě pro tuto funkci. Cdecl je standardní C konvence, stdcall je způsob používaný systémem Windows ve Win32 API (tedy Pascal konvence s obráceným pořadím parametrů a volným použitím registrů eax, ecx a edx) a fastcall je výše popsaná varianta předávání přes registr.

Průvodce studiem

Potřebujeme-li vracet větší množství dat, je s tím pochopitelně problém. Visual C++ v takovém případě změní podprogram tak, že má jeden další skrytý vstupní parametr, ve kterém je adresa pro uložení výsledku. Volající tedy před voláním musí připravit dostatečně velký buffer a jeho adresu předá jako další parametr volání. Podprogram na danou adresu uloží výsledek a o víc se nestará. Volající pak svůj buffer zpracuje a uklidí jej. (Deklarativně v C++: Pro `struct A`, kde `sizeof(A)>8`, se funkce `A func();` přeloží jako `void func(A&);`.

Návratová hodnota typu double (64bitové FP číslo) se vrací na vrcholu FPU stacku.

Co se děje na zásobníku

Podívejme se nyní podrobněji, co se při volání děje na zásobníku. Neformálně řečeno, každým voláním na zásobníku vytvoříme jakýsi otisk, kterému říkáme *stack frame*, čili česky rámec funkce či rámec volání. Každé volání vytvoří další nový rámec.

Připomeňme, že zásobník roste dolů. Seshora jsou v něm tedy uloženy hodnoty tak, v jakém pořadí je tam ukládáme. Čili hodnoty jsou na zásobníku v tomto pořadí:

- Parametry volání (zprava doleva, či zleva doprava – dle konvence)
- Návratová adresa
- Lokální proměnné
- Další řídicí informace (např. odkaz na předchozí volání)

Každé volání stejného podprogramu vytvoří stejně velký otisk, různé podprogramy však pochopitelně potřebují různé množství paměti (neboť mají různý počet parametrů, různý počet lokálních proměnných apod.).

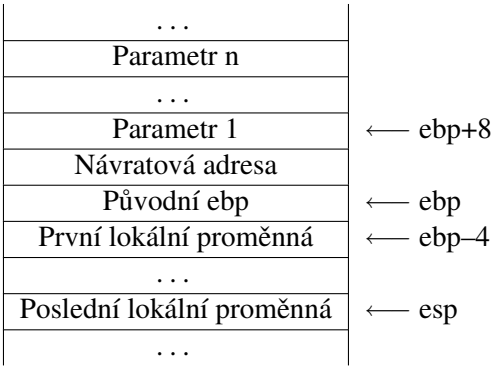
Při volání se používají dva důležité registry pro práci se zásobníkem. Registr `esp` vždy ukazuje na vrchol zásobníku. Každá instrukce `push` či `call` sníží `esp` o 4, každá instrukce `pop` či `ret` zvýší `esp` o 4. Nelze ukládat jinak než 4 bajty (čímž se některé věci komplikují, ale na zásobníku aspoň máme stále pořádek.)

Druhým použitým registrem je `ebp`. Pro pochopení jeho významu si představme následující situaci: Běží nějaká funkce s lokálními proměnnými a tato chce zavolat jinou funkci. Nejprve tedy musí na zásobník uložit všechny parametry pro volání. Jenže každé uložení jednoho parametru změní vrchol zásobníku, takže funkce se registrem `esp` nemůže dopídit ke svým lokálním proměnným. Ty jsou totiž na pevném místě v zásobníku přesně tam, kde byl jeho vrchol v okamžiku zavolání funkce. Jelikož se však na zásobník začalo připravovat další volání, předchozí pozici jsme ztratili. A co se s tím dá dělat? Do registru `ebp` si vždy na začátku podprogramu uložíme aktuální hodnotu `esp`. Naše lokální proměnné pak vždy najdeme přes `ebp`, i když se vrchol zásobníku bude měnit. Aby vše spolehlivě fungovalo, na začátku podprogramu samozřejmě musíme nejprve uložit dosavadní hodnotu ebp, abychom nezrušili odkaz na lokální proměnné nadřazené (tj. volající) funkce. Místo, kam ukazuje `ebp`, označujeme jako *dno zásobníku*.

Ebp ukazuje na dno zásobníku.

Nyní si proces volání podprogramu popišme v instrukcích procesoru (viz také obrázek 3):

1. Nejprve uložíme parametry. Dle konvence C ukládáme parametry zprava doleva. (push <data>)
2. Voláme podprogram , tím se na zásobník uloží adresa návratu. (call <adresa>)
3. Program dále pokračuje již v podprogramu. První instrukcí každého podprogramu musí být uložení registru ebp, kde je odkaz na rámec nadřazeného podprogramu. (push ebp)
4. Nyní do ebp uložíme aktuální esp, čímž si uložíme odkaz na pozici našeho rámce. (mov ebp,esp)
5. Vytvoříme potřebný prostor pro lokální proměnné. (sub esp,<velikost>)
6. Všechny registry kromě eax, které budeme měnit, uložíme na zásobník. (push)



Obrázek 3: Data na zásobníku v okamžiku volání podprogramu

Nyní může běžet vlastní kód podprogramu. Vstupní parametry jsou přístupné na adresách ebp+x, lokální proměnné pak na adresách ebp-x. První parametr je na adrese ebp+8, další ebp+12 atd. První lokální proměnná pak na adrese ebp-4, další ebp-8 atd. (Na adrese ebp je navíc odkaz na rámec nadřazené funkce, na adrese ebp+4 je návratová adresa. Tyto dvě hodnoty samozřejmě nesmíme měnit.)

Průvodce studiem

Programujete-li ve Visual C++, vypněte si v Project Properties/Code Generation položku Basic Runtime Checks. Je-li totiž zapnutá, překladač ukládá na zásobník řadu dalšího smetí, čímž se snaží kontrolovat chyby při používání paměti. Ačkoliv tyto kontroly jsou vynikajícím pomocníkem, znemožňují nám studium Assembleru. Proto je lepší je vypnout.

Řádné ukončení podprogramu provedeme pomocí opačné sekvence kroků:

1. Nejprve obnovíme hodnoty všech změněných registrů. (pop)
2. Zrušíme prostor pro lokální proměnné. Využijeme přitom faktu, že původní esp máme uložené v ebp. (mov esp,ebp)
3. Obnovíme ukazatel na rámec nadřazeného volání. (pop ebp)
4. Provedeme návrat. (ret)

Nyní je řízení předáno zpět volajícímu. Ten musí nejprve uklidit ze zásobníku všechny předané parametry volání. (add esp,<počet*4>)

Průvodce studiem

Zatímco pořadí parametrů je při volání velmi důležité, u lokálních proměnných tomu tak není. Když si prostor pro proměnné vytváříme sami, tak pochopitelně pouze my víme, která jeho část je pro kterou proměnnou. Systém nejlépe sami pochopíte, když si jej vyzkoušíte v praxi.

Další hardwarová podpora volání

Procesory x86 mají počínaje modelem 80286 podporu pro vytváření a úklid rámců na zásobníku v podobě dvou doplňkových instrukcí. Dnes se tyto instrukce příliš nepoužívají, uplatnění však najdou v jazycích s dynamickým rozsahem platnosti proměnných, jako je Turbo Pascal.

Instrukce `enter` vykoná úvodní sekvenci podprogramu, instrukce `leave` vykoná závěrečnou sekvenci. Tyto instrukce jsou však na současných procesorech při běhu pomalejší, než když vše napíšete ručně pomocí základních instrukcí, jak bylo popsáno výše. Proto je jejich využití (mimo zmíněný Turbo Pascal) prakticky nulové.

3.3 Přerušení

Jak již víme, přerušení je jedním z velmi důležitých prvků počítačů. Je to věc, která se týká hardwaru i softwaru. Původně přerušení sloužilo zejména k reagování na asynchronní události[2]. Nejčastějším případem asynchronní události je, když I/O zařízení chce komunikovat s procesorem. V okamžiku, když zařízení chce procesoru něco sdělit, vyvolá přerušení. Činnost procesoru je v tomto okamžiku přerušena a tento přejde do speciálního podprogramu zvaného *obsluha přerušení*. To, jakým způsobem procesor ví, kde má obsluhu přerušení, a jak od sebe rozliší jednotlivá přerušení, se na jednotlivých počítačích může dosti lišit. Přerušení však nastává vždy po vykonání celé instrukce; přerušit program uprostřed nějaké instrukce nelze. (Výjimkou je neošetřitelná chyba, kdy je vykonání instrukce přerušeno a zrušeno.)

Většina procesorů navíc také obsahuje instrukce, kterými je možno přerušení vyvolat programově. Toto je pak velmi podobné obyčejnému volání podprogramu. Rozdíl je v tom, že při

[2] Asynchronní událost je taková, se kterou se předem nepočítá. Nastane neplánovaně.

volání podprogramu musíme vždy uvést adresu tohoto podprogramu, zatímco při vyvolání přerušení jen uvedeme kód přerušení (číslo) a procesor vykoná podprogram obsluhy přerušení. V dalších odstavcích se opět podíváme, jak přerušení funguje na procesorech x86.

Průvodce studiem

Příkladem I/O zařízení může být klávesnice. Stisk klávesy je asynchronní událost, o které se procesor dozví díky přerušení, které klávesnice v okamžiku stisku vyvolá. Procesor tak přejde na obsluhu klávesnice, což je velmi krátký podprogram. Po načtení stisknuté klávesy z klávesnice skončí obsluha přerušení a opět pokračuje původní program.

Průvodce studiem

Přerušení se v počítačích používají ve velké míře. Dříve tomu tak však nebylo. Např. právě obsluha klávesnice přerušením je možná pouze tehdy, když klávesnice má vlastní inteligenci. Pokud by ji neměla, musel by procesor v pravidelných intervalech sám kontrolovat mačkání kláves.

Extrémním příkladem, jak se v historii o vše staral sám procesor, je kreslení obrazu. Zatímco dnes toto zajišťuje grafická karta, v dávnější minulosti to zajišťoval procesor. Většinu výpočetního času tak samozřejmě strávil právě vykreslováním obrazu, ale takový počítač byl samozřejmě mnohem levnější, než kdyby musel obsahovat vlastní čip pro kreslení obrazu. Např. Sinclair ZX81 (viz obrázek 4) z roku 1981 běžel jen na čtvrtině výkonu, zbytek byl použit pro kreslení. Obraz však šel programově vypnout (čímž se počítač 4× zrychlil).

Obrázek 4: Počítač Sinclair ZX81

Přerušení jsou obsluhována ve spolupráci hardwaru (procesoru) a softwaru (operačního systému). V okamžiku požadavku přerušení se nejprve dokončí aktuální instrukce. Pak procesor sám uloží aktuální stav vykonávání programu (čili především návratovou adresu) na zásobník a přejde na obsluhu přerušení. Operační systém se stará o tabulku adres obslužných podprogramů. Procesor x86 má registr IDTR obsahující deskriptor tabulky IDT. IDTR tedy ukazuje na IDT,

což je seznam deskriptorů jednotlivých obslužných podprogramů. Tyto podprogramy mohou být i v jiném procesu, než právě běží. V tom případě procesor uloží svůj kompletní stav a přepne se do kontextu onoho procesu, který má zajistit obsluhu přerušení. Zde tedy jde o poměrně složitou hardwarově podpořenou operaci, podrobnosti odložíme na později. Samotný obslužný podprogram pochopitelně klasicky uloží všechny registry, které bude měnit, a chová se téměř stejně jako každý jiný podprogram. Na konci má pak speciální instrukci `iret`, kterou aktivuje proces návratu z přerušení. Je-li adresa návratu v jiném procesu, procesor obnoví dříve uložený stav a přejde do původního kontextu. Pak pokračuje ve vykonávání původního programu.

Během obsluhy přerušení je další přerušení zakázáno. Slouží k tomu speciální příznak (bit v registru eflags). Je však zvykem, že je-li to možné, obslužný podprogram přerušení opět povolí. Může pak nastat situace, že procesor je přerušen z přerušení. Na konci obsluhy přerušení je příznak přerušení každopádně nastaven do původního stavu, v jakém byl před přerušením.

Výše popsaný systém s registrem IDTR používá na počítačích PC např. systém Windows. Naproti tomu MS–DOS používal jednodušší systém, daný použitým procesorem Intel 8088: Prvních 1024 bajtů fyzické paměti je vyhrazeno pro 256 adres obsluh přerušení. Každé přerušení má tedy 4bajtovou adresu, čili klasicky segment + offset. Při přerušení jsou na zásobník uloženy jen registry CS, IP a FLAGS, dál nic.

Systém použitý v MS–DOSu umožňuje, aby libovolný program přepsal adresu obsluhy přerušení z původního podprogramu na sebe. Toho využívaly viry – „pověsily“ se na obsluhy důležitých přerušení a mohly tak ovlivnit, co se stane před či po zavolání skutečné obsluhy. Tento ráj virů byl samozřejmě na obtíž, proto se později od tohoto způsobu obsluhy přerušení ustoupilo. (Vyžádalo si to nový mikroprocesor. Samotný operační systém bez hardwarové podpory dokonalou bezpečnost zaručit nemůže. MS–DOS přitom zůstal u původního systému až do poslední verze, dal tedy před vyšší bezpečností přednost větší zpětné kompatibilitě.)

IDTR a IDT jsou operačním systémem chráněny proti změnám, takže ve Windows již viry prakticky neexistují. (Ačkoliv existuje celá řada jiných škodlivých programů, kterým laici obvykle říkají „viry“, skutečné viry v původním slova smyslu ve Windows opravdu téměř nejsou.)

Hardwarově je přerušení na počítačích PC ještě podpořeno systémem priorit. Přerušení vyšší priority může přerušit běžící obsluhu přerušení nižší priority, ale ne naopak. Objeví-li se požadavek přerušení v okamžiku, kdy je přerušení zakázáno (nastavením příznaku přerušení), počítač si jej pamatuje a vyvolá jej později, jakmile bude přerušení povoleno. Tato „paměť na přerušení“ je samozřejmě limitována, ale obvykle stačí.

Shrnutí

Tato kapitola se věnovala řízení výpočtu, což je téma volně navazující na vykonávání jednotlivých instrukcí probrané v kapitole předchozí. Dozvěděli jsme se, že základem je vykonávání instrukcí v tom pořadí, v jakém jsou zapsané za sebou v paměti. Dále jsme se věnovali skokům, přerušení a podprogramům, kterými lze výpočet větvit či strukturovat. V této souvislosti jsme zcela vynechali problematiku podmíněného vykonávání kódu, které bude věnován čas zejména na cvičeních (předmětu Operační systémy) a také je probrána v publikaci [Kep08a]. Naopak jsme se ale podrobně věnovali zásobníku a seznámili jsme se s tím, jak je tento konstrukt využíván při přerušení, volání podprogramů, předávání parametrů apod. Pochopení, jak funguje zásobník a co vše díky němu počítač dokáže, také patří k základním cílům úvodních kapitol tohoto kurzu.
Tato kapitola také uzavírá úvodní blok, který se věnuje spíše samotnému počítači, který jsme si představili jako komplexní systém (stroj) skládající se z hardwaru a softwaru. Následující kapitoly se pak budou věnovat méně hardwarovému a naopak více softwarovému pohledu na počítač, protože operační systémy, které nás zajímají především, patří mezi software.

Pojmy k zapamatování

- skok – podmíněný a nepodmíněný
- volání podprogramu
- konvence předávání parametrů
- obsluha přerušení
- IDT (interrupt descriptor table) a IDTR
- vrchol a dno zásobníku
- stack frame (rámec volání na zásobníku)

Kontrolní otázky

1. *Vyjmenujte základní druhy instrukcí vedoucích ke změně pořadí vykonávání instrukcí.*
2. *Vysvětlete rozdíl mezi skokem, voláním podprogramu a přerušením.*
3. *Který způsob předávání parametrů do podprogramů je nejvýhodnější? Proveďte diskuzi.*
4. *Popište podrobně, co se děje na zásobníku při volání funkce od jeho přípravy až po návrat.*
5. *Kdy přesně vzniká a zaniká rámec volání funkce na zásobníku? Proveďte diskuzi.*
6. *Jak byste implementovali přístup k lokálním proměnným a parametrům funkce, kdyby kromě vrcholu zásobníku (esp) nebyl k dispozici druhý registr (ebp)? Diskutujte výhody a nevýhody takového řešení.*
7. *Kdy se procesor může nebo nemůže přerušit? Uveďte všechna pravidla.*
8. *Systém MS-DOS byl náchylný na viry, protože do systémové tabulky obsluh přerušení mohl kdokoliv zapisovat cokoliv. Diskutujte možné způsoby řešení této situace (a) s pomocí hardwaru a (b) bez pomoci hardwaru.*
9. *Jak už víme, položky IDT mohou odkazovat také do cizích procesů. Uveďte možná využití takové vzdálené vazby.*

4 Operační systém

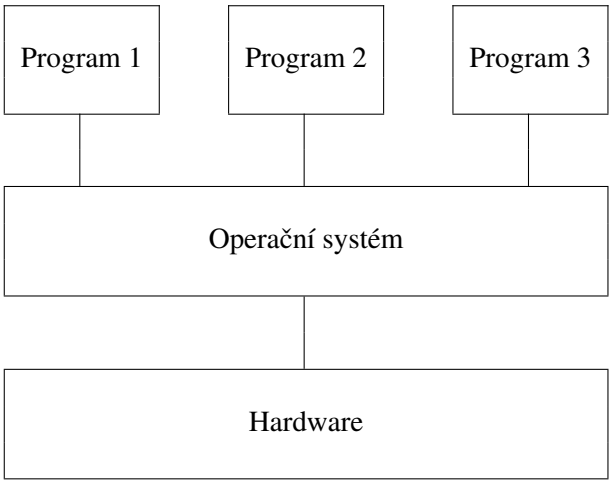
Studijní cíle: Po úvodních třech kapitolách, ve kterých jsme se seznámili s počítačem a jeho fungováním, se nyní dostáváme k samotnému operačnímu systému. Tato kapitola je tedy jakýmsi druhým úvodem v tomto kurzu a studenti budou seznámeni s tím, co to vlastně je operační systém, proč vlastně existují operační systémy a jaké jsou jejich základní funkce.

Klíčová slova: operační systém, rozhraní systému, sálový počítač, osobní počítač

Potřebný čas: 80 minut.

4.1 Co je to operační systém

V předchozích kapitolách jsme se dívali na počítač z pohledu hardwaru a rozdělili jsme ho na 3 základní části: CPU, operační paměť a I/O zařízení. Když se však na stejný stroj (počítač) podíváme z pohledu softwarového, vidíme jeho strukturu zcela jinak: hardware, operační systém a aplikační programy. Jak můžete vidět na obrázku 5, operační systém je jakýmsi prostředníkem mezi hardwarem a aplikačními programy.



Obrázek 5: Softwarová struktura počítače

Jak už víme, jedním z důvodů vzniku a existence operačního systému (dále jen OS) je velká složitost hardwaru počítače. OS je základním programem v počítači, který nabízí vyšší úroveň abstrakce při práci s hardwarem, čímž velkou měrou zjednodušuje ostatní programy. Důležitým poznatkem tedy je, a vidíme to i na obrázku 5, že OS má dvě odlišná rozhraní: jedním komunikuje s hardwarem, druhým komunikuje s ostatními programy, které obvykle dělíme na systémové – ty pomáhají operačnímu systému – a aplikační – ty slouží přímo uživatelům (lidem).

OS má dvě odlišná rozhraní.

Další pohled na význam OS je zřejmý každému, kdo někdy pracoval s PC: Operační systém umožňuje, aby v počítači současně běželo více programů než jeden. Obecně lze říci, že hardwarová podpora pro souběh programů je či může být jen velmi malá, zatímco právě OS k umožnění souběhu přispívá velmi velkou měrou. Z hlediska souběhu aplikací musí OS především provádět pečlivou a spravedlivou *správu zdrojů* – dle typu jednotlivých součástí počítače jsou tyto díky operačnímu systému buď rozdělovány mezi aplikace (např. operační paměť), sdíleny v pravidelných časových úsecích (např. CPU), přidělovány do dočasného výhradního vlastnictví (např. klávesnice) či virtualizovány pro umožnění transparentního sdílení funkcionality (např. obrazovka, tiskárna, disk, reproduktory).

OS zajišťuje správu zdrojů.

Fakt, že OS má kromě vnějšího rozhraní, kterým nabízí svou funkcionalitu ostatním programům, také druhé vnitřní rozhraní, umožňuje, aby stejný operační systém a potažmo stejné aplikace byly provozovány na různých počítačích. V současnosti jsou největší rozdíly zřejmě na poli

grafických karet – zatímco hardwarová řešení jednotlivých výrobců mají spolu jen velmi málo společného, uživatel těžko uvidí nějaký rozdíl v chování počítače. (Naopak nejmenší pokrok v této oblasti je již tradičně na poli CPU – provozování stejných programů na jiném procesoru není obvykle možné.)

Ačkoliv je vidět, že OS je důležitou součástí počítače, přesná definice, co je to operační systém, neexistuje. Každý počítač má jiný účel, proto také jednotlivé operační systémy se liší. OS je jednou ze součástí počítače, která se podílí na tom, aby tento účel byl naplněn. OS může mít mnoho podob. Literatura také někdy cynicky definuje operační systém jako „to, co dostanete, když si objednáte „operační systém““ (viz [SGG05]).

Jiná, pragmatičtější definice říká, že operační systém je „to, co běží při každém provozu počítače“. Tato definice zohledňuje právě i rozdíly mezi různými počítači. Na malém zabudovaném počítači jsou logicky i funkce operačního systému omezeny – není zde systém oken, ani podpora práce s hard diskem. Naopak na PC si dnes práci bez oken či hard disku těžko dovedeme představit, jsou to tedy dvě ze základních součástí operačního systému.

Průvodce studiem

Definice operačního systému v posledních letech také velmi zajímá antimonopolní úřady v mnoha zemích světa. Důvodem je Microsoft a jeho systém Windows, který dle názorů konkurenčních firem má už funkcí příliš mnoho. Uživatelům takto bohatě vybavený systém samozřejmě vyhovuje, úřady v některých zemích však už několikrát nařídily, aby Microsoft některé části ze svého operačního systému odstranil. Z dlouhodobého hlediska je přístup úřadů jistě správný – zatímco dnes uživatelé přijdou o část softwaru, který „byl v ceně“, v budoucnu lze díky hospodářské soutěži očekávat bohatší nabídku s mnoha alternativami různých výrobců.

Cíle operačního systému

Jednodušší než odpovědět na otázku: Co to je operační systém?, může být odpovědět na otázku: Co operační systém dělá? Co jsou jeho cíle?

Primární cíle OS: Počítač má být příjemný a pracovat efektivně.

Operační systém umožňuje uživatelům snazší práci s počítačem. Na příkladu výše zmíněného systému oken je toto velmi zřetelné a je to možná i základní cíl operačního systému na PC. Na jiných počítačích je zase primárním cílem efektivní využití hardwaru, který je k dispozici. Toto je praxí u počítačů hodně drahých (např. velká výpočetní centra) či drahých vzhledem k jejich využití (např. nemá smysl dávat PC do automatické pračky, když by tím řídicí počítač byl dražší než celý zbytek pračky). Zajímavé je, že tyto dva alternativní cíle jsou často oba přínosné, ale jsou také ve sporu – zvyšováním uživatelské přívětivosti totiž snižujeme efektivitu využití hardwaru a obráceně. V historii byly počítače přece jen dosti pomalé, proto velmi výrazně převažoval právě požadavek jejich efektivního využití. Toto také mnoho let determinovalo vývoj v oblasti operačních systémů i studia jejich teorie.

Průvodce studiem

Závěr předchozího odstavce diplomaticky říká, že studium operačních systémů je z historických důvodů zaměřeno na témata týkající se efektivního využití zdrojů v počítači a naopak prakticky ignorována je otázka grafického uživatelského rozhraní, které je dnes velmi aktuálním tématem u operačních systémů z hlediska uživatelů.

Autor této učebnice zastává názor, že otázky grafického uživatelského rozhraní určitě patří do studia operačních systémů a že tam chybí pouze z důvodu historických – autoři učebnic i učitelé se v této oblasti jako dinosauři stále drží témat zavedených již v 70.letech 20.století. Ne nadarmo jsou dinosauři také ústředním grafickým motivem knihy [SGG05].

Díky tomu, že OS vystupuje jako správce všech hardwarových zdrojů (součástí počítače), snaží se také zajistit určitou bezpečnost. Např. když si jeden program založí soubor na disku, ostatní programy do tohoto souboru nic zapisovat nesmí. Tento typ bezpečnosti stojí na tom, že OS odstiňuje jednotlivé programy a tyto o sobě navzájem pokud možno ani neví. Pro maximální bezpečnost je samozřejmě nutná jistá podpora hardwaru, ale základní úroveň odstínění aplikací dokáže zajistit již sám operační systém. Např. tisk na tiskárně byl již na starších verzích Windows (98 a dříve) rozumně ošetřen a nebyly s ním žádné potíže, přestože technicky tehdy aplikacím nic nebránilo, aby tisk ostatních aplikací porušily.

Aplikace na sebe navzájem nevidí.

4.2 Typy systémů

O tom, že existují různé typy počítačových systémů, jsme se již zmínili v kapitole 1. Víme také, že tato rozmanitost je výsledkem jejich historického vývoje a také obrazem toho, že každý počítač je primárně určen k jinému účelu. Nyní se k tomuto tématu vrátíme a uvedeme si základní typy počítačových systémů ze softwarového hlediska.

Průvodce studiem

Jak lze vypozorovat z popisu v následujících odstavcích, mnoho vlastností původně specifických jen pro velké sálové počítače se postupně dostávalo a dostává do počítačů menších a menších. Tento trend je všeobecně platný – postupné zmenšování hardwaru umožňuje nasazovat počítače do stále více rozměrově či energeticky náročnějších prostředí. Nejprve se tam však vždy objeví počítače jednoduché, jakoby vzaté z historie, a později se jejich „dokonalost“ zvyšuje a jejich možnosti se sbližují s počítači většími.

4.2.1 Sálové počítače

Jak víme, prvními prakticky užitečnými počítači byly sálové počítače z doby kolem konce 2. světové války.

První počítače zpracovávaly *dávkové úlohy*. Během výpočtu nebyla interakce s uživatelem možná – čili programátor dodal program (úlohu), operátor jej dal do počítače (např. ve formě děrné pásky) a po zahájení výpočtu se čekalo, dokud počítač nedodal výsledek. Ten pak operátor odevzdal programátorovi. Počítač pak zahálel, dokud operátor nepřipravil k výpočtu další program (úlohu). Tento způsob využití byl značně neefektivní, právě proto se jednotlivé úlohy spojovaly do dávek. Počítač pak dostával úlohy po dávkách – po vykonání jedné úlohy mohl okamžitě pokračovat na další úloze a nezahálel čekáním na operátora.

Dávkové zpracování však především umožnilo, aby počítač efektivně využíval své zdroje. Základní model zpracování má totiž zásadní výkonnostní hrdlo v tom, že zařízení, ze kterého čte data (dříve děrné štítky, později magnetické pásky, dnes magnetické či optické disky), je podstatně pomalejší než samotný procesor, takže při sekvenčním (tj. postupném) zpracovávání úloh počítač neustále zahálí čekáním na načtení vstupních dat. Při zavedení dávkového zpracování může počítač (s pomocí vhodného operačního systému) zpracovávat více úloh současně.

V jednom okamžiku může jeden počítač samozřejmě zpracovávat jen jednu úlohu. Jakmile však nastane situace, kdy procesor musí čekat na externí zařízení (nejčastěji načtení vstupních či zápis výstupních dat), namísto zahálky přejde k vykonávání další úlohy. Jakmile by tato na něco měla čekat, procesor opět přejde k další úloze, atd. V okamžiku, kdy některá z pozastavených úloh může pokračovat, procesor se k ní vrátí a bude s ní pokračovat. Pokud tedy operátor dodává dostatečný počet úloh, počítač je efektivně využit a nikdy nezahálí. Tento model zpracování úloh se nazývá *multiprogramování*.

Multiprogramování – v počítači je současně více programů.

Průvodce studiem

Multiprogramování je běžné i u lidí. Např. právník také nemá jen jednoho klienta – spoustu času by totiž jen seděl a čekal na vyřízení nějakých žádostí o dokumenty, o které žádal a které mají přijít poštou, atp. Tím, že má právník více klientů a dle svých časových možností se věnuje každému z nich „chvilku“, dokáže svůj pracovní čas efektivně využít.

Multiprogramování klade poměrně velké nároky na operační systém. Má-li být počítač efektivně využit zpracováváním více úloh současně, těchto úloh by mělo být tolik, aby počítač nikdy nezahálel. Pak ale musí operační systém rozhodovat, kterou z řady čekajících úloh předá procesoru k vyřešení přednostně. Úlohy také mají své nároky na operační paměť – pokud je jich mnoho, do paměti se všechny nevejdou; i řízení využití paměti má na starost operační systém. Tyto úkoly patří k nejdůležitějším, které operační systém plní, a budeme se jim podrobněji věnovat v dalších kapitolách.

Průvodce studiem

Aby počítač mohl zpracovávat více úloh současně, musí mít minimálně tyto dvě hardwarové funkce: 1. oddělená činnost CPU a periferií, 2. přerušení. Oddělení činnosti periferií od CPU dosáhneme tím, že jim dáme vlastní inteligenci. Počítač tak místo zbytečné zahálky při čekání na periferie může vykonávat kód jiné úlohy. Periferie se později díky přerušení může sama ozvat a vyžádat si opětovnou pozornost CPU.

Dalším vývojovým stupněm multiprogramování jsou systémy sdílející čas – *time sharing systems*. Původní princip dávkového zpracování byl nahrazen možností interaktivní práce člověka s počítačem. Sálové počítače byly původně koncipovány tak, že s jedním počítačem pracovalo víc lidí. Lze dokonce říci, že přímo mnoho lidí současně – hardware byl velmi drahý a díky sdílení času bylo možno výpočetní výkon počítače efektivně využít.

Systémy sdílející čas mají řadu výhod a používají se dnes na velkých serverech i na malých osobních počítačích. Tyto počítačové systémy samozřejmě pro svůj chod potřebují ještě složitější operační systémy, než měly dávkově pracující stroje. Moderním trendem je také nasazování více CPU jednotek do jednoho počítače. Tyto *víceprocesorové počítače* jsou opět cenově výhodné, neboť umožňují zvýšit výpočetní výkon při zachování složitosti samotných procesorů a ceny periferií, které jsou sdílené (takže máme jen jedno I/O zařízení, ale více procesorů jej používá společně).

4.2.2 Osobní počítače

Samotné osobní počítače (PC – personal computer) se objevily již v 70.letech 20.století. Jejich tehdejší procesory postrádaly možnost chránit operační systém před chybami (či útoky) aplikací, což značně omezovalo jejich využitelnost pro více uživatelů či aplikací současně (a zároveň to umožnilo zaplavení počítačů viry). Tato situace se změnila v polovině 80.let (procesor 80386), ale teprve začátkem 90.let se začaly masově rozšiřovat novější operační systémy na úkor dřívějšího MS–DOSu. V současnosti používané systémy (Windows NT, Linux, MacOS X) hodně těží z principů, které již před mnoha lety byly použity u sálových počítačů.

4.2.3 Víceprocesorové počítače

Důležitým hlediskem je také počet CPU v počítači. Poskytuje-li operační systém sdílení času pro umožnění běhu více aplikací současně, tak se sama nabízí možnost využít více CPU

pro rychlejší běh celého systému. Víceprocesorové počítače, které bývají také označovány jako *těsně vázané systémy*, obsahující více CPU jednotek, které spolu sdílejí zbytek počítače (sběrnici, paměť a periferie), jsou samozřejmě variantou cenově velmi výhodnou (levnější než několik kompletních počítačů). Vyžadují však ještě složitější operační systémy a také složitější hardwarovou architekturu, která soužití více CPU v jednom počítači umožní.

Dnes nejběžnější variantou víceprocesorových systémů jsou symetrické multiprocesory (SMP), kde jsou si všechny CPU rovny. (Příkladem je systém Windows NT.) Programování aplikací pro SMP systému je poměrně jednoduché a rovnost mezi CPU jednotkami umožňuje efektivní využití všech zdrojů. Další alternativou jsou pak asymetrické systémy, kde každá CPU jednotka má specifickou úlohu. Symetričnost systému může být daná jak hardwarem, tak stavbou operačního systému. Např. systém Solaris existuje v symetrické i (starší) asymetrické verzi pro stejný hardware. [SGG05]

Průvodce studiem

Jak víme, v moderním počítači je mnoho mikroprocesorů. Pokud pouze jeden z nich má úlohu CPU a ostatní slouží jako pomocné obvody inteligentních periferií, nepovažujeme takový počítač za víceprocesorový. Toto je sice čistě technicky zcela nesprávné, ale odráží to fakt, že pomocné procesory jsou již dnes zcela běžné a jsou používány doslova všude. Teprve počítače obsahující více procesorů sloužících jako CPU a umožňujících zvýšit výkon počítače jejich spoluprací na výpočtech považujeme za víceprocesorové.

Extrémním příkladem toho, co se za víceprocesorový systém nepovažuje, jsou moderní grafické karty. Jejich procesory jsou velmi výkonné výpočetní jednotky, zcela běžně v matematických operacích i výkonnější než samotné CPU. Přesto přítomnost takového čipu v počítači není důvodem k označení „víceprocesorový“.

4.2.4 Distribuované systémy a klastry

Dalším typem počítačových systémů jsou *distribuované systémy*. Jde de facto o více samostatně fungujících počítačů propojených do sítě. Toto propojení může mít samozřejmě mnoho podob a s pomocí speciálního distribuovaného operačního systému mohou všechny takto propojené počítače fungovat jako jeden celek. Těmto systémům také říkáme *volně vázané systémy*.

Na rozdíl od víceprocesorových systémů, kde několik CPU obvykle sdílí stejnou operační paměť, distribuované systémy obecně nemají tuto vlastnost a jednotlivé části systému – výpočetní uzly – spolu pouze komunikují pomocí zasílání zpráv. Distribuovaný operační systém zajišťuje tuto komunikaci a může do určité míry i simulovat sdílenou paměť a zakrýt tak fyzické rozdíly mezi volně a těsně vázaným systémem.

Distribuované systémy také často rozdělujeme dle vnitřní organizace uzlů na systémy typu klient–server, kde jeden uzel má výsadní postavení a řídí celý výpočet, a systémy typu peer–to–peer, kde jsou si všechny uzly rovny. Módním pojmem je také *klastr* (anglicky cluster), označující distribuovaný systém, který vznikl pouhým propojením klasických počítačů v počítačové síti (a dodáním specifického softwaru). Všechny tyto typy distribuovaných systémů jsou v posledních letech velmi aktuální, což souvisí zejména s nízkými náklady na jejich provoz (ve srovnání s velkými superpočítači) a rozvojem internetu. Jelikož se však jedná o téma dotýkající se spíše počítačových sítí, nebudeme se mu dále nijak podrobněji věnovat.

4.2.5 Další typy počítačových systémů

Existuje samozřejmě ještě celá řada dalších počítačových systémů. Např. [SGG05] uvádí v přehledu ještě *systémy pracující v reálném* čase (nebo stručně „systémy reálného času“, anglicky

real–time systems) a *kapesní počítače* (PDA). V prvním případě jde o počítače a operační systémy zaručující vyřízení určitých požadavků v pevně daném maximálním čase, příkladem může být např. systém elektronického vstřikování paliva v automobilech. Tyto systémy mohou být implementovány hardwarově i softwarově a jsou samozřejmě zaměřeny vždy na úzkou aplikační oblast. Např. Windows NT systémem reálného času není (i když počítačové hry či přehrávání videozáznamů jisté požadavky na vyřizování událostí v pevném čase mají). Ve druhém případě (PDA) jde o malé přenosné počítače, které de facto kopírují běžné osobní počítače, ovšem jsou velmi limitovány energeticky, paměťově, prostorově atp.

Samozřejmě lze očekávat, že během následujících let se současné technické limity podaří překonat a programování PDA bude více a více podobné (dnešnímu) programování osobních počítačů. Podobně je tomu i se systémy pracujícími v reálném čase. V posledních letech je trendem používání běžných univerzálních operačních systémů (např. Linux či Windows) ve spotřební elektronice (např. satelitní přijímače, DVD přehrávače či rekordéry apod.). Tyto aplikace obvykle používají nějaký specializovaný hardwarový obvod (např. dekodér MPEG videa) doplněný o počítač s upravenou verzí některého běžného operačního systému. Tento celek pak sice nemusí být nutně systémem reálného času, ale svým chováním se mu přinejmenším velmi blíží.

Další specializované systémy spadající do disciplíny operačních systémů jsou systémy multimediální, které se týkají studia multimédií (založeno na zvuku a obrazu). Multimédia pak úzce souvisejí se systémy reálného času a distribuovanými systémy. Multimediální operační systémy jsou tedy jakýmsi mezioborovým průsečíkem několika disciplín a mají úzké napojení na praktické aplikace v oblasti počítačů, domácí elektroniky apod.

Shrnutí

V této kapitole jsme se seznámili s pojmem operační systém. Pokusili jsme se jej definovat jak přímo, tak nepřímo skrz jeho cíle. Dozvěděli jsme se, že hlavním posláním operačního systému je abstrahovat hardwarové zdroje a zjednodušit práci s nimi a umožnit současné zpracování více úloh. Dále jsme se v této kapitole seznámili s několika základními typy systémů a ukázali jsme si, jak se počítače a jejich operační systémy během let vyvíjely. Z tohoto přehledu bylo také patrné, že i nejsložitější funkce operačních systémů se postupně dostávají z velkých (sálových) superpočítačů do menších a menších počítačů a rozdíly mezi nimi se tak postupně zmenšují. Výjimkou jsou samozřejmě počítačové systémy zaměřené na nějakou speciální aplikační oblast; ty mohou mít specifické vlastnosti bez ohledu na jejich velikost či složitost.

Pojmy k zapamatování

- cíle operačního systému
- sálový počítač
- osobní počítač
- víceprocesorový počítač
- distribuovaný systém
- těsně/volně vázaný systém
- počítačový klastr

Kontrolní otázky

1. *Jmenujte základní účel(y) operačního systému.*
2. *Jaký přínos má to, že operační systémy mají (minimálně) dvě různá rozhraní?*
3. *Popište hlavní rysy počítačů, kterým říkáme „sálové“.*
4. *Co je hlavní výhodou multiprogramování?*
5. *Jaký je rozdíl mezi těsně a volně vázanými systémy? Jmenujte výhody a nevýhody obou řešení.*

6. *Lze zajistit bezpečnost systému bez podpory hardwaru (tj. jen napsáním kvalitního operačního systému)? Zdůvodněte.*
7. *Popište rozdíl mezi symetrickým a asymetrickým multiprocesorem.*
8. *Jaké jsou hlavní rozdíly mezi operačním systémem pro sálové a osobní počítače?*

5 Správa procesoru

Studijní cíle: Centrem všeho dění v počítači, nebo alespoň dle von Neumannova schématu, je procesor (CPU). Tato kapitola se věnuje jeho správě. Studenti budou seznámeni s pojmy proces a vlákno, které zde budou vysvětleny především teoreticky a v kapitole následující pak i na příkladu dvou operačních systémů: Unix a Windows NT.

Klíčová slova: proces, vlákno, plánovač úloh, PCB

Potřebný čas: 100 minut.

5.1 Procesy

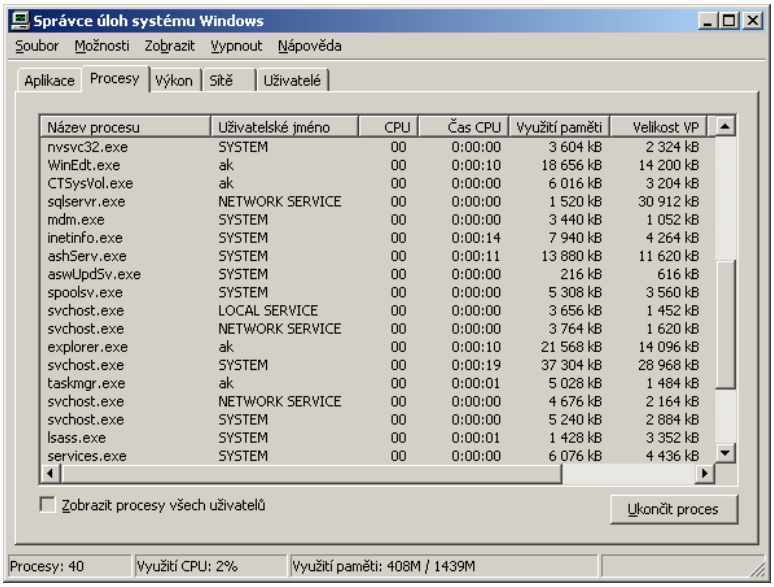
Činnosti, které provádí procesor, můžeme nazývat různými způsoby. V systémech, které jsme si představili v předchozí kapitole, bývají nazývány např. job či task. Obecný termín pro označení činnosti procesoru, bez ohledu na konkrétní situaci, je pak *proces*. Neformální definice procesu je velmi jednoduchá: Je to běžící program.

Proces je běžící program.

Máme-li počítač vypnutý, žádný proces tam není. Programy jsou obvykle uloženy např. na disku, ale nejsou vykonávány. Po zapnutí počítače se spustí nejprve operační systém, pak můžeme spouštět aplikační programy. Spuštěním nějakého programu je vytvořen nový proces. Pro jednoduchost lze tento princip brát i doslova: Proces vznikne spuštěním programu. Spuštěním více programů vznikne více procesů. Opakovaným spuštěním stejného programu také vznikne více procesů.

Průvodce studiem

Pojem proces je obvykle dobře chápán i proto, že každý dnes zná aplikaci „Správce úloh“ systému Windows (viz obrázek 6). Tam je seznam právě běžících procesů zobrazen.



Obrázek 6: Aplikace Správce úloh (Task Manager) systému Windows XP.

Proces je víc než jen program v operační paměti. Kromě kódu programu obsahuje také aktuální činnost, kterou reprezentují hodnoty všech registrů procesoru, programový zásobník (používaný při rekurzi a pro ukládání lokálních proměnných) a samozřejmě také data. Proces obvykle má kromě tzv. globálních či statických dat také další data na haldě (anglicky heap).

Průvodce studiem

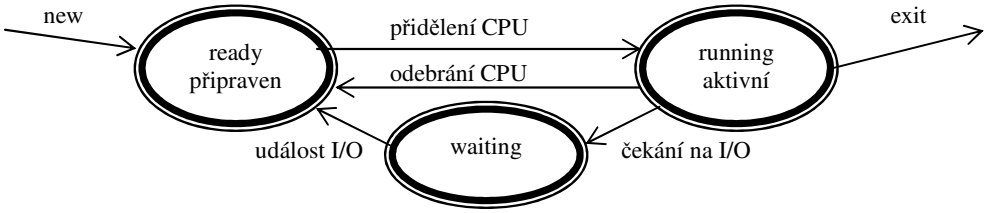
Data na haldě jsou vytvářena dynamicky, v řadě vyšších programovacích jazyků pomocí operátoru `new`. Oproti tomu statická globální data mají pevnou velikost i pozici v paměti, často jsou dokonce přímo uložena v souboru spolu s programem. V některých programovacích jazycích taková data definovat či používat nelze, nebo se jim říká jinak.

5.1.1 Životní cyklus procesu

Proces jakožto dynamický děj v přesném okamžiku vzniká, pak probíhá a v jiném přesném okamžiku končí. Během svého života proces prochází různými stavy; zde je důležitá role operačního systému jakožto správce procesů. Jelikož procesor fyzicky dokáže zpracovávat jen jednu úlohu, čili jeden proces, operační systém „dávkuje“ procesor a po krátkých časových úsecích jej střídavě přiděluje jednotlivým procesům. Tyto časové úseky jsou opravdu velmi malé, aby jej uživatelé pracující s počítačem nijak nepociťovali – u jednotlivého procesu mohou být řádově stovky přepnutí za sekundu a v celém počítači řádově tisíce (technicky vzato horní mez je dána rychlostí procesoru).

Životní cyklus procesu je obecně znázorněn na obrázku 7. Po vzniku procesu tento vždy přejde do stavu **ready**. V tomto stavu čeká, až operační systém rozhodne, že má proces běžet. V tom okamžiku přejde do stavu **running**. Proces pak běží, než nastane některá ze tří situací:

1. Ukončení procesu – Proces přestane běžet, když skončí. (Skončit přitom může právě jen ze stavu running.)
2. Čekání na I/O zařízení – Pracuje-li proces prostřednictvím operačního systému s nějakým externím zařízením, pak na něj velmi často musí čekat. V tom okamžiku proces přechází do stavu **waiting**. V tomto stavu je tak dlouho, než zařízení odpoví/dokončí operaci. Pak proces přechází do stavu **ready**.
3. Nenastane-li žádný ze dvou předchozích případů, pak po určitém čase operační systém sám procesu procesor odebere.



Obrázek 7: Stavy procesu.

Ukončený proces lze chápat jako proces v dalším (čtvrtém) stavu. To proto, že obvykle ještě nějakou dobu po skončení procesu o něm operační systém uchovává informace a je třeba nějak identifikovat, že proces již skončil. Podobně při vzniku procesu je tento vytvářen jakoby v dalším (pátém) stavu, který identifikuje, že proces je vytvářen a ještě nikdy nepřešel do stavu ready.

PCB nese informace o procesu.

Údaje o procesu si operační systém uchovává ve struktuře *PCB* (Process Control Block). Její přesná podoba samozřejmě závisí na konkrétním operačním systému, ale obsahuje obvykle minimálně informaci o aktuálním stavu procesu, hodnoty registrů procesoru, data používaná pro plánování běhu procesů, informace o používané paměti, statistické údaje o běhu procesu či informaci o právě používaných zdrojích.

5.1.2 Přepínání procesů

Přepínání procesů, čili střídavé předávání procesoru mezi jednotlivými procesy, zajišťuje operační systém. Samotné aplikační procesy se v ideálním případě nemusejí na této činnosti podílet a ani ji nemusí nijak pocítit – říkáme, že přepínání je „transparentní“. (Každý proces má tedy dojem, že v počítači běží jen on sám.)

Samotné přepnutí pak probíhá vždy pomocí přerušení a přes operační systém, tj. běžící proces je přerušen, běží OS a ten uloží aktuální stav procesoru do PCB záznamu přerušeného procesu. OS potom vybere, který proces bude dále běžet, obnoví stav procesoru dle jeho PCB záznamu procesu a ukončí přerušení. Místo do původního procesu se z přerušení vrací do nového procesu, který nyní běží.

Výběr, který následující proces bude běžet, musí být co nejrychlejší a také dostatečně spravedlivý. Algoritmus / část operačního systému toto zajišťující se nazývá *plánovač* (scheduler). Scheduler používá obvykle nějakou formu fronty čekajících procesů v kombinaci se systémem priorit. Podrobněji se u této problematiky zastavíme později v sekci vláken (kapitola 5.3).

Přepínání procesů je často podporováno přímo hardwarově a má to dva důvody:

1. Každý procesor může mít jiné registry a operační systém nemusí všechny znát. Přepínání procesů bez uložení a obnovení **všech** registrů by mělo fatální následky.

2. Přepnutí musí být co nejrychlejší. Ukládání jednotlivých registrů pomocí řady instrukcí by však příliš rychlé nebylo. Při hardwarové podpoře může celá akce proběhnout rychleji.

Procesory x86 jsou řešeny takto: Každý proces má svůj blok paměti TSS (Task State Segment), kam se při pozastavení jeho běhu stav procesoru automaticky ukládá (a při obnovení běhu procesu se odtud znovu obnoví). Registr TR (Task Register) je selektorem TSS běžícího procesu, odkazuje samozřejmě přímo do GDT. Pro evidenci procesu pak stačí v PCB nést hodnotu TR.

5.1.3 Kooperativní a preemptivní systémy

Z hlediska podporovaného či preferovaného způsobu přepínání procesu dělíme operační systémy na *kooperativní* a *preemptivní*.

V kooperativním systému aplikace spolupracují na předávání procesoru.

Kooperativní – toto je jednodušší model přepínání, kdy aplikační procesy samy musí spolupracovat na přepnutí. Každý proces musí co nejčastěji předávat řízení zpět do operačního systému. Proces tedy musí volat nějakou funkci operačního systému, z tohoto volání se pak procesor vrátí až po dalším přepnutí.

Nevýhodou tohoto přístupu je, že klade velké nároky na „slušnost“ aplikací – bude-li nějaký program méně slušný, může vážně narušit plynulý běh celého systému.

Preemptivní systém sám řídí přepínání procesoru.

Preemptivní – toto je dokonalejší model přepínání, kde operační systém v případě dlouhého běhu nějakého procesu sám (preemptivně – preventivně) odebere procesor a dá jej jinému procesu. Preemptivní systémy jsou spolehlivější, tvorba aplikací pro ně je přitom i jednodušší. Složitější je ale samozřejmě stavba preemptivního operačního systému.

Příkladem kooperativního systému je Windows 3.1 či starší verze MacOS. Příkladem preemptivního systému je Windows NT či Linux.

Průvodce studiem

U preemptivních systémů lze také diskutovat o tom, „jak moc“ je daný systém preemptivní. Např. Windows 95 byl preemptivní systém a v případě poruchy nějaké aplikace, kdy tato odmítla sama slušně vrátit procesor, operační systém procesor preemptivně odebral. Jenže

běh počítače byl v té chvíli neplynulý, viditelně zpomalený a bylo značně poznat, že „něco není v pořádku“. Současné verze Windows s jádrem NT tento neduh nemají.

5.2 Strategie přidělování procesoru

Jak již bylo řečeno, řídit přidělování procesoru je nutné k tomu, aby každý počítač i s pouze jedním procesorem mohl vykonávat více procesů současně. Tuto činnost má na starosti *plánovač úloh* (scheduler).

Scheduler se při plánování snaží dosáhnout následujících cílů:

1. Férovost ke všem procesům – Scheduler nesmí nikoho bezdůvodně zvýhodňovat. (Bezdůvodné by bylo například zvýhodnění vzniklé jako důsledek nekvalitního plánovacího algoritmu.)
2. Efektivita využití CPU – Procesor nesmí zahálet, pokud je pro něj práce. Proto scheduler nesmí přidělovat procesor těm procesům, které jej zrovna nepotřebují, ani na malý okamžik.
3. Minimalizace doby odezvy – Uživatel počítače by měl mít přednost před úlohami na pozadí, takže platí pravidlo: „Co je nejvíc vidět, to dělat napřed.“
4. Minimalizace doby průchodu systémem (turnaround) – U každého krátce existujícího procesu je žádoucí minimalizovat čas od spuštění po jeho ukončení.
5. Maximalizace odvedené práce (throughput) – Sledujeme-li dlouhodobě množství práce odvedené počítačem (např. za hodinu apod.), mělo by být maximální možné.

V konkrétních situacích se samozřejmě mohou priority těchto cílů lišit. Například u osobních počítačů je velmi důležitá doba odezvy, zatímco na serverech může být nejdůležitější např. throughput.

Schedulery obvykle používají následující dvě strategie či jejich kombinaci – systém cyklické obsluhy a prioritní systém.

5.2.1 Cyklická obsluha (round robin)

Princip cyklické obsluhy dodržuje především spravedlivost. Procesy jsou zařazeny do cyklické fronty a procesor je jim férově přidělován ve stylu „každý chvilku tahá pilku“.

Každý proces dostane procesor na určité pevně stanovené *kvantum* času. Problémem může být stanovení optimální velikosti kvanta, protože je-li kvantum příliš malé, roste režie systému (spojená s přepínáním procesů); je-li naopak kvantum příliš velké, prodlužuje se doba odezvy jednotlivých procesů (každý férově čeká ve frontě, až na něj přijde řada).

Další nedostatky tohoto systému, je-li použit v čisté podobě, jsou u důležitých systémových procesů – i ty musejí čekat ve frontě, až na ně přijde řada. Podobně pro systémy reálného času se tento princip nehodí vůbec. Obecně lze říci, že systém cyklické obsluhy je 100% spravedlivý, ale neumožňuje efektivní využití zdrojů.

5.2.2 Systém priorit

V prioritním systému je každému procesu přiděleno číslo označující jeho důležitost (čili prioritu). Procesor je pak přidělen vždy procesu s nejvyšší prioritou. Mapování priorit na čísla je obvykle lineární a může mít buď souhlasný, nebo opačný trend, tj. velká priorita může být

reprezentována malými, nebo naopak velkými čísly. V současnosti převládá druhý způsob: nejmenší priorita je reprezentována nulou a s rostoucími čísly priorita roste.

Výhodou prioritního systému je okamžitá odezva – ať už je v daný okamžik nejdůležitější cokoliv, daný proces má nejvyšší prioritu a zaručený přístup k procesoru. Nevýhodou je naopak nižší spravedlnost, kdy v extrémním případě se může i stát, že některý proces se k procesoru nedostane nikdy.

5.2.3 Praktické strategie

V praxi se užívá výhradně kombinací výše uvedených strategií (tj. ne některá z variant, ale vždy obě dohromady), čímž je dosaženo vyvážení jejich kladů a záporů. Základem takové kombinované strategie je prioritní systém, tj. přednost má proces s nejvyšší prioritou. Procesy se stejnou prioritou jsou obsluhovány cyklicky, čili na stejné úrovni priority vždy spravedlivě. Priorita se navíc může dynamicky měnit, tj. dlouho čekajícím procesům se zvyšuje, hodně pracujícím procesům se naopak snižuje, také procesům komunikujícím s uživatelem či periferiemi obecně se zvyšuje atp.

Strategie přidělování procesoru si ještě podrobněji probereme na příkladech konkrétních operačních systémů, nejdříve si však musíme představit pojem vlákna, se kterým plánování úzce souvisí.

5.3 Vlákna

5.3.1 Zavedení pojmu vlákno

Až dosud bylo povídání této kapitoly dosti intuitivní, pojem vlákna jej však poněkud zkomplikuje. V současných operačních systémech se totiž procesor nepřiděluje přímo procesům, ale vláknům. Samotná reprezentace vláken může být v jednotlivých operačních systémech různá, závisí to zejména na jejich stáří (jedná se totiž o poněkud později adoptovaný prvek), proto se kromě obecného povídání zastavíme také u příkladů konkrétních systémů.

Vlákno (anglicky thread, nebo delším thread of execution) je prvek reprezentující vykonávání kódu procesu. Podporu vláken do operačního systému dostaneme tak, že vyjmeme část odpovědností z procesu a osamostatníme je. Jak jsme si uvedli na začátku této kapitoly, informace o procesu jsou uchovány v PCB. Do vlákna z nich oddělíme především hodnoty všech registrů procesoru a údaje potřebné k plánování. Některé údaje budeme muset zdvojit – např. statistické údaje či informace o stavu bude nyní mít proces i vlákno (neboť každá entita má svůj stav). V procesu zůstanou informace o používané paměti a dalších zdrojích.

Jeden proces může mít několik vláken.

Vztah proces–vlákno však nemusí mít jen povahu 1:1 a v tom je hlavní význam vláken. Bude-li mít jeden proces vláken víc, každé z nich vykonává svůj kód, má své hodnoty registrů a svůj vlastní programový zásobník. S ostatními vlákny však sdílí operační paměť a všechny ostatní zdroje.

Zavedení vláken má zejména tyto výhody:

- Přepnutí mezi vlákny je rychlejší než přepnutí mezi procesy. Přepne se totiž jen to, co souvisí s vláknem – tj. stačí vyměnit obsahy pracovních registrů. Při přepnutí procesů naopak musí dojít k přepnutí celého kontextu, včetně mapování paměti atp. (Moderní procesory podporují přepínání hardwarově, proto nemusejí být rozdíly nijak dramatické.)
- Vlákna spolu sdílejí prostředky, čímž je mohou šetřit. Např. potřebují-li dvě vlákna pracovat se stejným datovým souborem, stačí jej načíst do paměti jen jednou.

Vlákna sdílejí prostředky.

- Zavedení vláken umožňuje značně zjednodušit některé typy programů. Potřebuje-li program ze své povahy vykonávat víc různých činností současně, je jednodušší řešit to

pomocí vláken než velmi složitými úpravami kódu jednoho vlákna. (Např. textový editor Word používá druhé vlákno pro kontrolu pravopisu na pozadí. Bez možnosti použití vlákna by tato úloha byla velmi složitá.)

5.3.2 Vlákna v kontextu dřívějších znalostí

V kontextu našich dosavadních znalostí přináší pojem vlákna řadu nových skutečností. Neformálně řečeno, vlákno je „vykonávání kódu“, zatímco proces je „paměť a další prostředky“. Čili proces má paměť, včetně paměti s kódem programu. Vlákno tento kód vykonává, ale samo svůj vlastní kód nemá. Dokonce i programový zásobník, který má vlákno samo pro sebe, je v paměti procesu. Vlákna tedy mohou používat i zásobníky jiných vláken téhož procesu – nepoužívají je pomocí push/pop, ale mohou získat adresu na proměnné tam uložené a pracovat s ní.

Vlákno nemá svůj vlastní kód.

Průvodce studiem

Vlákna se masově rozšířila především díky operačnímu systému Windows NT (a později Windows 95). Systémy MS-DOS a Windows 3.1 naopak vlákna neznaly. Systémy Windows NT 3.51 SP3 a novější znají i fibers, další variantu vláken.)

Zavedením vláken se mění životní cyklus procesu. Proces startuje spolu s jedním vláknem, končí v okamžiku ukončení posledního vlákna. Tři stavy uvedené na obrázku 7 se vztahují ke každému vláknu samostatně. Procesor je totiž přidělován jednotlivým vláknům, nikoliv procesům. Proces nevykonává kód, tudíž procesor dostat nemůže.

První vlákno tedy vzniká spolu s procesem a začne vykonávat jeho kód „od začátku“ (kde je onen správný „začátek vykonávání kódu“ pro první vlákno, je uvedeno v souboru s programem). Každé další vlákno může začít vykonávat kód z jiného místa (a obvykle tomu tak je); další vlákna přitom mohou vznikat jen tak, že je vytvoří některé vlákno již existující.

Přepínání se týká přímo vláken. Zdali při přepnutí dojde k přepnutí také procesu, je pouze sekundární záležitost. Strategie plánování se také týkají vláken – round robin i priority mají jednotlivá vlákna. Vlákna téhož procesu jsou tedy plánována nezávisle na sobě a každé může mít jinou prioritu. Proces sám o sobě prioritu k ničemu nepotřebuje, protože se plánování neúčastní.

Průvodce studiem

Aby nebylo vše tak přímočaré, ve Windows mají i procesy prioritu. S procesy se při plánování sice přímo neoperuje, ale priorita procesu se sčítá s prioritou vláken, které do něj patří. Změnou priority procesu tedy ovlivňujeme priority všech vláken. Tento vztah však ve Windows není přímo lineární, podrobněji se s ním seznámíme později.

Každé vlákno samozřejmě má vlastní programový zásobník. Jelikož ale vlákno ve skutečnosti sdílí adresový prostor s ostatními vlákny v rámci procesu, jeho programový zásobník existuje ve skutečnosti v rámci adresového prostoru procesu. Data na zásobníku jednoho vlákna jsou tedy přístupná i jiným vláknům v rámci procesu.

Každé vlákno má svůj zásobník.

5.3.3 Vlákna na single–user úlohách

Vlákna se velmi často používají i na single–user úlohách (tj. v programech používaných jedním uživatelem). Proč tomu tak je? Uveďme nejobvyklejší důvody:

Práce na popředí a na pozadí – Jedno vlákno řeší GUI (tj. pracuje na popředí), další vlákno nebo i několik vláken něco počítá na pozadí. Příkladem je kontrola pravopisu na pozadí, kterou provádí textový editor.

Asynchronní zpracování – Toto je případ, kdy nějakou činnost spouštíme asynchronně, tj. bez ohledu na průběh výpočtu programu (např. časovačem). Příkladem může být zálohování dat v pravidelných intervalech či zobrazování času na hodinách.

Rychlejší vykonávání kódu – Má-li počítač více než jeden procesor (CPU), může je cíleně využít k celkovému zrychlení výpočtu.

Modulární architektura – Uvádí se, že modulární vícevláknové řešení velkých systémů je obecně flexibilnější a efektivněji využívá systémové zdroje. Příkladem takového systému je např. DirectShow.

5.3.4 Technická realizace vláken

Vlákna jsou implementována knihovnou, nebo přímo v jádru.

Vlákna mohou být do systému zanesena dvojím způsobem – buď je jejich podpora věcí nějaké doplňující knihovny (která případně může být i dodávána spolu s operačním systémem, ale jedná se o běžnou knihovnu, která nevyžaduje žádné speciální funkce po jádru operačního systému), nebo jsou vlákna podporována přímo jádrem. Oba tyto přístupy mají své výhody a nevýhody.

Jsou-li vlákna implementována v knihovně, což je historicky první způsob, jak byla vlákna zavedena, z hlediska systému se vícevláknový program stále jeví jako jediný proces. Výhodou tohoto přístupu je, že přepínání vláken je o něco málo rychlejší, neboť při něm nedochází k přechodu mezi uživatelským a privilegovaným režimem CPU. Aplikace, přesněji řečeno knihovna také může ovlivňovat plánování vláken – když si vlákna přepíná sama, může používat jiný plánovací algoritmus, než používá operační systém. Nevýhodou tohoto přístupu je právě fakt, že z pohledu operačního systému se aplikace jeví jako jediný proces. Pokud některé vlákno např. bude čekat na dokončení I/O operace, aplikace bude blokována na úrovni procesu a nepoběží žádné vlákno. Obecně: Vlákna jednoho procesu nikdy nemohou běžet současně, takže ani nelze využít více procesorů (pokud by v počítači byly).

Podpora vláken v jádru operačního systému odstraňuje nevýhody zmíněné v předchozím odstavci. Přepínání vláken je zde sice pomalejší, neboť jde vždy cestou přes jádro, z ostatních pohledů je však tento přístup výhodnější. Vlákno je zde již jednotkou přidělování procesoru na úrovni operačního systému, takže vlákna jednoho procesu mohou běžet současně (máme-li v počítači více CPU), stejně tak čekání jednoho vlákna ostatní vlákna neblokuje.

Podporuje-li operační systém vlákna přímo, nic mu nebrání poskytovat současně i výše zmíněnou knihovnu pro vlákna. Aplikace pak mohou využívat kombinace obojího a tím dosáhnout efektivnějšího využití zdrojů. Nevýhodou je samozřejmě vyšší pracnost pro programátora.

5.3.5 Vztah proces–vlákno

Obvyklý vztah mezi procesy a vlákny je typu 1:N, tj. jeden proces má několik vláken. Na počítačových klastrech je možno potkat také vztah N:1, kdy jedno vlákno může putovat mezi počítači. Tato varianta je samozřejmě využívána zřídka, protože přesun vlákna na jiný výpočetní uzel je časově náročná operace. Podobně můžeme definovat i zobecněný případ M:N, kdy každý proces má několik vláken a ta mohou být rozmístěna na různých výpočetních uzlech. Tuto variantu považujme pro její technickou náročnost za zcela teoretickou. Poslední nezmíněný případ, tedy vztah 1:1, je případem systému, kde vlákna nejsou vůbec zavedena (historické systémy).

Shrnutí

Tato kapitola je první z bloku věnovaného správě procesoru. Představili jsme si především dva základní pojmy: proces a vlákno. Vysvětlili jsme si také, jak se plánuje přidělování procesoru procesům a vláknům a jak jsou vlákna technicky realizována v systému. V následující kapitole se podíváme na procesy a vlákna v konkrétních operačních systémech.

Pojmy k zapamatování

- proces
- životní cyklus procesu
- vlákno
- plánovač úloh
- preempce
- cyklická obsluha (round–robin)
- priorita

Kontrolní otázky

1. *Popište obecný životní cyklus procesu.*
2. *Vysvětlete, proč TSS odkazuje přímo do GDT, a ne do LDT.*
3. *Proč je přepínání procesů obvykle řešeno hardwarově? Má to i nějaké nevýhody? Proveďte diskuzi.*
4. *Plánovací algoritmus určuje pořadí vykonání procesů. Máme-li n procesů, které mají proběhnout v nějakém pořadí vždy celé, kolik možných variant plánů existuje?*
5. *Které z následujících operací vyžadují privilegovaný režim? Nastavení časovače, čtení hodin, instrukce simulující zastavení programu výjimkou, zákaz přerušení, modifikace tabulek informací o I/O zařízeních, přepnutí z uživatelského do privilegovaného režimu, přístup k I/O zařízení.*
6. *Kooperativní multitasking je považován obecně za méně dokonalý. Jaké v tomto systému vidíte výhody? A kterou z jeho nevýhod vidíte z dnešního pohledu jako klíčovou? Proveďte diskuzi.*
7. *Strategie cyklického a prioritního přidělování procesoru se v praxi v čisté podobě nepoužívají. Uveďte několik příkladů, k jakým problémům by nasazení v jejich čisté podobě vedlo.*
8. *Uveďte několik konkrétních příkladů, kde více vláken pomáhá zkvalitnit práci v jednouživatelském systému.*
9. *Uveďte méně obvyklé vztahy proces – vlákno. Který způsob vazby považujete za zcela nejokrajovější (nejméně obvyklý)?*

Cvičení

1. Napište program pro sečtení všech prvků v poli, který využije vlákna k rychlejšímu běhu. K simulaci víceprocesorového počítače vložte za každou dílčí operaci součtu čekání 1 milisekundu. Funkci programu ověřte změřením výpočetního času pro různé počty vláken (na dostatečně velkém poli).

6 Správa procesoru v praxi

Studijní cíle: V této kapitole navážeme na znalosti z kapitoly předchozí a seznámíme se s praktickými implementacemi správy procesoru, konkrétně na systémech Windows NT a Unix. Systém NT si v některých bodech popíšeme poněkud detailněji, protože se jedná v současnosti o nejrozšířenější operační systém počítačů PC.

Klíčová slova: správa procesoru, Unix, POSIX, Windows NT, priorita, afinita

Potřebný čas: 150 minut.

6.1 Unix – Vytvoření, plánování a stavy procesu

Systémy označované jako Unix jsou operační systémy určené primárně pro serverové počítače a existují již několik desetiletí. Za tu dobu vzniklo velké množství verzí, my si popíšeme především obecně platné vlastnosti unixových systémů. Do této skupiny patří také systém Linux, což je nekomerční operační systém, který poskytuje stejné rozhraní (API) zvané POSIX. Právě rozhraní POSIX je spojujícím prvkem všech současných unixových systémů, díky tomuto rozhraní jsou programy mezi různými operačními systémy z této rodiny do jisté míry přenositelné.

Průvodce studiem

Ačkoliv tato kapitola je rozdělena na povídání o Unixu a o Windows, i systém Windows NT v některých verzích umí rozhraní POSIX. Není to však jeho primární rozhraní a v praxi se používá jen velmi zřídka.

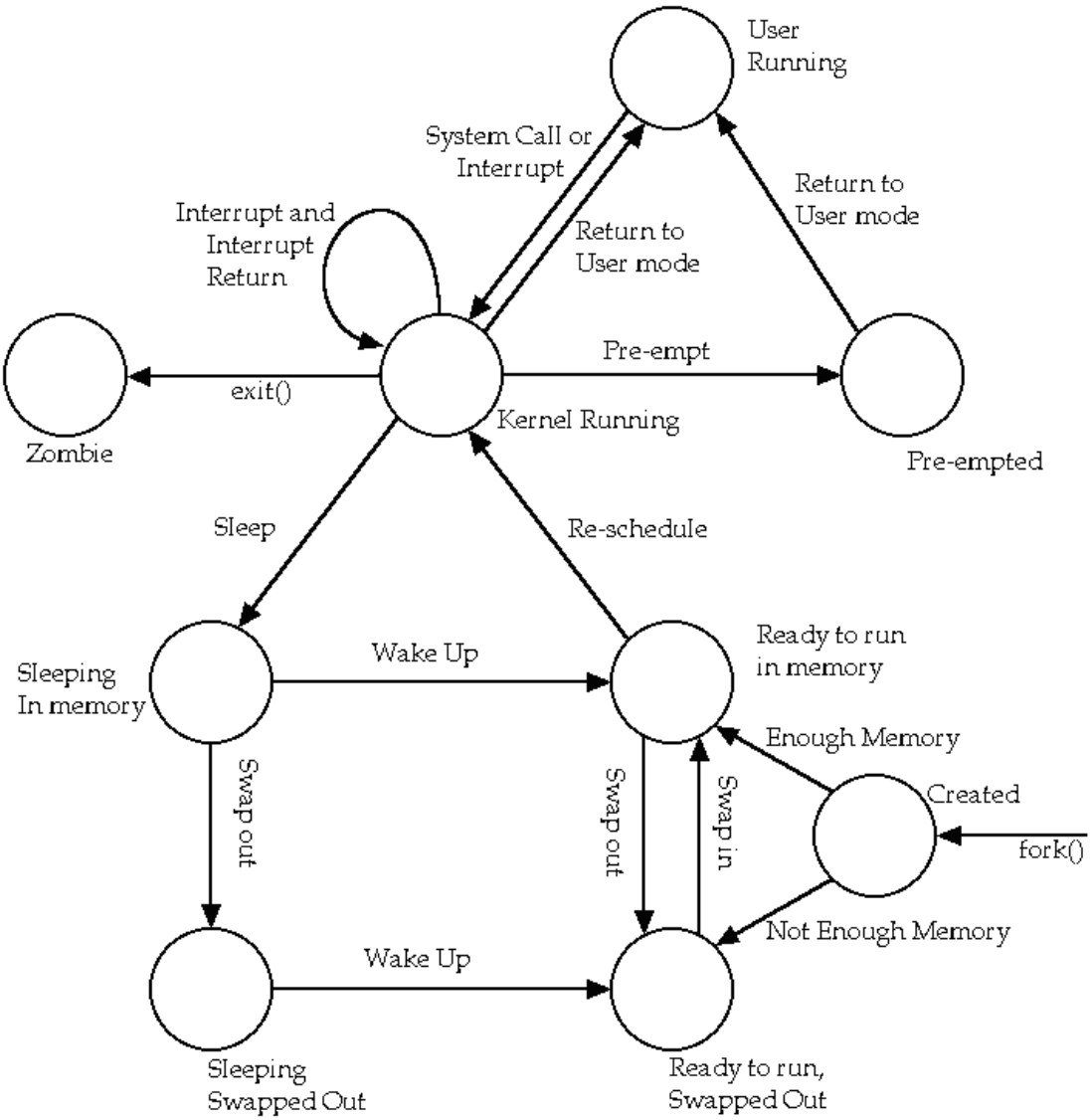
6.1.1 Stavy procesu

Stavy procesů v Unixu jsou znázorněny na obrázku 8. Rozlišujeme zde devět stavů:

1. Created – Proces je vytvořen.
2. Ready to run in memory – Proces je připraven k běhu a čeká, až na něj vyjde řada (přesněji řečeno: až jej plánovač vybere k běhu).
3. Ready to run, swapped out – Proces je připraven k běhu, ale není v paměti.
4. Sleeping in memory – Proces čeká na prostředek aj.
5. Sleeping swapped out – Proces čeká a navíc není v paměti.
6. Kernel running – Běží jádro systému.
7. User running – Proces běží (pouze zde se vykonává vlastní kód programu).
8. Pre-empted – Běh procesu je násilně přerušen (preempce).
9. Zombie – proces skončil, ale ještě je o jeho existenci záznam v systému.

Při studiu životního cyklu a stavů procesu samozřejmě nejde jen o to pojmenovat nějak oněch 9 různých stavů, ale důležité je také všimnout si a pochopit přechody mezi stavy. Například volání služeb OS probíhá přímo v aplikačním procesu (tj. proces je přepnut do kernel režimu a sám si vykoná kód služby OS). Během vykonávání služeb OS nemůže být proces přerušen preempcí. Proces vykonávající kód jádra však může být přerušen (hardwarovým i softwarovým) přerušením. Probíhající obsluha přerušení může být také přerušena dalším přerušením.

Popišme si ještě, jak v Unixu funguje preempce: V základním případě se jí účastní 2 procesy, označme si je A a B. Proces A je v režimu „User Running“ a vykonává svůj kód. Preempce



Obrázek 8: Stavy procesu v Unixu. [Bac86, BoCe05]

je navázána na časovač. Jakmile nastane přerušení časovače, je aktivována obsluha přerušení (proces A je tedy přepnut do režimu jádra) a aktivován kód plánovače. Plánovač (nyní běžící jako proces A) přepne řízení na proces B. Plánovač (nyní běžící jako proces B) pak změní stav procesu A na „Pre-empted“ a ukončí obsluhu přerušení. Tím je proces B přepnut zpět do jeho předchozího stavu (obvykle „User Running“).

6.1.2 Vytvoření procesu

Procesy v Unixu vytvářejí hierarchii.

Příkaz fork klonuje (tj. zdvojí) proces.

Procesy v Unixu vytvářejí hierarchii, tj. každý proces má svého rodiče (což je proces, kterým byl vytvořen). Při nabíhání systému vzniká iniciační proces, který je kořenem stromu, další procesy mohou vniknout jedině klonováním nějakého existujícího procesu – k tomu slouží příkazy *fork* a *exec*.

Příkaz fork klonuje volající proces, čili po zavolání se funkce „vrací“ do dvou identických procesů. Rodiče a potomka rozlišíme pouze dle návratové hodnoty této funkce (rodičovi vrací číslo procesu potomka, potomkovi vrací nulu), jinak jsou skutečně identické. Systém se přitom snaží o maximálně úsporné řízení paměti – kód a většina dat je v obou těchto procesech totožná, paměť je tedy sdílená. K implementaci klonování paměti se ještě vrátíme v kapitole 9.

Po vytvoření klonu pak příkazem exec nahradíme kód volajícího programu novým programem.

Tím je vytvoření procesu dokončeno.

6.1.3 Plánování procesů

Ve starších verzích Unixu se používal následující systém plánování: Systém používá tzv. prioritní frontu, čili několik front procesů, kde každá fronta odpovídá jiné prioritě. Procesy jsou obsluhovány dle priority a na stejné úrovni cyklicky. Klíčový je přitom princip zařazování procesů do front: Nový proces dostane vysokou prioritu. Jak proces stárne, jeho priorita je pak snižována; proces je přeřazen do nižší úrovně priority vždy při preempci, zatímco při dobrovolném odevzdání procesoru je mu priorita zachována.

Tento přístup samozřejmě upřednostňuje krátké úlohy, což může být výhodné. Nevýhodou může být hladovění (starvation) starších procesů. Například při rychlém přidávání nových procesů se ty nejstarší nedostanou vůbec k procesoru, protože nový proces vždy dostane přednost a běží alespoň 1 časové kvantum.

Od verze 4 používá Unix tři třídy priorit:

- Reálný čas (pro kritické procesy)
- Systémové procesy
- Uživatelské procesy

Proces se nemůže dostat do jiné třídy priorit, než ve které vznikl.

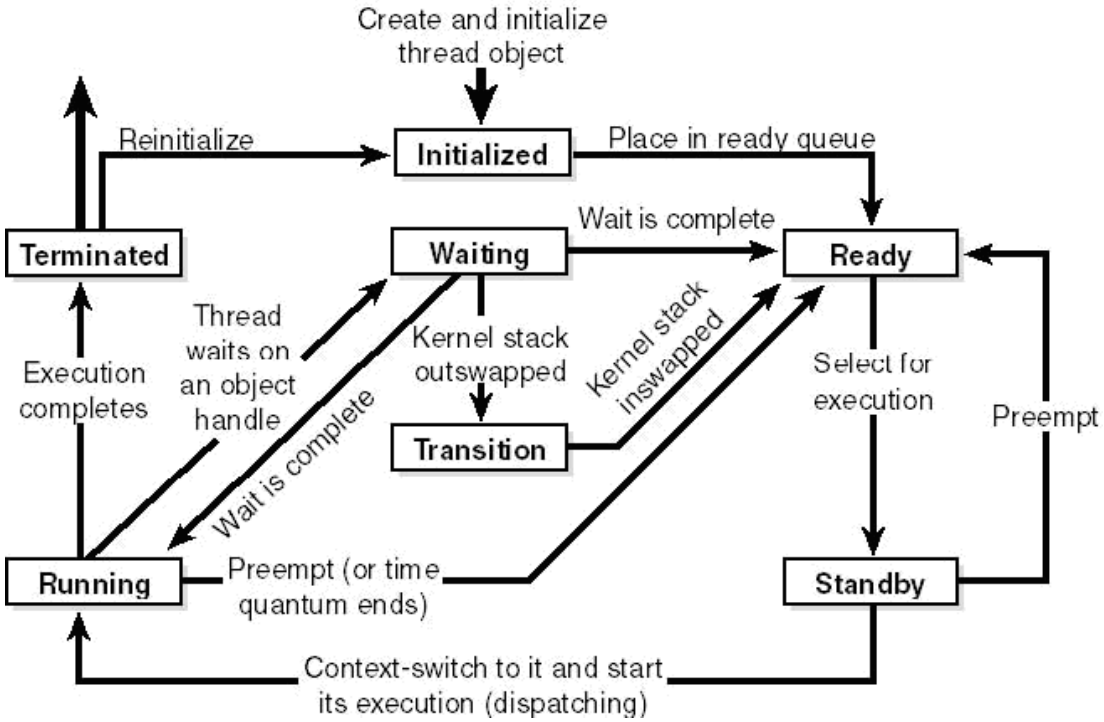
Plánování procesů v Linuxu je podrobně popsáno např. v knize [BoCe05] (viz také http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html).

6.2 NT – Vytvoření procesu, plánování a stavy vlákna

6.2.1 Stavy vlákna

V NT se plánování zúčastňují vlákna (nikoliv procesy), jejich možné stavy jsou znázorněny na obrázku 9. Rozlišujeme jich sedm:

1. Initialized – Pomocný stav, přechodně během inicializace vlákna.
2. Ready – Čekající (pouze z těchto vláken vybírá scheduler další k běhu).
3. Standby – Vlákno je připraveno k běhu na konkrétním procesoru. Jen jedno vlákno může být v tomto stavu (na každý procesor). Tento stav může skončit
   (a) přechodem do stavu Running (když se Running stav uvolní).
   (b) preempcí (jiné vlákno začne být Standby, toto je vráceno do fronty Ready).
4. Running – Běžící (je vykonáván kód programu). Tento stav může skončit
   (a) preempcí s přepnutím na vlákno vyšší priority. Toto vlákno je vráceno do Standby nebo na začátek své fronty v Ready.
   (b) uplynutím kvanta. Toto vlákno je pak umístěno na konec své fronty v Ready.
   (c) dobrovolným přechodem do stavu Waiting.
   (d) ukončením vlákna.
5. Waiting – Čekající (na nějaký objekt). Po skončení čekání se vlákno přepne do stavu Ready, nebo přímo do Standby či Running (to nastane, pokud má vysokou prioritu).
6. Transition – Vlákno má zásobník mimo fyzickou paměť (odstránkovaný). Jakmile se zásobník dostane zpátky do paměti, vlákno přechází do stavu Ready.



Obrázek 9: Stavy procesu v NT. [SoRu05]

7. Terminated – Ukončeno. Z tohoto stavu lze vlákno znovu oživit do stavu Initializing.

Nejvíc vláken je obvykle ve stavech Waiting nebo Ready. Plánovač vybírá vlákna pro běh právě jen z Ready vláken. Ta má organizována do speciální prioritní fronty, technicky je samostatná fronta pro každou hodnotu priority. Jak si ukážeme později, vlákna jsou obvykle přepínána v okamžiku přerušení časovače, kdy je dosud běžící vlákno zařazeno na konec své fronty a pro běh je vybráno jiné vlákno. Jakmile se však objeví vlákno s vyšší prioritou ve stavu Ready, než má vlákno Running, dojde k okamžitému přepnutí. (Obdobně to platí i pro stav Standby.) Tomuto okamžitému přepnutí říkáme preempce – běžící vlákno je vráceno na začátek své fronty (či do stavu Standby) a nové vlákno ihned běží. Všimněte si, že preempce vrací vlákno na začátek fronty, nikoliv na konec, kam se dostává jedině po dokončení celého časového úseku (až do přerušení pravidelným časovačem) bez preempce.

6.2.2 Vytvoření procesu

Proces v NT vzniká spuštěním programu.

Systém NT nevyžaduje u procesů vztah rodič–potomek jako unixové systémy. Nový proces vzniká vždy spuštěním nějakého spustitelného souboru (funkce *CreateProcess*). NT přitom používá princip tzv. subsystémů, díky němuž lze v rámci jednoho operačního systému provozovat programy určené pro různé operační systémy. Spustitelný soubor má obvykle známou příponu exe a je ve formátu PE (Portable Executable). Tyto soubory mají hlavičku, podle které OS mj. pozná, v jakém subsystému se má program spustit. Spouštěcí kód v jádru systému NT je jen jeden a umožňuje spouštění programů mezi různými subsystémy. V jednom souboru lze také mít více verzí programu, což se však v praxi příliš nepoužívá.

Průvodce studiem

Příkladem uložení více verzí programu do jednoho souboru je ona známá hláška „This program cannot be run in DOS mode.“, která se zobrazí, spustíme-li Win32 program v DOSu. Systém MS-DOS sice neumí poznat, že jde o program pro Windows (resp. Win32), ale při spuštění spouští první verzi programu v souboru, kde je krátký program právě pro

systém MS-DOS, který vypíše onu hlášku. Při spuštění téhož souboru ve Windows systém najde Win32 záznam a dá mu přednost. Ve spustitelných souborech typu PE lze takto mít pohromadě verze pro různé operační systémy či procesory (x86, Itanium, Alpha apod.).

Množina podporovaných subsystémů se liší dle verze Windows. Během let jsme se mohli setkat kromě subsystému Win32, což je samozřejmě primární rozhraní systému, které nejčastěji používáme, také se subsystémy Win16, DOS, OS/2 či POSIX. Některé tyto subsystémy jsou mezi sebou provázané, čili nejde vždy o čisté implementace fungující jen pomocí funkcí jádra NT. Samotné jádro systému NT poskytuje tzv. „NT API“, což je nedokumentovaná sada funkcí, kterými jsou pak implementovány jednotlivé subsystémy. NT API poskytuje univerzální funkcionalitu, aby pomocí něj mohly být implementovány všechny subsystémy.

Průvodce studiem

Rozhraní Win32 se proslavilo zejména díky systému Windows 95. Prvním systém, kde bylo implementováno, však bylo Windows NT. Ani jeden z těchto systémů však Win32 nemá jako primární rozhraní. Zatímco NT má své vlastní NT API, Windows 95 má Win32 implementováno nad starším 16bitovým rozhraním Win16, které převzalo ze své předchozí verze (Windows 3.11).

Jistou nevýhodou je, že NT API je skutečně nedokumentované. Má to ale i jeden pozitivní důsledek: Programy napsané pro nějaký subsystém nepoužívající přímo NT API jsou přenositelné i na jiné operační systémy s daným API. Navíc je tím zajištěna jistá čistota programů, neboť není možno míchat různá API dohromady.

Průvodce studiem

Univerzálnost systému NT má i některé poměrně zajímavé důsledky. Např. souborový systém NTFS je možno použít pro všechny subsystémy, takže umí pracovat jak v režimu bez rozlišení velkých a malých písmen (pro Win32), tak v režimu velikost písmen rozlišujícím (pro POSIX). Podobně příkaz fork pro duplikování procesu je přítomen i v NT, i když ve Win32 režimu jej nemůžeme přímo použít.

6.2.3 Plánování vláken

Plánování v NT je na úrovni vláken.

Jak již víme, v systému NT se plánují vlákna, nikoliv procesy. Z toho nám hned vyplyne jedna důležitá vlastnost NT: Založí-li si jeden proces 9 vláken (funkcí *CreateThread*) a druhý jen jedno, první proces bude mít 90% procesoru pro sebe. Toto samozřejmě platí za předpokladu, že všechna tato vlákna budou mít stejnou prioritu, neboť plánování v NT je prioritní a potom cyklické na stejné úrovni priorit. Vlákno s vyšší prioritou má tedy vždy přednost. Aby nedocházelo k hladovění (starvation) vláken s nejnižší prioritou, v systému jsou další doplňkové mechanizmy – ty si představíme níže.

Vlákno vybrané k běhu dostane CPU na dobu jednoho časového kvanta. Různé verze OS mohou mít (a mají) různě dlouhá kvanta, rozlišujeme zde systémy Workstation (např. Windows XP nebo Windows 2000 Professional) a Server (např. Windows Server 2003 nebo Windows 2000 Server). Běžící vlákno může být přerušeno i předčasně (před skončením přiděleného kvanta), pak hovoříme o preempci. Systémy NT již od poměrně dávných časů také podporují víceprocesorové počítače a pro tento účel má proces i každé jeho vlákno také afinitu (anglicky affinity) – je to bitová maska určující, na které CPU může být vlákno plánováno. Výchozí

nastavení je, že vlákno může běžet na libovolném CPU; u jednoprocesorových počítačů afinita samozřejmě nemá význam.

Na plánování nemá vliv, zda vlákno právě vykonává kód jádra systému, nebo běžný uživatelský kód. Jakmile má menší prioritu než jiné vlákno, je přepnuto. Vykonávání kódu jádra systému proto může být plánovačem de facto přerušeno, systém tím nijak netrpí. Kritická místa, například kód samotného plánovače, jsou před přepnutím chráněna pomocí systému přerušení. Přerušení má v počítači z principu vyšší prioritu než jakékoliv vlákno. Kód běžící jako přerušení tedy nemůže být přerušen žádným vláknem. Počítače PC ještě rozlišují mnoho různých přerušení, které mezi sebou také mají prioritu a platí, že kód obsluhy přerušení může být přerušen jen přerušením s vyšší prioritou, než má on sám. Systém priorit přerušení je vlastně úplně samostatný systém priorit, který je nadřazen prioritám vláken. V NT má také hodnoty 0–31 a vlákna mohou dostat prioritu přerušení jen 0 nebo 1, plánovač má vždy 2 a další vyšší čísla jsou pro skutečná přerušení v počítači.

Scheduler/dispatcher

Částí jádra odpovědnou za plánování a přepínání vláken je scheduler/dispatcher, jednoduše řečeno „plánovač“. Ten je aktivován když

a) nějaké vlákno přejde do stavu Ready (nově vytvořené nebo po skončení čekání). Když toto má vyšší prioritu, přijde na řadu preempce.

b) běžící vlákno opustí stav Running (konec kvanta, konec vlákna, či přechod do stavu Waiting).

c) dojde ke změně priority (libovolného) vlákna (buď cíleně voláním funkce OS, nebo když prioritu vláknu změní sám systém).

d) dojde ke změně afinity (libovolného) vlákna.

Číslování priorit

Priority jsou v NT číslovány 0–31.

Priority vláken jsou v jádru NT číslovány 0–31, je jich tedy přesně 32. Vyšší prioritu než 31 přitom mají již zmíněná přerušení. Systém používá některá vyšší čísla také pro označení nejvíce preferovaných operací. (Priorita běžícího kódu se dočasně zvýší, systém se tedy tváří, jakoby to bylo přerušení, čímž zajistí, že daný kus kódu nebude během vykonávání přepnut na žádné jiné vlákno. Priority vláken a priority přerušení fungují ve skutečnosti nezávisle na sobě, ale můžeme je chápat i jako společnou věc, což pomůže snáze pochopit chování systému v určitých speciálních situacích.)

V souvislosti s prioritami je nutno rozlišit základní pojmy:

Třída priority je vlastností procesu Win32. Je to vždy jedna z několika hodnot popsaných níže. (Není to číslo, ale pojmenovaná hodnota z výčtového typu.)

Priorita procesu je hodnota jádra NT v rozmezí 0–31.

Priorita vlákna je udávána relativně (tj. odchylkou) vzhledem k prioritě jeho procesu. Sečtením této hodnoty s prioritou procesu pak získáme opět číslo 0–31, podle kterého se pak plánují vlákna plánovačem (až na výjimky, viz dále v textu).

Scheduler/dispatcher dynamicky mění priority vláken tak, aby systém jako celek běžel co nejlépe (podrobněji vysvětlíme níže). Proto priorita vypočtená jako součet priority procesu a relativní priority vlákna není často výslednou prioritou, ale jen hodnotou, která se dále upravuje systémem.

Aby nedocházelo k nechtěným situacím, priority jsou rozděleny do tří samostatných rozsahů a platí pravidlo, že scheduler/dispatcher nikdy nezmění prioritu vlákna tak, aby se dostalo do jiného rozsahu.

Idle úroveň (0) Toto je vyhrazeno pro „Zero page thread“, což je jedno systémové vlákno starající se o nulování nepoužívané paměti. Jelikož má nejnižší prioritu, běží pouze tehdy, když není vůbec nic jiného na práci. Systém NT totiž z bezpečnostních důvodů nikdy nepřidělí paměť procesu bez toho, aby ji vynuloval. (Pokud je CPU vytížen a Zero page thread neběží, paměť je nulována až v okamžiku alokace.) Jakmile jsou všechny volné paměťové stránky vynulované, CPU spí (vykonáním instrukce hlt spí do dalšího přerušení).

Dynamické úrovně (1–15) Zde jsou běžné uživatelské aplikace i část operačního systému. Číslo 15 je obvykle používáno jen časovačem (aby měl vyšší prioritu než vše ostatní).

Real-time úrovně (16–31) Toto je pouze pro kritické části operačního systému. (Nutno však podotknout, že ani zvýšení priority do této hladiny nezaručuje přidělení CPU v nějakém pevném čase. Proto NT není systémem reálného času.)

Průvodce studiem

Pro zajímavost: Zero page thread není ve skutečnosti běžné vlákno s prioritou 0, ale jen kód, který ani nemá žádnou prioritu a jednoduše běží, když plánovač nemá žádné jiné vlákno k běhu. Prioritu 0 ve skutečnosti nelze žádnému vláknu nastavit a kód Zero page thread je vykonáván skutečně jen tehdy, když plánovač nemá jiného kandidáta na běh. (V programu Správce úloh se Zero page thread zobrazuje pod tajemným označením „Nečinné procesy systému“ a i další podobné programy mu dávají jiná dosti podivná jména.)

Třídy priorit

Subsystém Win32 používá systém tzv. tříd priorit. Každý proces patří do jedné z následujících šesti tříd.

Real-time (24) reálný čas (používají jen kritické části jádra systému a ovladačů zařízení)

High (13) vysoká priorita

Above normal (10) nad normálem (není ve starších verzích Windows)

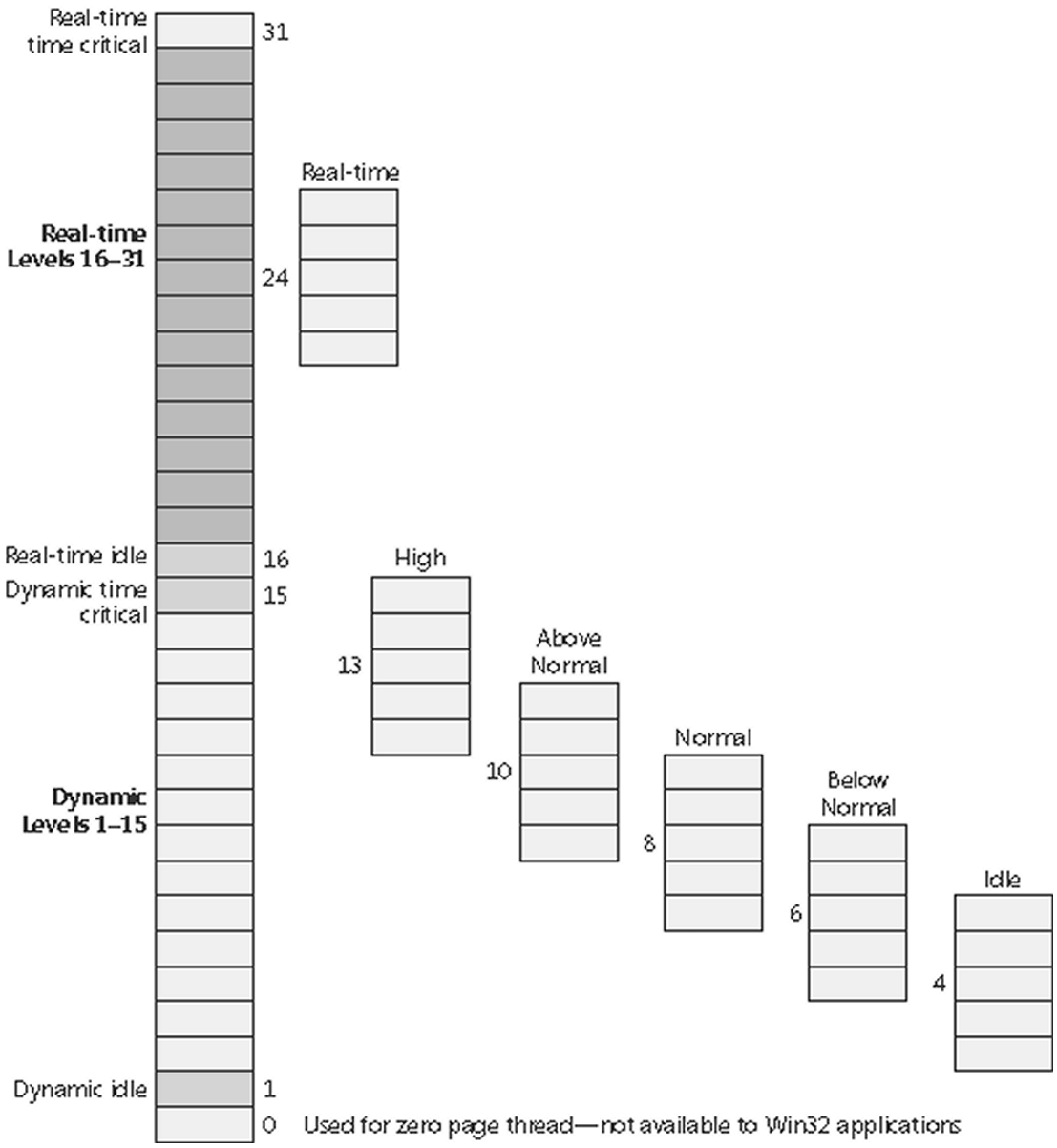
Normal (8) normální priorita

Below normal (6) pod normálem (není ve starších verzích Windows)

Idle (4) nečinný proces

Třída priority určuje, přibližně jakou prioritu budou mít všechna vlákna daného procesu. Číslo v závorce za názvem třídy je právě tou hodnotou priority, která je výchozí. Můžeme jej nazývat bázovou prioritou procesu.

Třída priority je tedy vlastností procesu a určuje výchozí prioritu pro všechna vlákna tohoto procesu. Každé vlákno ve Win32 má prioritu určenou jako relativní hodnotu (normálně tedy 0, vyšší priority jsou kladná čísla, nižší priority jsou záporná čísla). Změníme-li třídu priority běžícího procesu, změní se tím priority všech jeho existujících vláken. Relativní priorita vlákna přitom nemůže být libovolné číslo, ale jen jedna ze sedmi konstant, které numericky vyjadřují hodnoty -15, -2, -1, 0, +1, +2, +15. Vlákna jednoho procesu tedy nemohou mít libovolnou prioritu, ale jen několik vybraných hodnot v okolí bázové priority procesu nebo jednu z mezních hodnot 1 (idle) nebo 15 (time critical). Graficky ukazuje mapování priorit Windows API do NT čísel obrázek 10.



Obrázek 10: Mapování priorit Windows API do čísel priorit NT jádra. [SoRu05]

Průvodce studiem

Jak uvádí [SoRu05], některé systémové procesy mají i jiné bázové priority, než ty, které jsou ve výše uvedeném seznamu tříd priorit. Tyto však nelze nastavit přes Windows API, ale jen přímým voláním příslušné funkce NT jádra. (Toto nezkoušejte, je to nedokumentovaná funkcionalita a slouží to k jemnému vyladění běhu systému.)

Vlákno má bázovou a aktuální prioritu.

Z pohledu plánovače má každé vlákno ve skutečnosti dvě hodnoty priority: bázovou (base) a aktuální (current). Bázová priorita vlákna je součtem hodnoty třídy priority procesu a relativní priority vlákna, jak jsme si vysvětlili výše. Aktuální priorita vlákna je druhá hodnota, která může být vyšší než bázová. Smyslem tohoto řešení je, že zatímco sám program a potažmo programátor ovlivňuje jen bázovou prioritu, operační systém ji může dle potřeby dočasně zvyšovat (anglicky: boost). Tato zvýšení (boosting) ovšem programátor nemůže běžnými prostředky ovlivnit ani zjistit. Dodejme, že ke zvyšování priorit dochází jen u vláken dynamickém rozsahu (čili u vláken s prioritou 1–15 může OS prioritu „tajně“ zvyšovat až na 15).

Časová kvanta

Již hodně jsme slyšeli o časových kvantech sloužících jako základní jednotka času přiděleného vláknu. Víme také, že kvantum začíná a končí vždy při přerušení časovače. Systém NT však ve skutečnosti počítá čas kvanta tak, že tento neodpovídá celočíselně počtu přerušení časovače. Kvantum se totiž nepočítá přímo dle přerušení časovače, ale v jakýchsi abstraktních časových jednotkách. Celé kvantum má velikost 6 jednotek = 2 tiky (workstation) nebo 36 jednotek = 12 tiků (server). (Tik je doba mezi sousedními dvěma přerušeními časovače (je to konstanta), časová jednotka NT je tedy třetinou tiku a kvantum je 6 či 36 jednotek dle verze systému.)

Průvodce studiem

Workstation a server jsou dvě řady systému Windows. (Windows XP a Vista patří do řady workstation, Windows 2003 a 2008 je server a Windows 2000 má obě verze.) Výchozí délku kvanta lze však změnit v systémovém registru.

Důvodem šestinásobně větší délky kvanta na serverech je, že se předpokládá výskyt procesů či vláken, které během jednoho kvanta stihnou vykonat svůj úkol a skončit. Tím, že se čas přidělený vláknu nekouskuje na menší dílky, se zvyšuje propustnost systému a zkracují se časy odezvy. A to je u serveru jistě příjemné. Naopak na běžné pracovní stanici (workstation) takto dlouhá kvanta nejsou žádoucí.

Každé vlákno si pamatuje „své kvantum“, tj. v jedné své proměnné, i když neběží, si pamatuje, kolik časových jednotek má nebo bude mít k dispozici. Jakmile běží, tak tato hodnota ubývá až k nule a po skončení běhu se tato hodnota opět nastaví na 6 či 36.

Přerušení časovače odečte 3 jednoty z kvanta běžícího vlákna. Jakmile je kvantum vyčerpáno, plánovač vybere další vlákno k běhu. Nastane-li přerušení časovače v okamžiku obsluhy jiného přerušení, odčítá se tomu vláknu, které běželo před přerušením. (Pozn. Toto není chyba. Kdyby tomu tak nebylo, každé přerušení těsně před tikem hodin by způsobilo prodloužení reálného běhu vlákna. A to by mohlo ohrozit stabilitu systému, neboť vlákno by v nejhorším případě mohlo běžet až nekonečně dlouho.)

Když vlákno na něco čeká (např. voláním WaitForSingleObject), zmenší se po dokončení čekání počítadlo kvanta o 1 jednotku. (Opět nutné pro stabilitu systému, ze stejného důvodu jako v předchozím odstavci.) Vlákno po dokončení čekání tedy nedostává celé kvantum, ale má jen to, co mělo před čekáním -1. Vlákna s prioritou 14 a vyšší však po skončení čekání dostanou celé nové kvantum (tj. 6 či 36 jednotek).

Samotná délka tiku hodin závisí na HAL (hardware abstraction layer). A co to vlastně znamená? HAL je modul stojící mimo jádro NT a jsou v něm implementovány procedury závislé na konkrétním hardwaru. Zatímco systém NT není omezen na konkrétní typ počítače či procesoru a má vždy stejné systémové jádro, HAL obsahuje součásti na hardwaru závislé. A délku tiku hodin právě určuje HAL, nikoli jádro NT. U platformy x86 se uvádí délka tiku přibližně 10ms na jednoprocesorových a přibližně 15ms na víceprocesorových počítačích.

V systémovém registru Windows lze nastavit:

- Zda chceme krátká, či dlouhá kvanta (6 nebo 36 jednotek).
- Zda chceme upřednostnit vlákna procesu na popředí ($3\times$ delší kvantum), nebo ne. (Toto lze nastavit i v ovládacím panelu Systém – sekce Výkon, jako výběr mezi optimalizací pro „Programy“ nebo „Úlohy na pozadí“.)
- O kolik jednotek se má zvýšit priorita vlákna na popředí po dokončení čekání (může být 0, 1 nebo 2; viz níže).

Na neserverových počítačích má vždy jeden proces delší kvanta. Je to proces, kterému patří okno na popředí, a týká se to všech jeho vláken. Okno na popředí je to, které má klávesový fokus (tj. kam právě jde klávesový vstup) – všem vláknům tohoto procesu je prodlouženo kvantum na trojnásobek. Ve Windows NT před 4.0 se toto řešilo zvyšováním priority těchto vláken o 2 body. To však způsobovalo, že plánovač vybírat tato vlákna vždy přednostně a pokud program na popředí prováděl intenzivní výpočty, ostatní procesy v systému nedostávaly CPU téměř vůbec. Řešení s prodloužením kvanta zachovává férové plánování, kdy se na každého dostane, a zároveň dává víc času procesu na popředí, což zlepšuje uživatelskou odezvu.

6.2.4 Zvyšování priority

Na zvyšování priority (priority boosting) jsme již narazili výše, nyní si jej popišme podrobněji. Týká se jen dynamických úrovní (1–15) a vždy platí: current priority >= base priority, systém tedy prioritu jen zvyšuje a nikdy ji nesnižuje pod bázovou hodnotu.

Systém mění prioritu vlákna v těchto případech:

Po dokončení I/O operace. Připomeňme, že po dokončení každého čekání se sníží počítadlo kvanta o jedničku. Podle typu dokončené I/O operace se však také dočasně zvýší bázová priorita daného vlákna. O kolik přesně se zvýší, to si rozhoduje ovladač zařízení sám, doporučeno je:

+1 disk, CD-ROM, paralelní port, grafická karta

+2 síťová karta, pojmenovaná roura, sériový port

+6 klávesnice, myš

+8 zvuková karta

Po uplynutí jednoho kvanta se priorita sníží o 1 stupeň, postupně se takto snižuje až na původní bázovou prioritu. Mělo by přitom platit: Čím kratší čas ovladač potřebuje, tím dostane vyšší prioritu. Navíc, vlákna s vyšší prioritou mohou stále přerušit takto boostované vlákno, které před boostováním mělo prioritu nižší. Po znovuaktivaci však toto vlákno opět pokračuje boostované, priorita mu klesá jen po dokončení celého časového kvanta.

Po čekání na událost nebo semafor. Připomeňme, že po dokončení čekání se vláknu sníží počítadlo kvanta o jedničku. Bázová priorita se mu však na dobu jednoho kvanta zvýší o jedničku.

Při některých událostech, např. uvolnění kritické sekce, je aplikován ještě větší boost: Má-li čekající vlákno prioritu 13 nebo nižší, dostane prioritu o 1 vyšší, než mělo vlákno, které aktivovalo danou událost. Navíc má-li kvantum menší než 4 jednotky, zvýší se mu na 4. (Vlákno má malé kvantum, když při posledním běhu neběželo po celou délku kvanta.) Tento speciální „větší“ boost je ukončen po doběhnutí kvanta.

Když kterékoliv vlákno procesu na popředí dokončí čekací operaci. Aktuální priorita se zvýší +2 na dobu jednoho kvanta. Velikost tohoto zvýšení je jednou z hodnot nastavitelných v systémovém registru (+0, +1 nebo +2), Windows Server zde má nastavenu nulu.

Když se vlákno obsluhující GUI probudí kvůli aktivitě v okně. Toto je totéž jako v předchozím případě, ale týká se i procesů na pozadí. Aktivita v okně (GUI aktivita) je, když se něco děje a přijde zpráva do okna. Vlákno přitom může být na popředí a zároveň čekat na GUI, pak je aplikován tento i předchozí boost současně (obvykle tedy +4).

Když vlákno v Ready stavu už dlouho neběželo. Tato situace může nastat u vláken připravených k běhu, avšak majících malou prioritu. Každé vlákno, které neběží 300 tiků (což jsou 3 až 4 sekundy), dostane prioritu 15 a dvojnásobné kvantum. Po dokončení tohoto

dvojkvanta nebo při přerušení preempcí se priorita tomuto vláknu vrací rovnou zpět na původní úroveň. Dlouho neběžící vlákna se přitom kontrolují $1\times$ za sekundu.

6.2.5 Symetrický multiprocessing (SMP)

Jak víme, systém NT je mimo jiné navržen i pro symetrický multiprocessing. V praxi to znamená, že můžete mít v počítači více procesorů (nebo vícejádrový procesor) a používat stejný operační systém jako na jednoprocesorovém počítači.

> **Průvodce studiem**
>
> Symetrický multiprocessing podporovaly i poměrně dávné verze NT. Běžně však byly omezeny na 2 procesory, jinak bylo nutno zakoupit dražší verzi. Omezení počtu procesorů u NT přitom platí dodnes – kdo chce hodně procesorů, musí si připlatit za „vyšší“ verzi systému.

Důvodem, proč se u SMP musíme zastavit, je, že přítomnost více procesorů výrazným způsobem komplikuje plánování vláken. Zatímco na jednoprocesorových počítačích scheduler/dispatcher vždy jednoduše vybírá k běhu vlákno s nejvyšší prioritou, na víceprocesorových by tento jednoduchý model nebyl optimální. Nutno také upozornit, že ani skutečné řešení plánovače není zcela optimální, protože by to bylo algoritmicky příliš složité a tím by vlastně narostla režie systému natolik, že by výsledný optimální systém byl vlastně pomalejší. Víceprocesorové počítače tedy používají speciální scheduler/dispatcher, který je jakýmsi kompromisním řešením mezi matematicky optimálním víceprocesorovým plánováním a obyčejným jednoprocesorovým plánováním.

Maska afinity určuje seznam povolených procesorů pro vlákno.

Každý proces má *masku afinity* (affinity mask) – bitovou masku určující seznam povolených procesorů, na kterých může běžet. Každé vlákno má také affinity mask (při založení vlákna je tato hodnota okopírována z afinity procesu). Každé vlákno má dále dva identifikátory procesoru: *ideální procesor* a *minulý procesor*. Ideální procesor je procesor, kde vlákno „chce“ běžet. Tato hodnota je nastavena při vzniku vlákna tak, aby vlákna byla rovnoměrně rozdělena mezi procesory (čili cyklicky). Minulý procesor je procesor, na kterém vlákno posledně běželo. Tato hodnota je sledována proto, neboť může být lepší plánovat vlákno opakovaně na stejný procesor z důvodu lepšího využití cache (procesor si možná ještě pamatuje něco z minulého běhu v cache – díky tomu dosáhneme vyššího celkového výkonu počítače).

Výběr procesoru k běhu vlákna

Podívejme se nyní na algoritmus výběru procesoru k běhu vlákna. Tento algoritmus přichází ke slovu v okamžiku, kdy vlákno přechází do stavu ready, tehdy je totiž snaha, aby vlákno okamžitě přešlo do stavu running, či alespoň standby (má-li dostatečně vysokou prioritu, viz také kapitolu 6.2.3).

Přednost má nečinný procesor. Je-li jich víc, přednost má ideální procesor, potom minulý procesor, potom aktuální procesor (tj. ten, který provádí tento kód). Pokud jsou nečinné jen ostatní procesory, vybere se kterýkoliv z nich, pouze musí odpovídat nastavení affinity mask. Na Windows XP a novějších se zde také zohledňují hyperthreading a NUMA systémy. Hyperthreading je technologie dvou logických procesorů na jednom fyzickém, kde máme jen jedno plnohodnotné jádro CPU vykonávající kód, ale na něm jsou vytvořeny dva logické CPU a přepínání mezi nimi je rychlejší. Přednost při plánování dostávají ty procesory, kde jsou oba ve dvojici volné. NUMA (non–uniform memory access) jsou systémy, kde je nesourodá paměť a jednotlivé procesory nebo jejich skupiny mají své lokální paměti. V tom případě dostávají přednost ty volné procesory, které jsou ve stejné skupině jako procesor ideální.

Není-li žádný povolený procesor v masce afinity nečinný, systém se pokusí umístit vlákno preempcí. Nejprve zkusí ideální procesor, potom minulý procesor. U každého se snaží nahradit standby a running vlákno aktuálním. Má-li standby vlákno nižší prioritu, dojde k preempci. Má-li i running vlákno nižší prioritu, dojde k preempci. Z uvedeného plyne, že pokud ideální ani minulý procesor nemají běžící vlákno s nižší prioritou, aktuální vlákno není umístěno nikam a jde do čekací fronty. Zde tedy může nastat situace, že vlákno s vysokou prioritou čeká na procesor, zatímco na jiném procesoru běží jiné vlákno s nižší prioritou.

Průvodce studiem

Připomeňme, že celá tato mašinérie se týká případu, kdy máme vlákno a hledáme pro něj procesor, což je aktuální obvykle u vláken s vyšší prioritou. Vlákno, které nemá běžet, nepotřebuje ani přidělovat procesor. Vybraný procesor má pouze vlákno aktuálně běžící (tj. ve stavu running) a vlákno vybrané pro příští běh (tj. ve stavu standby) – v obou případech je to právě jedno vlákno.

Nejprve je snaha umístit vlákno na nějaký volný procesor dle pravidel popsaných výše. Pokud se to nepodaří, kontrolují se priority vláken ve stavech standby a running (a to pouze na procesorech popsaných výše). Je-li nalezeno vlákno s nižší prioritou ve stavu standby či running, dochází k preempci. (Čili nové vlákno je umístěno namísto dosavadního vlákna s nižší prioritou.)

Častější situaci, kdy vybíráme vlákno k běhu na uvolněném procesoru, popisuje následující sekce.

Výběr vlákna pro procesor

Nyní se zaměřme na situaci, kdy je procesor uvolněn a hledá se další vlákno k běhu. Tato situace nastává, když skončí časové kvantum, vlákno samo odevzdá procesor zpět systému, změní se priorita vlákna atp. Vybírá se pak nové vlákno pro tento konkrétní procesor a nehledí se na to, co se právě děje na procesorech ostatních. Další vlákno je vybíráno z fronty čekajících vláken s nejvyšší prioritou. (Je 32 čekacích front, pro každé číslo aktuální priority jedna. Zde pracujeme s jednou z nich – tou, která odpovídá vláknu či vláknům s nyní nejvyšší aktuální prioritou.) Vybráno je první z nich, které má procesor označen jako minulý či ideální, je ve stavu ready déle než 3 tiky hodin nebo má prioritu 24 či vyšší. Není-li žádný takový kandidát, vezme se první vlákno z této fronty. Přitom se ignorují vlákna, u kterých nevyhovuje maska afinity.

Jak je vidět z popisu algoritmu výběru, na multiprocesorech NT nezajišťuje, že poběží vlákno s nejvyšší prioritou. (Maska afinity má přednost před prioritou.)

Průvodce studiem

Běží-li tento algoritmus na jednoprocesorovém počítači, chová se přesně stejně jako dříve popsaný plánovací algoritmus NT. Prostudováním těchto odstavců tedy možná i lépe pochopíte plánování v jednoprocesorovém systému NT (zejména některé netriviální detaily).

Windows Server 2003 a novější

Systémy Windows Server 2003 a novější mají samostatnou frontu čekajících vláken pro každý procesor. Zrychluje to plánování, neboť na dřívějších systémech nemůže plánovací algoritmus běžet současně na více procesorech a při souběhu plánování tak musí některé procesory čekat. (Jelikož plánování je časováno časovačem, při přerušení časovače se běžně plánuje víc než jeden

CPU. Proto čím více je v počítači CPU, tím víc tam vadí starý plánovací algoritmus. Celkové časy strávené nečinným čekáním sice nejsou velké, ale již při 3 CPU dochází u starších verzí NT vždy ke kolizi při plánování. Plánuje se totiž 1 CPU při každém druhém přerušení, takže 3 CPU už určitě udělají kolizi.) Má-li WS2003 prázdnou frontu u daného procesoru, spustí na něm idle vlákno a to pak prohledává fronty čekajících vláken na jiných procesorech a najde-li vhodné, vezme si jej. Na NUMA systémech přitom dává přednost procesorům ve stejné skupině.

6.3 Vlákna v Linuxu

Na závěr kapitoly se ještě stručně zastavme u vláken systému Linux. Tento systém se navenek tváří jako Unix, neboť podporuje standardní rozhraní (API) POSIX. Interně systém podporuje vlákna a všechny je zveřejňuje jako samostatné procesy. Tento přístup je tedy na první pohled dosti odlišný od systému NT, avšak praktické chování a vlastnosti obou systémů jsou co do vláken stejné. (Oba systémy jsou z tohoto hlediska rovnocenné.)

Vlákno v Linuxu vytvoříme příkazem **clone**, který je zobecněním příkazu **fork**. Zatímco fork klonuje celý proces se vším všudy, u clone můžeme nastavit, které kontexty chceme klonovat, či sdílet s rodičovským procesem. Clone navíc spouští nový proces na libovolné funkci (zatímco v případě forku oba procesy pokračují na stejném místě kódu). Funkce **sys_clone** je kompromisem – chová se jako clone, ovšem oba procesy pokračují ve vykonávání kódu jako v případě forku.

Kontexty, které lze v novém procesu buď naklonovat, nebo nechat sdílet s rodičovským procesem, jsou především:

- Odkaz na rodičovský proces. Sdílíme-li jej, oba procesy mají stejného rodiče, klonujeme-li, volající proces je rodičem nového.
- Otevřené soubory. Při sdílení mají procesy stejnou tabulku otevřených souborů. Při klonování dostane nový proces kopii tabulky v okamžiku vytvoření, takže sice také dostane přístup ke všem otevřeným souborům, ale jen skrze kopii.
- Jmenný prostor souborového systému. Nastavení tohoto kontextu je vyhrazeno pro procesy s vyšším oprávněním; každý jmenný prostor v Linuxu určuje hierarchii připojených svazků souborového systému a obvykle není důvod na tom u jednotlivých procesů něco měnit.
- Tabulka signálů.
- Adresový prostor. Jeho sdílení je základním předpokladem, abychom mohli procesu říkat vlákno.

Nově vytvářený proces můžeme také vytvořit pozastavený. (Chová se stejně jako poslání signálu **SIGSTOP**; pozastavený proces začne (opět) běžet po signálu **SIGCONT**.) Můžeme také nastavit pozastavení volajícího procesu až do ukončení volaného (což se chová stejně jako funkce **exec**).

Flexibilita Linuxu jde tak daleko, že vlákna ve smyslu, jak je chápeme my, mohou v systému figurovat buď jako plnohodnotné procesy se samostatnými čísly (PID), které mají nastavenou sdílenou paměť (případně i další součásti), nebo skutečně jako vlákna, vystupující ve skupinách pod stejným číslem procesu (PID). (Ačkoliv vlákno je interně vždy na úrovni procesu, funkce operující s identifikátory mohou vracet zástupný identifikátor skupiny vláken (TGID).) Tyto jemné nuance nemusíme příliš podrobně zkoumat, mohou se však hodit např. pro dosažení 100% kompatibility s jinými operačními systémy (pro software vyvinutý původně pro jiný systém a běžící nyní na Linuxu).

U Linuxu se někdy uvádí, že kromě systémové podpory vláken lze vlákna používat také na úrovni knihovny. Zjištění podrobností nemusí být zrovna snadné, neboť literatura se této problematice obvykle nevěnuje s dostatečnou přesností. Ve skutečnosti se vlákna v Linuxu nejčastěji

používají jako v Unixu, tedy přes systémové rozhraní POSIX s vláknovou knihovnou Pthreads. Tato knihovna ve výchozím provedení implementuje vlákna na uživatelské úrovni, přepínání je tedy rychlejší, než u vláken v jádře systému, ale negativa tohoto řešení výrazně převládají. Systémová vlákna tak, jak byla popsána výše, nejsou kompatibilní s Pthreads a navíc mají ve většině distribucí Linuxu řadu skrytých nedostatků (např. existence řídicího vlákna vázaného na jeden fyzický procesor, přes které jde řízení vláken v procesu a také všechny synchronizační operace), které rovněž snižují výkon (stejně jako uživatelská vlákna v knihovně). Pouze některé nejnovější distribuce Linuxu implementují Pthreads pomocí systémových vláken; dostupné zdroje tuto variantu uvádějí jako jednoznačně lepší z hlediska výkonu. (Pro podrobnosti k tomuto tvrzení viz dokumentaci k nejnovějšímu jádru systému Linux 2.6 [Lov03].)

Shrnutí

V této kapitole jsme probrali správu procesoru v operačních systémech Unix, Windows NT a Linux. Teoretické pojmy zavedené v kapitole předchozí jsme tedy zasadili do kontextu skutečných operačních systémů. První část kapitoly byla věnována problematice vytvoření, plánování a stavů procesu v systému Unix. V další části jsme se věnovali stejné problematice v systému Windows NT a bylo ukázáno, že speciálně u vláken se řada věcí se v těchto dvou systémech vzájemně liší. Závěr kapitoly byl pak věnován problematice vláken v Linuxu, který k věci přistupuje ještě dalším jiným způsobem. Zatímco v NT jsou vlákna základem celého systému správy procesoru, v systémech Unix a Linux je základním prvkem proces, zatímco vlákna byla přidána až do pozdějších verzí.

Pojmy k zapamatování

- třída priorit
- sheduler/dispatcher
- (časové) kvantum
- HAL (hardware abstraction layer)
- zvyšování priority (priority boosting)
- ideální procesor
- minulý procesor
- maska afinity (affinity mask)
- příkazy exec, fork, clone, sys_clone
- příkazy CreateProcess, CreateThread

Kontrolní otázky

1. *Popište principiální rozdíl mezi chápáním vláken v systémech Unix a NT.*
2. *Popište, jak swapování ovlivňuje stavy procesů v Unixu.*
3. *Popište postup vytvoření nového procesu v Unixu.*
4. *Vyjmenujte situace, při kterých vlákno v Unixu opustí stav Running.*
5. *Jak se vytváří nový proces v NT?*
6. *Jak a kdy může nastavení afinity pomoci zrychlit vícevláknový program?*
7. *Co dělá „zero page thread“, jakou má prioritu a proč?*
8. *Jak se liší plánování procesoru na desktopové a serverové verzi Windows? Jaké mají tyto rozdíly smysl či cíl?*
9. *Vyjmenujte situace, při kterých se zvyšuje priorita vlákna v NT. U každé situace popište, proč je v ní zvýšení priority výhodné či přínosné.*
10. *Proč je plánování vláken u víceprocesorových počítačů komplikovanější, než u jednoprocesorových? Uveďte, co je zdrojem problémů.*
11. *Vysvětlete algoritmus výběru procesoru pro vlákno v NT.*
12. *Vysvětlete algoritmus výběru vlákna pro běh na procesoru v NT.*
13. *Popište situaci vláken v Linuxu. Jak jsou do systému zanesena a jaké to má důsledky?*

7 Synchronizace vláken a procesů

Studijní cíle: Tato kapitola je pokračováním bloku o procesech a vláknech. Tentokrát se seznámíme se synchronizačními objekty operačního systému, které slouží k synchronizaci vláken a procesů. Kromě synchronizace se zmíníme také o komunikaci mezi vlákny a procesy, což je příbuzné téma. U praktických částí se opět budeme věnovat především systému Windows NT.

Klíčová slova: synchronizace vláken, synchronizace procesů, semafor, mutex, kritická sekce

Potřebný čas: 120 minut.

7.1 Principy synchronizace

7.1.1 Úvod

Procesy komunikují zasíláním zpráv.

Samostatné (nezávislé) procesy spolu obvykle komunikují pomocí zasílání zpráv. Tento model je obecně použitelný jak pro procesy běžící na jednom počítači, tak v prostředí počítačové sítě. Problematika komunikace zasíláním zpráv je však mimo oblast studia operačních systémů (alespoň pro nás).

Vlákna jednoho procesu mohou mezi sebou rovněž komunikovat zasíláním zpráv, tento druh mezivláknové komunikace se však v praxi téměř nevyskytuje. Díky sdílené paměti totiž máme mnohem zajímavější možnosti, jak vlákna nechat komunikovat či přímo spolupracovat. Díky sdílené paměti často stačí místo nějaké formy komunikace použít sdílená data (např. ve formě globální proměnné, může ale jít i o ne–globální data, pokud o nich všechna zainteresovaná vlákna vědí). Práce se sdílenou pamětí však přináší jednu velkou starost navíc: Jelikož vlákna spolu sdílejí veškerou operační paměť, mohou ji libovolně číst i zapisovat. Jednotlivá vlákna přitom nevědí, co dělají ta ostatní, a to může mít na datech katastrofální následky. Nejčastější problémy při sdílení paměti můžeme neformálně popsat takto: Zatímco jedno vlákno zapisuje novou hodnotu, jiná vlákna mohou pracovat se starou hodnotou a později také chtít zapsat svou novou hodnotu. Tím se ztratí hodnota prvně zapsaná. Druhým typickým případem je práce se složitějšími daty, která patří k sobě. Jakmile jedno vlákno začne aktualizovat (zapisovat novou hodnotu), dělá to postupně, čili zapisuje jednotlivé proměnné, které tvoří větší celek. Ostatní vlákna v ten okamžik ale mohou číst hodnoty a získají tím částečně nové a částečně staré hodnoty, tedy jako celek neplatné. Tyto problémy mohou mít za následek chybný výpočet programu, nebo dokonce jeho zhroucení či uváznutí v mrtvém bodě.

Vlákna mezi sebou musíme synchronizovat.

Základním nástrojem pro spolehlivé fungování vícevláknových programů je mezivláknová (či meziprocesní) synchronizace. Synchronizaci realizujeme pomocí tzv. synchronizačních primitiv (základních prvků), které poskytuje operační systém. Mezi základní patří jmenovitě např. *událost*, *semafor*, *mutex* či *kritická sekce*.

Průvodce studiem

Otázka toho, které synchronizační primitivum je skutečně „základní“, je čistě věcí přístupu. Můžeme se dohodnout, že např. semafor budeme považovat za nejzákladnější prvek synchronizace, neboť se s ním dobře pracuje a ostatní primitiva pomocí něj lze naimplementovat také. Jeho fyzická implementace v systému je však o něco složitější než třeba u události. Událost tedy můžeme považovat za základní primitivum právě proto, že jde o velmi jednoduchý prvek a jeho realizace je možná jen s několika málo instrukcemi procesoru.

U objektově orientovaných systémů je pak za základní synchronizační primitivum obvykle považován jedině *monitor*.

Realizace synchronizačních primitiv je většinou možná jen spoluprací systému a hardwaru a samotné uživatelské programy by ji samy realizovat nedokázaly. Fakt, které primitivum je interně v systému realizováno pomocí kterého, není pro uživatele–programátora vůbec důležitý. Podstatný je jen význam a funkčnost primitiv, ten je (nebo by alespoň měl být) u všech operačních systémů stejný (až na drobné nuance).

7.1.2 Atomické operace

Společným rysem synchronizačních primitiv je, že mají schopnost provést nepřerušitelně více jednoduchých operací v řadě. Operace, která sice může být složena z více dílčích kroků, ale je nepřerušitelná, se nazývá *atomická*. Bez ohledu na počet procesorů a vláken, atomická operace vždy proběhne celá v jeden okamžik bez možnosti kolize s jiným vláknem. Tuto vlastnost (atomicitu) si ukážeme na několika příkladech chyb, které mohou nastat při souběhu dvou vláken.

Atomická operace je nepřerušitelná.

První příklad: Práce se zásobníkem

Jak víme, zásobník je jednoduchá dynamická datová struktura, do které můžeme ukládat data (operace push) a potom je opět vyzvednout (operace pop), přičemž platí princip LIFO „co jde poslední dovnitř, to jde první ven“. Představme si zásobník implementovaný pomocí pole, čili stejně jako systémový programový zásobník v počítači, a podívejme se na operaci push.

Přidání prvku na zásobník provedeme ve dvou jednoduchých krocích:
1. Posuneme ukazatel vrcholu o jednu buňku níže.
2. Zapíšeme na pozici vrcholu nová data.

Tento popis plně odpovídá jak zmíněnému programovému zásobníku v počítači, tak jakémukoliv zásobníku implementovanému jednoduchým polem.

Při souběhu dvou vláken, tedy při současné práci dvou vláken s jedním zásobníkem, může nastat situace, kdy se obě vlákna snaží současně vložit nový prvek na zásobník. V této situaci tedy každé z vláken provede oba výše uvedené kroky, přičemž chyba nastane kdykoliv, kdy některé vlákno neprovede oba své kroky dohromady, ale vloží se mezi ně vlákno druhé. Označíme-li si vlákna A a B, pak při pořadí AABB nebo BBAA k chybě nedojde, ale při pořadí ABBA, ABAB, BAAB nebo BABA dojde k chybě – výsledný stav zásobníku nebude správný, protože položka pod vrcholem bude mít náhodnou hodnotu a hodnota, která tam měla být uložena vláknem, bude buď ztracena, nebo bude nesprávně uložena na vrcholu (a položka patřící na vrchol bude ztracena).

Korektním řešením operace push je provedení obou dílčích kroků atomicky. Tj. zaručíme, že mezi první a druhý dílčí krok se nedostane jiné vlákno provádějící také operaci push na stejném zásobníku. (Tento postulát však nevylučuje další korektní řešení daného problému.)

Druhý příklad: Inkrementace proměnné

Jako druhý příklad si uveďme jednoduchou operaci inkrementace proměnné (čili zvýšit hodnotu o jedna). Tato operace je provedena pomocí jediné instrukce procesoru, takže by se mohlo zdát, že automaticky atomickou operaci a není třeba ji dále diskutovat. Opak je však pravdou.

Skutečně, při souběhu vláken může inkrementace způsobit chybu na víceprocesorových počítačích. Zatímco procesor nelze přerušit uprostřed vykonávání jednotlivé instrukce, běží-li současně více procesorů tak mohou pracovat s pamětí libovolně a nejde o přerušení. Konkrétně inkrementace proměnné je operace složená ze tří kroků:
1. Procesor přečte proměnnou.

2. Procesor inkrementuje hodnotu.
3. Procesor zapíše hodnotu.

Ačkoliv na první pohled může vypadat podivně, proč je tak jednoduchá operace rozdělena do tří kroků, při drobném zamyšlení by mělo být jasné, že operační paměť podporuje jen operace čti a zapiš a k inkrementaci hodnoty musíme nutně znát hodnotu inkrementované proměnné.

Tento příklad nám tedy ukázal, že na víceprocesorových systémech mohou „nefungovat“ i ty programy, které na jednoprocesorových fungovaly bezvadně. (A to doslova – tento typ chyby skutečně na jednoprocesorovém počítači nemůže nastat, nelze jej tedy ani vysledovat a odladit.)

7.1.3 Hardwarové atomické operace

Na jednoprocesorových systémech dosáhneme atomické operace tak, že po dobu vykonávání kritického kódu zakážeme přerušení. Tento postup je jednoduchý, spolehlivý a funkční v zásadě na všech procesorech, které nemusejí pracovat v reálném čase. Je přitom samozřejmě čistě na naší libovůli, jak velkou část kódu provedeme takto „atomicky“. Řada operačních systémů však nedovoluje, aby zákazy přerušení prováděly přímo uživatelské procesy, neboť se zakázaným přerušením je zablokován také scheduler a nemůže probíhat přepínání vláken. Řešení je zde ještě poměrně snadné – operační systém obsahuje sadu základních atomických operací, takže kód zákazu přerušení je pouze v jádru systému a uživatelské procesy přerušení zakázat nemohou. Potřebuje-li proces vykonat delší kus kódu atomicky, musí vhodným způsobem použít atomické operace systému (což si představíme později – viz povídání o semaforech, mutextech atp.).

> **Průvodce studiem**
>
> V některých zdrojích se můžeme setkat s informací, že pro zajištění atomicity stačí, aby v daný okamžik dané vlákno mělo nejvyšší prioritu. Tento postup však není bezpečný, neboť i vlákno s nejvyšší prioritou může být přerušeno. (S odkazem na popis plánovače úloh v kapitole 5.)

U víceprocesorových systémů je situace ještě složitější. Zákaz přerušení totiž ovlivňuje jen chování jednoho procesoru, navíc jednotlivé procesory mohou do paměti přistupovat tak, že cizí procesor se dostane k paměti i během provádění jediné jednoduché instrukce (viz příklad inkrementace hodnoty na začátku této kapitoly – jedná se o skoro nejjednodušší instrukci, a přesto není atomická). Procesory navržené pro víceprocesorové systémy obvykle poskytují nějaké speciální atomické instrukce, základní teoretické modely si nyní představíme.

Test–and–set

Test–and–set je základní model atomická operace.

V literatuře je nejčastěji prezentován model *test–and–set* (TAS – otestuj a nastav). Je-li operace TAS implementována jako atomická instrukce procesoru, může být základem mezivláknové synchronizace.

Instrukce typu TAS atomicky testuje hodnotu v paměti a nastaví ji na novou hodnotu. Po provedení této instrukce máme k dispozici původní hodnotu z paměti a v paměti je nová hodnota. TAS může být implementováno nejen samotným procesorem, ale také jiným hardwarovým obvodem, typicky třeba víceportovým paměťovým modulem (ten umožňuje více přístupů současně a pro synchronizaci podporuje operaci TAS).

Compare–and–swap

Compare–and–swap je v praxi nejčastěji používaný model atomické operace.

Ačkoliv teoretická literatura lpí na modelu TAS, v praxi se velmi rozšířil také model *compare–and–swap* (CAS – porovnej a vyměň). Je vědecky dokázáno, že tento model je nadřazený modelu TAS (existují úlohy, které pomocí CAS lze řešit efektivněji) [Her91].

Instrukce typu CAS porovná hodnotu v paměti s jinou hodnotou a pokud jsou si rovny, hodnotu v paměti vymění s jinou (třetí) hodnotou. K dispozici pak máme výsledek testu rovnosti a v případě rovnosti také původní hodnotu z paměti. Totéž lze popsat i jinými slovy: Instrukce typu CAS má tři operandy a je atomická. Nejprve porovná první s druhým. Jsou-li rovny, okopíruje třetí do druhého a vrací true. Nejsou-li rovny, okopíruje druhý do prvního a vrací false. Na procesorech x86 je CAS implementováno instrukcemi CMPXCHG a CMPXCHG64B (32bitová a 64bitová verze). Na víceprocesorových systémech přidáváme ještě prefix LOCK, který blokuje přístup na sběrnici ostatním procesorům.

Fetch–and–add

Další běžnou atomickou operací je *fetch–and–add*, která atomicky přečte hodnotu proměnné a přičte k ní jinou hodnotu. Vrací přitom původní hodnotu. Na procesorech x86 je toto implementováno instrukcí XADD, opět v kombinaci s LOCK u víceprocesorových systémů. Dodejme ještě, že operace fetch–and–add je rovněž funkčně podřízená CAS, stejně jako TAS.

7.1.4 Softwarová implementace atomicity

Jako příklad čistě softwarového řešení atomicity uveďme Petersonův algoritmus [Pet81]. Jde o řešení pro dvě vlákna, ovšem je možno jej zobecnit i na více vláken. [Hof90]

> **Průvodce studiem**
>
> Otázky synchronizace se začaly řešit již v 50.letech 20.století. Zatímco dnes má každý běžný procesor několik chytrých atomických instrukcí, pomocí nichž lze synchronizaci snadno zvládnout, tehdejší hardware samozřejmě úrovně dnešních mikroprocesorů nedosahoval. Jedním z elegantních softwarových řešení je právě tento Petersonův algoritmus, který se ale objevil až v roce 1981. Některé zdroje uvádějí i starší algoritmy, jako např. Dekkerův z roku 1965, je však otázkou, nakolik byly tyto algoritmy všeobecně známy v době svého vzniku.

kód 1. procesu:

```
lockA = true;
turn = B;
while(lockB && turn!=A) { }
...
lockA = false;
```

kód 2. procesu:

```
lockB = true;
turn = A;
while(lockA && turn!=B) { }
...
lockB = false;
```

V místě tři teček doplníme libovolný kód, který má být atomický. (Není podstatné, zda se tento kód u jednotlivých vláken liší, nebo je stejný.) Tento kód ve skutečnosti není opravdu atomický, neboť algoritmus jen zajišťuje, že jakmile se začne vykonávat kód na místě tří teček v jednom procesu, pak nebude přerušen kódem na místě tří teček ve druhém procesu. Jedná se tedy o mírnější podmínku než u atomicity a říkáme ji *vzájemné vyloučení*. Jinými slovy lze říci, že Petersonův algoritmus řeší atomicitu jen pro speciální případ, kdy jsou v systému jen dva procesy. Nelze jej tedy použít pro více procesů či pro vlákna sdílející tentýž kód; takové zobecnění by bylo možné, avšak značně složité a z praktického hlediska nezajímavé. Proto se zobecňováním základního algoritmu zabývat nebudeme.

V následujících sekcích si některá základní synchronizační primitiva představíme podrobněji. Všechna se chovají jako objekty – je to tedy vždy nějaká datová struktura obsahující několik operací, které s ní lze dělat. Tyto operace jsou obvykle atomické a jsou tím hlavním, co nás bude zajímat. Naopak informace o tom, jaká data jsou vlastně skryta uvnitř jednotlivých synchronizačních objektů, pro nás vůbec podstatná (ani zajímavá) není.

7.1.5 Semafor

Semafor řídí přístup k prostředku.

Semafor je chráněná proměnná a je základním nástrojem pro řízení (a omezení) přístupu ke sdíleným prostředkům.

Průvodce studiem

Předchozí věta prakticky zcela popisuje semafor. Co jsme se tedy dověděli: Semafor je chráněná proměnná, tj. představuje proměnnou, ke které je chráněný přístup pomocí několika operací semaforu (které si představíme níže). Proměnná je chráněná, tzn. pomocí operací, které jsou k dispozici, je zajištěno, že její stav je vždy korektní.

Dále jsme se dověděli, že semafor slouží k řízení přístupu ke sdíleným prostředkům. Hodí se tedy všude tam, kde máme nějaké prostředky, které lze sdílet, ale jejich množství je omezeno (často na pouhý jeden přístup, ale může být povoleno i více přístupů současně).

Uvnitř semaforu je skryta celočíselná proměnná, říkáme ji *hodnota semaforu*, nebo jednoduše počítadlo. Tato hodnota je inicializována na počet prostředků, které jsou na začátku k dispozici. Potom máme k dispozici dvě atomické operace, nazýváme je P a V.

Operace P čeká, než je k dispozici aspoň jeden prostředek (počítadlo > 0). Potom sníží hodnotu počítadla o jedničku.

Operace V oznamuje vrácení prostředku, čili zvýší hodnotu počítadla o jedničku.

V praxi tedy dochází k tomu, že v operaci P se vlákno při nedostatku prostředků hlídaných semaforem zastaví a čeká, až bude prostředek k dispozici. Díky tomu, že tato operace je atomická (a čekání je pasivní), jedná se o efektivní způsob řízení přístupu ke sdíleným prostředkům. A v neposlední řadě také jednoduše použitelný. Dodejme ještě, že operaci P může vykonat i jiné vlákno než operaci V. Je tedy možné, a v praxi se to skutečně často používá(!), že jedno vlákno vykonává operaci P, čímž získá prostředek, ale zcela jiné vlákno vykonává operaci V, čímž prostředek uvolňuje. Jedná se pak vlastně o jakési vytváření prostředků jedním vláknem a jejich konzumaci jiným.

Průvodce studiem

Nicneříkající P a V jsou všeobecně přijaté standardní názvy atomických operací semaforu, se kterými se pracuje v teoretické literatuře. V jednotlivých operačních systémech jim obvykle odpovídají jinak (lidsky) pojmenované funkce, např. **down** či **wait** pro P a **up** či **signal** pro V. Příklad programování se semaforem si ukážeme později. Některé implementace mohou poskytovat navíc i další operace, typická je například možnost zjistit hodnotu vnitřního počítadla.

7.1.6 Mutex a kritická sekce

Mutex a kritická sekce jsou dva úzce související pojmy. Mutex je zkratkou z anglického „mutual exclusion“ (vzájemné vyloučení) a kritická sekce je nástrojem k jeho dosažení. Synchronizační objekty mutexu a kritické sekce jsou de facto totožné, jedná se o synonyma. V praxi se však někdy vyskytují i jako dva ne zcela stejné prvky. (Rozdíl mezi nimi ve Windows si popíšeme níže v samostatné sekci.)

Mutex chrání část kódu před souběhem.

Smyslem vzájemného vyloučení je, že pouze jedno vlákno v dané chvíli vykonává chráněný kód. Mutex je tedy obdobou semaforu, jenže tentokrát nejde o chráněnou proměnnou, nýbrž část kódu.

Průvodce studiem

Už z úvodního popisu mutexu je zřejmé, že pomocí semaforu lze implementovat mutex a obráceně. Že všechny synchronizační objekty jsou nahraditelné, jsme si ostatně „prozradili“ už v úvodu kapitoly.

Mutex je obvykle implementován hardwarově, existují však i softwarová řešení. Zatímco semafor je obvykle nakonec implementován pomocí mutexu, pro mutex mají moderní procesory speciální instrukci. Je totiž zjevné, že umíme-li provést část kódu s vyloučením běhu ostatní vláken, jde o atomickou operaci, čímž máme vyřešen problém semaforu. Samotné vzájemné vyloučení se implementuje hardwarově pomocí instrukcí provádějící operace typu „test–and–set“ nebo „compare–and-swap“.

Mutex má dvě atomické operace: Vstoupit do kritické sekce, opustit kritickou sekci. (Na rozdíl od semaforu se zde nepoužívají žádné jednopísmenné názvy). Jedna kritická sekce může mít v kódu i více vstupů a výstupů. Znamená to pak, že všechny označené části kódu patří do společné sekce a platí pro ně vzájemné vyloučení. Na začátku kritické sekce každé příchozí vlákno musí vykonat operaci vstupu. V tom okamžiku systém testuje, zda v kritické sekci už nějaké jiné vlákno není. Pokud je volno, vlákno má povolen vstup, v opačném případě je uspáno a čeká na uvolnění sekce.

Na rozdíl od semaforu zde neexistuje počítadlo umožňující více než jeden přístup v daný čas. (Bylo-li by více přístupů současně, nebylo by to už „mutually exclusive“.) Vlákno, které již je v kritické sekci, však může znovu projít přes vstup do sekce a tento proběhne okamžitě. Vlákno tedy může libovolně krát vstoupit do téže sekce. Na konci musí vlákno opustit kritickou sekci tolikrát, kolikrát do ní vstoupilo. Opustit sekci musí vždy přesně to vlákno, které do ni vstoupilo, ne nějaké jiné. (Zde vidíme zásadní rozdíl oproti semaforu.)

Průvodce studiem

Předchozí odstavec popisuje velmi důležitou vlastnost mutexu: Vlákno může opakovaně vstoupit do kritické sekce. Požadováno pouze je, aby počet vstupů a opuštění kritické sekce byl stejný a aby vše proběhlo ve stejném vlákně.

7.1.7 Událost a signál

Událost a signál jsou opět dva termíny pro totéž. Vyskytují se v programování počítačů v mnoha podobách; my se nyní zaměříme speciálně na signály související se synchronizací vláken a procesů.

V Unixu (a Linuxu) jsou signály základním nástrojem meziprocesní komunikace. Díky dominantnímu postavení jazyka C se s touto formou signálů setkáme i v ostatních systémech, i když jejich význam tam již nemusí být tak velký jako v Unixu. V teoretické rovině odpovídají události softwarovým přerušením a signály jejich handlerům. Operační systém tedy definuje sadu (několika málo desítek) událostí, které mohou nastat, a jednotlivé procesy mohou těmto událostem přiřadit obslužné funkce (handleru). V okamžiku, kdy nastane daná událost, systém „vyšle signál“ neboli zavolá handler – toto proběhne stejným způsobem, jak jsme si popsali u fyzických přerušení v kapitole 3.3. Pokud proces signálu nepřiřadí žádný handler, je volán výchozí handler. U některých signálů je toto samozřejmě žádoucí (např. signál výpadku paměťové stránky – viz kap. 9 na straně 84).

Ve Windows najdeme pod pojmem událost zcela jiný synchronizační prvek. Jak si podrobněji popíšeme dále v samostatné sekci, Windows umožňuje procesům vytvářet vlastní události

a používat je k synchronizaci. Nejsme tedy omezeni na několik pevně daných událostí, ale vytvoříme si libovolně vlastní. Naproti tomu tyto události nejsou navázány na chování systému, signály tedy musíme posílat sami. Dalším rozdílem je, že Windows nepoužívá pro aplikační programy přerušení, takže nelze jednoduše nastavit funkci, která se má zavolat při výskytu události. Ačkoliv to může působit jako nepříjemné omezení, pomocí událostí jsou ve Windows implementovány ostatní synchronizační objekty (včetně semaforu a mutexu).

7.1.8 Monitor

Monitor je prvek objektově orientované synchronizace.

Monitor je synchronizační prvek, který se přímo v Unixu či Windows NT neobjevuje. Najdeme jej však v moderních objektově orientovaných systémech, jmenovitě platformy Java a .NET jej deklarují jako svůj základní synchronizační prvek.

Monitory využívají objektově orientovaný princip, díky čemuž je programování s nimi poměrně snadné. Složitější však může být pochopit jejich princip. Popis monitorů se v různých publikacích liší, právě podle toho, nakolik se autoři snažili o podání popisu přesně, nebo naopak srozumitelně. My si monitory popíšeme z hlediska platformy .NET.

Připomeňme, že cílem či úkolem monitoru je zajistit, aby při souběhu vláken nedocházelo k chybám, jejichž příklady byly uvedeny na začátku této kapitoly. Jelikož pracujeme objektově, ochranu dat můžeme zajistit tak, že zaručíme, že s daným objektem pracuje v jeden okamžik jen jedno vlákno. Jak víme, s daty objektu mohou pracovat jen metody tohoto objektu. Základní synchronizaci tedy lze provést tak, že uzavřeme kód všech metod do jedné kritické sekce. Monitor tento princip realizuje a ještě rozšiřuje. Objektově orientovaný systém automaticky poskytuje monitor ke každému objektu. Využijeme-li jej, máme k dispozici zámek, kterým lze snadno zamykat přístup k objektu. Celé metody či části kódu, které mění stav dat v objektu, které chceme hlídat, označíme k zamknutí. Jedná se tedy o syntakticky jednodušší variantu kritické sekce, neboť se nemusíme starat o vytvoření a zrušení kritické sekce a vstup a výstup je připojen přímo k objektu (pomocí odkazu sám na sebe (this) – každý objekt má přece svůj monitor). Výhodou monitoru tedy také je, že těžko uděláme chybu – neuvádíme nikde odkazy na kritickou sekci, se kterou pracujeme, používáme jen odkazy typu this.

Zatím jsme tedy popsali, jak monitor nahrazuje kritickou sekci s tím, že má jednodušší zápis a menší riziko lidské chyby. Monitor má však i další důležitou schopnost, kterou již kritickou sekci převyšuje. Opustit zámek monitoru totiž lze nejen opuštěním monitoru (na konci hlídané části kódu), ale také čekáním na událost spřaženou s monitorem. Díky této atomické dvojoperaci lze atomicky uvolnit zámek monitoru a zároveň začít čekat na událost. Signál o události přitom pošle jiné vlákno v tomtéž monitoru. Pro příklad použití tohoto konstruktu uveďme řešení problému dodavatel–odběratel, který jsme si představili v části popisující semafory: Jedno vlákno vytváří produkt a druhé na něj čeká. Toto se může (ale nemusí) opakovat. Vlákno–konzument vstoupí do monitoru a čeká na událost. Vlákno–producent vstoupí do monitoru a signalizuje událost. Díky monitoru může existovat i několik producentů a konzumentů a systém vždy spolehlivě funguje (produkt se dostane vždy k právě jednomu konzumentovi).

Pomocí monitorů lze řešit všechny klasické synchronizační problémy. Na závěr dodejme, že některé klasické učebnice paralelního programování používají pro výuku synchronizace konstrukty mutex a condition variable (tzv. podmíněná proměnná), jejichž kombinací lze až na složitější syntaxi řešit paralelní úlohy velmi podobným způsobem jako s monitory.

Průvodce studiem

Hovoříme-li o paralelních „úlohách“, je to nenápadný odkaz na to, že problémy synchronizace spolupracujících vláken, se kterými se v praxi člověk může setkat, lze většinou převést na některou z několika málo klasických paralelních úloh. Diskuze algoritmů pro řešení těchto úloh je nad rámec kurzu Operační systémy, implementace známých algoritmů

pomocí příkazů operačního systému je však jedním z témat, se kterými se studenti setkají ve cvičení.

7.2 Synchronizace v NT

Podívejme se nyní blíže na synchronizační primitiva, která nabízí Windows NT. Hovoříme o tzv. „objektech“, protože jde několik skupin funkcí pracující s identifikátorem příslušné entity – jde tedy de facto o objekty.

V systému NT jsou všechny synchronizační objekty odvozeny od společného základu – jde o objekty, na které lze čekat (čekatelný objekt – waitable object). Jelikož systém není nativně explicitně objektový, některé systémové funkce jsou platné pro různé typy synchronizačních objektů. Z objektově orientovaného hlediska to odpovídá situaci, kdy každý jednotlivý synchronizační objekt, který můžeme používat, dědí z abstraktního bázového typu „waitable object/class“.

7.2.1 Událost

Jak už bylo zmíněno v předchozí sekci, základním (a nejjednodušším) synchronizačním objektem v NT je událost. Událost je objekt, na který lze čekat. Čekající vlákno neprovádí žádný jiný kód. Čekání končí buď neúspěšně po určitém čase (timeout), nebo v okamžiku, kdy jiné vlákno signalizuje událost.

Při práci s událostí máme k dispozici několik nastavení, kterými můžeme její chování ovlivnit. Funkcí *CreateEvent* můžeme vytvořit událost buď typu „auto reset“, nebo „manual reset“. Dále zde určíme výchozí stav události (signalizována/nesignalizována). Událost lze při vytváření pojmenovat, pak ji lze sdílet mezi procesy. Uvedeme-li jméno, musí toto být jednoznačné v rámci celého systému (jedné uživatelské session), podle kterého ho lze najít a otevřít v jiném procesu voláním *OpenEvent*. Obvyklejší je však použití nepojmenovaných událostí, ty lze sdílet jen mezi vlákny jednoho procesu. Vytvořením události získáme její identifikátor (tzv. „handle“). Po skončení práce s událostí zrušíme handle voláním *CloseHandle*. Událost zanikne po zrušení posledního handlu (nepojmenovaná nejpozději při skončení procesu).

Vlákno může na událost čekat kteroukoliv čekací funkcí, např. *WaitForSingleObject* (čeká na jeden objekt) nebo *WaitForMultipleObjects* (čeká na více objektů, nebo jeden z mnoha). Událost musíme signalizovat jiným vláknem pomocí funkce *SetEvent*. Událost typu manual reset je pak signalizována tak dlouho, než někdo zavolá *ResetEvent*. Přitom jsou probuzena všechna čekající vlákna. Událost typu auto reset je signalizována jen do doby, než probudí jedno vlákno. Čeká-li více vláken, jedno z nich (nelze říci, které) je probuzeno.

7.2.2 Semafor

Semafory v NT fungují tak, jak byly popsány v sekci 7.1.5. Nový semafor vytvoříme funkcí *CreateSemaphore*, uvedeme přitom celkový počet hlídaných prostředků a výchozí počet volných. Můžeme také semafor pojmenovat, význam pojmenování je stejný jako u události (otevřeme jej voláním *OpenSemaphore*). Zrušení semaforu je stejné jako u události.

Získání prostředku semaforu provedeme pomocí kterékoliv čekací funkce (stejně jako u události). Čekáme-li na více semaforů, počítadlo se snižuje u každého. Čekáme-li vícekrát na tentýž, počítadlo se sníží jen jedenkrát. Pro uvolnění prostředku pak voláme *ReleaseSemaphore*; tato funkce také umožňuje zjistit stav vnitřního počítadla semaforu.

7.2.3 Mutex

Mutexy v NT fungují tak, jak byly popsány v sekci 7.1.6. Nový mutex vytvoříme funkcí *CreateMutex*, lze také nastavit, zda do něj vytvářející vlákno má přímo vstoupit. Pojmenování a zrušení mutexu funguje stejně jako u události a semaforu.

Vstup do mutexu provádíme opět pomocí kterékoliv čekací funkce. Uvolnění mutexu provedeme voláním *ReleaseMutex*. Skončí-li vlákno bez uvolnění mutexu, je tento ve stavu „abandoned“. Jiné vlákno pak může mutex získat standardním způsobem, čekací funkce ale vrací příznak chyby abandoned. S mutexem lze dále pracovat a po dalším uvolnění mutexu tento ztrácí příznak chyby. (Mutex s tímto příznakem však značí, že někde v kódu došlo k chybě a chráněná data nemusejí být v konzistentním stavu.)

7.2.4 Kritická sekce

Kritická sekce je v NT zvláštním druhem mutexu. Kritické sekce mají vlastní sadu funkcí a nelze je pojmenovat, čili je nelze sdílet mezi procesy. Jsou tedy rychlejší než mutexy, ale nelze je kombinovat s jinými objekty, protože nepoužívají ani standardní čekací funkce. Jinak se neliší od popisu v sekci 7.1.6.

Kritickou sekci vytvoříme voláním *InitializeCriticalSection*, vstoupíme do ní voláním *EnterCriticalSection* nebo *TryEnterCriticalSection* a opustíme ji voláním *LeaveCriticalSection*. Objekt kritické sekce zrušíme voláním *DeleteCriticalSection*. Žádná z těchto funkcí nemá nastavitelné parametry (kromě adresy objektu kritické sekce).

Průvodce studiem

Pokud můžete, dejte ve vašich NT programech kritické sekci přednost před mutexem.

7.2.5 Čekací funkce

Čekací funkce čeká na čekatelný objekt.

V předchozích odstavcích jsme několikrát narazili na tzv. čekací funkce, které jsou platné pro každý čekatelný objekt (waitable object). Kromě *WaitForSingleObject* a *WaitForMultipleObjects*, které jsme již zmínili výše. Pomocí těchto funkcí lze čekat na jeden objekt, jeden z mnoha nebo více objektů. Lze čekat nekonečně dlouho, nebo po zvolenou maximální dobu (do timeoutu). Další sada čekacích funkcí umožňuje čekat kombinovaně na čekatelné objekty, události posílané do oken, události dokončení I/O operace a asynchronní volání procedur. Podrobnosti lze nalézt v [MSDN]. Alternativou ke kombinovanému čekání může být také použití více vláken, kde každé bude čekat na něco jiného.

Důležitou funkcí je také *SignalObjectAndWait*, která atomicky signalizuje jeden objekt a čeká na jiný. Použití této funkce je de facto ekvivalentní s tím, když v monitoru čekáme na spřažený signál. Signalizovat lze semafor, mutex nebo událost; čekat lze na celou řadu čekatelných objektů.

Průvodce studiem

Při synchronizaci vláken ve Windows NT se *SignalObjectAndWait* používá velmi často, protože je to jediná funkce, která umí atomicky uvolnit objekt a čekat an jiný. Dosáhnout stejného efektu pouze bez této funkce je poměrně složité (i když možné). Zajímavé přitom je, že tato funkce je k dispozici až od Windows 2000.

7.2.6 Další synchronizační a komunikační nástroje

Dalším synchronizačním objektem je *čekatelný časovač* (waitable timer). Je to jeden z mnoha druhů časovačů v NT, který je specifický tím, že na uplynutí měřeného úseku času lze čekat pomocí čekacích funkcí. Jelikož NT má i jiné druhy časovačů, tento se v praxi používá jen výjimečně, když se nám hodí kombinovat jej s jinými čekatelnými objekty. Čekatelný časovač lze také napojit na APC (asynchronní volání procedury), což je rovněž poměrně málo používaná funkcionalita NT.

Dalším prvkem použitelným při práci s vlákny a procesy je mapování souboru do paměti. Namapujeme-li tentýž soubor do paměti ve více procesech, pak mají de facto sdílenou paměť. Toto je také jediný způsob, jak ve Windows sdílet operační paměť mezi procesy. Mapovanému souboru lze také nastavit souborové příznaky tak, že ani neexistuje nikde na disku a je jen v paměti. Sdílenou paměť pak lze bohatě využít ke komunikaci při spolupráci procesů, pro synchronizaci práce s ní pak použijeme některý synchronizační objekt (např. mutex nebo semafor).

Posledním prvkem, který zmíníme, jsou roury (pipe). Jde opět o prvek určený spíše ke komunikaci. Roura opět úzce souvisí se soubory, jedná se o datový kanál, který může být jednosměrný nebo obousměrný. Vždy má dva konce, z nichž každý může být nasměrován do libovolného procesu, ale i do souboru na disku, přes síť atp. S rourou se pracuje stejně jako se souborem na disku, tj. data můžeme číst a zapisovat. Rouru vytvoříme pomocí *CreatePipe* nebo *CreateNamedPipe*. Pojmenované roury mají ještě celou řadu dalších funkcí a lze jimi propojit i vzdálené počítače. Literatura uvádí, že pojmenované roury jsou základním komunikačním prostředkem ve Windows NT, použitým pro veškerou meziprocesní lokální i vzdálenou komunikaci (s tím důsledkem, že žádný jiný komunikační prostředek nemůže být rychlejší než pojmenované roury).

7.3 Futex

V závěru kapitoly se ještě krátce zastavme u Linuxu. V aktuálních verzích tohoto systému se jako základní primitivum používá *futex* (toto je i název příslušné systémové funkce, je to zkratka z Fast Userspace Mutex), což je speciální nízkoúrovňový prvek běžící převážně v user módu (tj. nepřepíná se do jádra). Do režimu jádra se přepíná jen při nutnosti řešit kolize mezi procesory a právě proto je velmi rychlý. Futex obsahuje počítadlo a lze jej využít k efektivní implementaci semaforu i mutexu dle standardu POSIX.

Přímé použití futexu programátorem obvykle není nutné, pro podrobnosti viz stejnojmenná funkce.

Shrnutí

Synchronizace je jedním z nezbytných pomocníků při práci s vlákny či obecně se sdílenou pamětí nebo jinými sdílenými prostředky. Základním nástrojem k synchronizaci je schopnost provádět několik po sobě jdoucích operací atomicky, kterou obvykle nabízí přímo procesor a jeho hardwarová platforma. Na základě jeho atomických instrukcí jsou v operačních systémech vybudovány synchronizační primitiva či objekty na vyšší úrovni abstrakce, která jsou pak nabízena uživatelským procesům. My jsme si popsali a naučili se používat především atomické operace test–and–set a compare–and–swap a synchronizační objekty semafor, mutex, kritická sekce, událost a signál. Dále jsme probrali také objektově orientovaný synchronizační prvek monitor a zmínili jsme také několik dalších synchronizačních a komunikačních pomocníků, které poskytuje Windows NT. Další praktické zkušenosti je možno získat na cvičeních (předmětu Operační systémy) či v publikaci [Kep08b].

Pojmy k zapamatování

- zasílání zpráv
- synchronizační primitivum
- atomická operace
- test–and–set
- compare–and–swap
- fetch–and–add
- Petersonův algoritmus
- semafor
- mutex
- kritická sekce
- událost
- signál
- monitor
- čekací funkce
- čekatelný objekt
- mapování souboru do paměti
- roura
- futex

Kontrolní otázky

1. *Které synchronizační primitivum je základní a proč?*
2. *Vysvětlete, proč se synchronizace používá častěji mezi vlákny než mezi procesy.*
3. *Co dalšího kromě vláken a procesů se synchronizuje pomocí synchronizačních primitiv? Uveďte také alespoň jeden konkrétní příklad.*
4. *Co je to atomická operace?*
5. *Jaké negativní důsledky může mít, když je atomická operace moc velká (hodně řádků kódu, déletrvající složitější algoritmus atp.)?*
6. *Uveďte praktický příklad kódu, který je chybný kvůli neatomickému provádění nějaké jednoduché operace ve více vláknech.*
7. *Jaká hardwarová atomická operace je „nejlepší“? Pokuste se to zdůvodnit.*
8. *Proč se Petersonův algoritmus v praxi téměř nepoužívá?*
9. *Jaký je principiální rozdíl mezi semaforem a mutexem?*
10. *Monitor je prvek s mnoha výhodnými vlastnostmi. Proč je k dispozici jen v objektově orientovaných systémech? Navrhněte, jak by mohl být přidán do neobjektových systémů.*
11. *Vysvětlete operaci compare–and–swap.*
12. *Ve Windows jsou události resetovány automaticky nebo manuálně. V jakých situacích se jeden nebo druhý typ používá? Popište je tak, aby byl patrný rozdíl, tj. proč je v některých situacích automatická či manuální událost nevhodná.*
13. *Jaký je ve Windows rozdíl mezi mutexem a kritickou sekcí? Jaký má toto členění přínos pro programy či programátory? Uveďte stručně výhody a nevýhody obou primitiv, kterými se liší.*
14. *Funkce SignalObjectAndWait je ve Windows atomickou operací. Kdyby tato funkce v systému nebyla, jak byste ji implementovali pomocí jiných primitiv? Jistě je více možností, takže vaše řešení zdůvodněte (uveďte výhody či nevýhody, které vás vedly k jeho použití).*

Cvičení

1. Dokažte, že při souběhu vláken při operaci push na zásobníku dojde při pořadí provedení ABBA, ABAB, BAAB nebo BABA k chybě. (Ukažte rozdíl mezi správným a skutečným stavem zásobníku.)
2. Problém kuřáků cigaret (Patil 1971 [Pat71]). Mějme tři kuřáky cigaret a jednoho agenta. Proces kuřák cyklicky zabalí cigaretu a vykouří ji. Potřebuje k tomu ale tři součásti: papír, tabák a zápalky. Jeden kuřák má jen papír, druhý jen tabák a třetí jen zápalky. Proces agent má neomezené množství všeho. Agent položí na stůl dvě náhodně vybrané součásti, kuřák, který právě tyto dvě potřebuje, je vezme, zabalí cigaretu, vykouří ji a dá signál agentovi, že dokouřil. Agent pak dá na stůl další dvě... Napište program implementující tuto situaci.
3. V MSDN si zjistěte, co je to APC a kdy se používá. V textu kapitoly byla uvedena informace, že APC je „poměrně málo používaná funkcionalita NT“. Co je důvodem? (Malý přínos APC, existence jiných podobných entit v systému, nebo uvedete něco jiného?)

8 Deadlock (uváznutí)

Studijní cíle: V této kapitole volně navážeme na kapitolu předchozí. Při spolupráci více vláken či procesů totiž může dojít k tzv. deadlocku (uváznutí), kdy výpočet nemůže dále pokračovat. Problematika deadlocku patří ke klasickým tématům studia operačních systémů.

Klíčová slova: deadlock, bankéřův algoritmus, graf prostředků, dvojfázové zamykání

Potřebný čas: 90 minut.

8.1 Úvod

Problém uváznutí, anglicky deadlock, patří ke klasickým tématům studia operačních systémů. Stručný úvod do problematiky je nasnadě: Máme-li vlákna či procesy synchronizované pomocí prvků uvedených v předchozím textu této kapitoly, máme v kódu mimo jiné také hodně míst, kde vlákno či proces čeká. Při masivním používání synchronizace není daleko k tomu, aby nastala situace, kdy každý na něco či někoho čeká, ale nikdo nedělá nic. Takové situaci se říká deadlock. Vlákno přitom může čekat jak na jiné vlákno, tak na nějaký hardwarový prostředek (v podobě systémového zdroje). Záleží na konkrétním operačním systému, jak jsou cíle čekání reprezentovány. (Jak už víme, v systému NT lze čekat na čekatelné objekty, ale také na události posílané do oken nebo události dokončení I/O operace.) Výskyt uváznutí je právě častější při práci se systémovými zdroji než při pouhé synchronizaci spolupracujících vláken.

Hovoříme-li o prostředcích, obecně platný postup používání prostředků má tři kroky:

1. Request – Proces si vyžádá prostředek od systému. Není-li prostředek právě k dispozici, vlákno čeká tak dlouho, než k dispozici bude.
2. Use – Vlákno má prostředek a pracuje s ním.
3. Release – Vlákno uvolní prostředek (vrátí jej „zpět“ systému).

příklad 1: otevření souboru, pak čtení/zápis, pak zavření

příklad 2: vyžádání hardwarového zařízení, práce se zařízením, uvolnění zařízení

8.2 Kdy nastává deadlock

Deadlock může nastat jen v případě, když platí tyto čtyři podmínky současně:

1. Mutual exclusion – Výlučné vlastnictví prostředků
   (Doslovný překlad: vzájemné vyloučení [procesů].) Alespoň jeden prostředek je ve výhradním vlastnictví, tj. vlastněn jedním procesem a jakékoliv jiné procesy v případě zájmu o něj musejí čekat.
2. Hold & wait – Drží a čeká
   Proces již vlastní nějaký prostředek, drží si jej a čeká na další.
3. No preemption – Není preempce
   (Alternativní překlad: neodnímatelnost [prostředků].) Když není preempce, proces má prostředek tak dlouho, než jej dobrovolně vrátí. Systém nemůže prostředek nikomu násilně odebrat.
4. Circular wait – Cyklické čekání
   Cyklickým čekáním nazýváme situaci popsanou na začátku kapitoly – když máme skupinu procesů, kde každý na někoho čeká. Nejjednodušší případ lze formálně popsat takto: Proces A vlastní prostředek 1 a chce prostředek 2. Proces B vlastní prostředek 2 a chce prostředek 1. Jakmile A požádá o 2 a B požádá o 1, dochází k cyklickému čekání. V obecném případě může být zúčastněn libovolný počet procesů vlastnících libovolné prostředky.

Průvodce studiem

Všimněte si, že čtyři uvedené podmínky nejsou zcela nezávislé. Např. čtvrtá podmínka automaticky implikuje druhou.

Průvodce studiem

Udělejme si jasno v tom, kdo na koho vlastně může čekat. V předchozí kapitole byla řeč o synchronizaci vláken, která se týká čistě jen vláken. Čeká-li nějaké vlákno kvůli synchronizaci, pak čeká na synchronizační objekt vlastněný jiným vláknem. V této kapitole hovoříme spíše o práci s prostředky, z hlediska deadlocku jsou však synchronizační objekty a systémové prostředky rovnocenné – na obojí lze čekat. Ostatně objekty i prostředky jsou vlastněny nějakými procesy či vlákny a čekáme-li na prostředek, pak čekáme na to, až ho proces či vlákno, které jej vlastní, uvolní.

8.3 Řešení deadlocku

K problematice deadlocku lze přistupovat v zásadě čtyřmi možnými způsoby:

1. Ignorance
   Toto je tzv. „pštrosí politika“ – problém nijak aktivně neřešíme („Nějak to prostě dopadne.“) Tímto způsobem se chová většina operačních systémů. (Přidejme jeden citát: „Maybe if you ignore it, it will ignore you.“ [Tan01])

2. Detekce a zotavení (detection & recovery)
   Deadlock řešíme, až nastane. Zjistíme-li jej, tak zrušíme jeden ze zúčastněných procesů.

3. Zamezení vzniku (prevention)
   Deadlock řešíme preventivně – zajistíme, aby některá ze čtyř podmínek deadlocku nemohla nikdy nastat. (Lze toho dosáhnout zavedením omezujících podmínek, jak lze žádat o prostředky.)

4. Vyhýbání se uváznutí (avoidance)
   Toto je nejsložitější přístup – systém vyhoví jen těm žádostem o prostředky, které nemohou vést k uváznutí, zatímco ostatní pozdrží. Tento přístup je nejpružnější, ale vyžaduje jistou spolupráci aplikací – ty musejí dopředu nahlásit, co vše budou v budoucnu potřebovat.

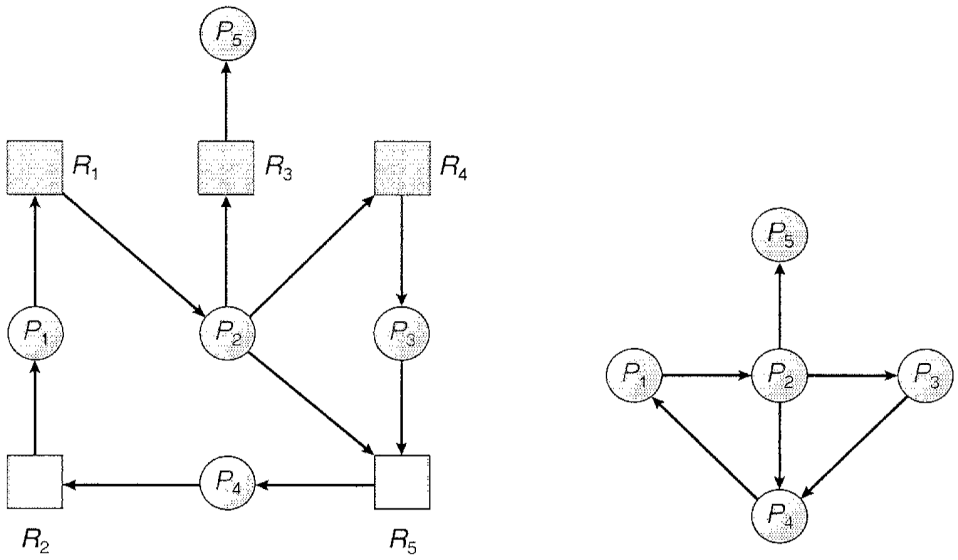
Jednotlivé přístupy si podrobněji probereme v následujících sekcích.

8.3.1 Detekce deadlocku

Nedělá-li systém prevenci, ani vyhýbání se deadlocku, pak deadlock může nastat. Systém by pak měl provádět detekci deadlocku a zotavení z něj, řeší se tedy dvě samostatné úlohy: 1. detekce, 2. zotavení. Ačkoliv tento přístup může vypadat jako poněkud méněcenný, pro některé systémy je výhodný – pokud deadlocky nastávají jen velmi zřídka, může být neekonomické provádět velmi složitou prevenci či vyhýbání se jim. Navíc každý systém obvykle obsahuje nástroje, kterými může uživatel (či správce) manuálně provádět zotavení (např. ukončení (kill) procesu, který běží s moc velkou prioritou nebo se jakkoliv pokazil); tyto nástroje pak mohou být použity k zotavení z deadlocku.

K detekci deadlocku používáme *alokační graf prostředků* nebo z něj odvozený *graf čekání* mezi procesy (viz obr. 11). Alokační graf je orientovaný graf znázorňující procesy jako kolečka

(P) a prostředky jako čtverečky (R). Orientované hrany vyznačují buď, že R „je vlastněn“ P, nebo že P „žádá“ R. Šipka od procesu k prostředku tedy znamená, že proces o něj žádá, obrácená šipka pak, že prostředek procesu patří. Předpokládáme přitom, že prostředky nelze sdílet, ovšem alokační graf lze samozřejmě zobecnit i na případ, že některé prostředky lze přidělit více procesům současně.



Obrázek 11: Alokační graf prostředků (vlevo) a jemu odpovídající graf čekání (vpravo). [SGG05]

Graf čekání (wait–for graph) zobrazuje „kdo na koho čeká“. Sestavíme jej velmi jednoduše z alokačního grafu tak, že vynecháme uzly prostředků a procesy spojíme tak, že hrana $P_1 \rightarrow P_2$ vede právě tehdy, když původně byly hrany $P_1 \rightarrow R$ a $R \rightarrow P_2$. Jsou-li v grafu čekání nějaké cykly, pak nastal deadlock.

Za účelem detekce deadlocku si tedy systém musí trvale udržovat graf čekání a pravidelně spouštět algoritmus hledání cyklů v grafu. To je samozřejmě výpočetně náročné, se složitostí rostoucí s počtem procesů (uzly) a počtem čekání mezi nimi (hrany). Otázka, jak často detekční algoritmus spouštět, tedy závisí na tom, jak moc je pravděpodobný výskyt deadlocku, a kolik vlastně máme v systému běžících procesů. Jelikož deadlock může nastat jen při čekání, nemá smysl spouštět detekci jindy než v okamžiku žádosti o prostředek, která způsobí čekání. Pokud tato žádost byla onou „poslední kapkou“ k deadlocku, pak rovnou známe i jeden z uváznutých procesů. Namísto prohledávání celého grafu tedy stačí vyjít od tohoto procesu a hledat jen cykly, ve kterých by byl obsažen.

Spouštět detekci při každé žádosti o prostředek, která není splněna bez čekání, je samozřejmě dost neefektivní a zpomalilo by to výkon systému. Je vhodnější detekci provádět jen jednou za čas, nebo při zjištění nevytíženého CPU (což jednak znamená, že CPU lze využít pro detekci, ale může to také znamenat, že počítač zřejmě nemá nic na práci, protože hodně procesů uvázlo v deadlocku).

8.3.2 Zotavení z deadlocku

Deadlock lze řešit dvěma způsoby: Buď odstřelíme (kill) jeden nebo více zúčastněných procesů, nebo násilně odebereme jeden nebo více zúčastněných prostředků.

Zotavení odstřelením procesu může proběhnout tak, že jsou odstřeleny všechny procesy zúčastněné v deadlocku a systém převezme zpět všechny jejich prostředky. To je značně likvidační, navíc to ani nemusí být jednoduché a rychlé. Druhou možností je vybrat jen jednoho kandidáta, odstřelit jej a zkontrolovat, zda tím byl vyřešen deadlock. Pokud ne, opakujeme proces zotavení

znovu. Nevýhodou tohoto postupu je, že musíme znovu a znovu projít detekcí deadlocku, což je také výpočetně náročné. Systém by zde navíc měl rozumně řešit otázku, který z procesů je ten vhodný kandidát na odstřelení. Řešení této otázky není jednoduché, jedná se v podstatě o obdobu plánování procesů (diskutováno v kapitole 5 na straně 41).

Zotavení preempcí prostředků může proběhnout i bez odstřelu běžících procesů. Pokud však tyto s možnou preempcí nepočítají, výsledek jejich výpočtu nebude korektní, což může v důsledku způsobit více škody než odstřelení procesu. Při preempci prostředků opět řešíme otázku, který prostředek vybrat k preempci a opět jde o poměrně složitou úlohu.

Odstřelení procesů i preempce prostředků s sebou nesou ještě jedno společné riziko – budeme-li takto postupovat, může dojít k hladovění (starvation), kdy opakovaně dochází k deadlocku, ten je řešen na úkor stejného procesu a jeho výpočet je opakován. Zajistit, aby nikdy nedošlo k hladovění tohoto typu, je samozřejmě také značně obtížné.

8.3.3 Zamezení vzniku

Zamezení vzniku (neboli prevence) deadlocku funguje na základě bodů uvedených v sekci 8.2. Deadlocku zamezíme tak, že zajistíme, aby některá z uvedených čtyř podmínek v systému nemohla nastat. Zastavme se nyní u jednotlivých případů.

Zamezení výlučnému vlastnictví

Výlučnému vlastnictví prostředků zamezíme teoreticky tak, že všechny prostředky budou sdílené. Jsou-li prostředky sdílené, pak je mohou používat všechny procesy současně a žádný nemusí čekat. V praxi však některé prostředky nelze sdílet, např. vypalovačka CD těžko bude vypalovat pro dva procesy současně – výsledek by byl nepoužitelný. U řady hardwarových zařízení však současné systémy poskytují jistý způsob sdílení, přinejmenším pomocí jisté úrovně virtualizace. Např. tiskárna rovněž nemůže tisknout dva dokumenty současně (podobně jako vypalovačka CD), ale systém ji virtualizuje pomocí spooleru a díky zařazování dokumentů do tiskové fronty se různým procesům může tiskárna jevit, jakoby ji měly ve výhradním vlastnictví. (Tiskové fronty byly jedním z prvních způsobů sdílení prostředků.) Podobně je sdílena také obrazovka nebo disková kapacita. Obecně však v systému mohou být zařízení, která sdílet nelze.

Zamezení držení a čekání

Držení a čekání zamezíme tak, že budeme vyžadovat, aby proces žádal o prostředky pouze tehdy, když žádné nevlastní. Tuto na první pohled přísnou podmínku lze splnit tak, že procesy jsou povinny žádat všechny prostředky před začátkem vlastního výpočtu. Tento přístup však není vhodný pro interaktivní aplikace, kde nemusí být předem jasné, které prostředky budou potřeba. (Např. počítačová hra může potřebovat přístup na síť pro hru více hráčů. Pokud však uživatel hru více hráčů nezapne, síťová komunikace nebude třeba, takže není důvod předem automaticky tento prostředek vyžadovat.)

Další alternativou je, že proces může žádat o prostředky i později než na začátku výpočtu a může je žádat i ve větším počtu současně, avšak nemůže nikdy žádat o další dříve, než systému vrátí všechny, které již má z dřívějška. Tento přístup je opět nepříliš vhodný pro interaktivní aplikace. Popsané algoritmy mají navíc dvě nevýhody. První je dosti nízká efektivita práce se zdroji – mnoho jich je přiděleno procesům, ale tyto je po delší dobu nemusejí využívat. Druhým problémem je nebezpečí hladovění. Vyžaduje-li proces ke své práci prostředky ve větším množství nebo alespoň nějaké populární, pak je velké riziko, že na ně bude donekonečna čekat, neboť v každý okamžik alespoň jeden z požadovaných prostředků vlastní jiný proces.

Umožnění odejmout prostředek

Zavedení preempce (odejmutí prostředku) je další možný přístup prevence deadlocku. V okamžiku, kdy proces žádá o prostředek, který mu nemůže být přidělen, systém odebere čekajícímu procesu všechny jeho prostředky a přidá je do seznamu prostředků, na které proces čeká. Proces pokračuje až v okamžiku, kdy může všechny požadované prostředky dostat, systém přitom musí zajistit obnovení jejich stavu tak, aby proces ani nepoznal, že mu byly prostředky odebrány. Druhou alternativou je postup, kde čekajícímu procesu nejsou odebrány všechny prostředky, ale při žádosti o prostředek se tento odebere preempcí tehdy, pokud jej vlastní proces, který sám na něco čeká. Tento postup je vhodný u prostředků, jejichž stav lze snadno obnovit, jako např. hodnoty registrů CPU.

Průvodce studiem

Příkladem systému, který používá prevenci deadlocku preempcí prostředků, je DirectX. U většiny prostředků se systém navíc nezabývá uložením jejich stavu, protože to z povahy věci není podstatné. (Např. obrazovka se překresluje celá mnohokrát za sekundu, takže nemá smysl obnovovat ji do stavu, který měla před dvojím přepnutím procesů.) Tato implementace systému 100% zamezuje deadlocku, ale nemusí být výpočetně úplně efektivní, protože procesy při práci s prostředky musejí předpokládat, že naprosto kdykoliv mohou o kterýkoliv prostředek přijít. Jelikož systém DirectX nepoužívá strukturované výjimky, kód programů je pak doslova prošpikován testováním návratových kódů jednotlivých systémových funkcí.

Zamezení cyklickému čekání

Poslední možnost prevence deadlocku je zamezení cyklickému čekání. Jak už bylo zmíněno, hold & wait je podmnožinou cyklického čekání, takže jistě nepřekvapí, že algoritmy zamezující těmto dvěma stavům si rovněž budou podobné. Cyklům zamezíme tak, že zavedeme globální pořadí prostředků, prostředky tedy budou mít svá globálně jedinečná čísla. Proces pak může žádat o prostředky pouze v tom pořadí, v jakém jsou očíslována. Alternativně je možno požadovat, že proces v okamžiku požadavku o prostředek nevlastní žádný s vyšším číslem (tj. mohl jej vlastnit dříve, ale už jej vrátil). Při implementaci číslování prostředků je nutno postupovat uvážlivě, pokud např. tiskárna bude mít menší číslo než disk, pak programy zřejmě nic nenatisknou, protože data k tisku je třeba nejdříve přečíst z disku. (I tento problém samozřejmě lze nějak řešit, ale vhodným číslováním je možno se mu zcela vyhnout.)

8.3.4 Vyhýbání se uváznutí

Prevence deadlocku diskutovaná v předchozích odstavcích funguje na principu omezení způsobu, jakým lze o prostředky žádat. Jak jsme si však ukázali, vedlejším efektem tohoto přístupu bývá malá efektivita využití prostředků a/nebo nízký výpočetní výkon systému. Jinou alternativou může být vyhýbání se uváznutí, kdy procesy mají plnou volnost v žádostech o prostředky, pouze se od nich vyžadují dodatečné informace o tom, co všechno ještě plánují žádat před tím, než své prostředky vrátí systému. Při každém dalším požadavku pak systém může díky informacím, které má, zkontrolovat, zda požadavku lze ihned vyhovět, nebo je třeba proces pozdržet, aby se systém vyhnul pozdějšímu deadlocku.

Průvodce studiem

Zjednodušeně lze vyhýbání se deadlocku popsat takto: Pokud systém dle svých znalostí „vidí“, že hrozí deadlock, určí jisté pořadí provádění procesů. Procesy pak neběží všechny

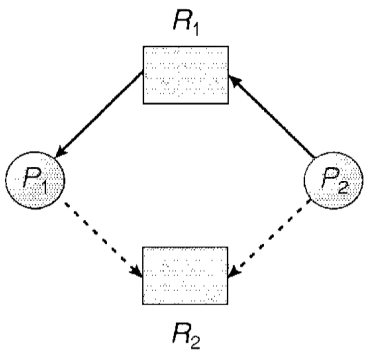
současně, ale v tom pořadí, ve kterém se systém deadlocku vyhne. (Některé procesy přitom mohou běžet současně, ale ne všechny.) Bohužel, jak si ukážeme níže, ne vždy takové bezpečné pořadí existuje. Klíčový je vždy okamžik, když někdo žádá o další prostředek – tehdy musí systém rozhodnout, zda je bezpečné mu jej přidělit hned, nebo až později.

Popišme si nyní jeden konkrétní algoritmus vyhýbání se uváznutí (deadlocku). Systém při něm od procesů požaduje, aby předem sdělily maximální počet prostředků každého typu, které mohou v budoucnu požadovat. Systém pak díky tomu může zajistit takové plánování procesů, kdy nedojde k cyklickému čekání, tedy nevznikne žádný deadlock.

Základem algoritmu je pojem *bezpečný stav*. Systém je v bezpečném stavu, jestliže existuje nějaké pořadí, v jakém bude systém odpovídat procesům na jejich požadavky, při kterém nedojde k deadlocku. De facto to znamená, že musí existovat jisté pořadí procesů, kdy jednotlivý proces může při žádosti o prostředky čekat pouze na procesy s nižším číslem. Pokud takové pořadí neexistuje, pak je stav nebezpečný. Bezpečný stav není nikdy deadlockem, avšak ani nebezpečný stav deadlockem být nemusí. Zavedení pojmu bezpečný stav přímo vede k algoritmu vyhýbání se uváznutí: Systém se vyhýbá uváznutí tím, že žádosti o přidělení prostředků vyhoví jen tehdy, kdy tímto nepřejde do nebezpečného stavu. V opačném případě proces čeká (přestože prostředek může být volný) a to tak dlouho, než se systém dostane do takového stavu, při kterém lze dané žádosti vyhovět a zůstat v bezpečném stavu. (Tj. proces musí čekat, než jiné procesy uvolní prostředky, díky kterým bude stav i po přidělení toho problematického bezpečný.)

Algoritmus na bázi alokačního grafu

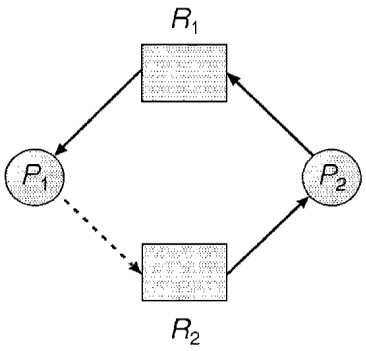
Máme-li v systému jen jeden prostředek od každého typu, můžeme pro vyhýbání se uváznutí využít algoritmus na bázi alokačního grafu prostředků, který byl představen v sekci 8.3.1. Graf rozšíříme o vyznačení plánovaných alokací prostředků, budeme je značit tečkovaně šipkou od procesu k prostředku. Od procesů žádáme, jak již bylo zmíněno výše, aby na začátku oznámily všechny plánované alokace, nebo aby oznamovaly nové plány pouze tehdy, kdy žádné prostředky nemají.



Obrázek 12: Alokační graf prostředků s vyznačením plánovaných alokací. [SGG05]

Příklad alokačního grafu vidíme na obrázku 12. Prostředek R2 je zatím volný, ale oba procesy jej mají v plánu alokovat. Požádá-li o něj nejprve P2, pak mu systém nesmí vyhovět, neboť by tím přešel do nebezpečného stavu (viz obrázek 13). Přidělením prostředku totiž v grafu vznikne hrana od prostředku k procesu, čímž v tomto případě vznikne cyklus.

Samotný cyklus ještě není deadlockem. Konkrétně na obrázku 13 je nebezpečný stav, ale deadlock ne. (Stejně jako v sekci 8.3.1, deadlock je v grafu vyznačen cyklem bez tečkovaných hran.)



Obrázek 13: Alokační graf prostředků s cyklem – nebezpečný stav. [SGG05]

Bankéřův algoritmus (Dijkstra 1965)

Algoritmus představený v předchozích odstavcích funguje při alokaci prostředků, kde od každého typu prostředku je v systému jen jedna instance. Systémy s větším počtem prostředků stejného typu mohou použít např. bankéřův algoritmus, který je obecnější, není však tak efektivní jako algoritmus alokačního grafu.

Bankéřův algoritmus má jméno podle své podobnosti s chováním bankéře, který ví, kolik maximálně budou všichni jeho klienti potřebovat, a jejich jednotlivé požadavky o další půjčky splňuje okamžitě, nebo pozdržuje na později tak, aby v každém okamžiku měl dostatečnou hotovost, aby mohl v konečném čase obsloužit všechny klienty.

Na začátku proces opět musí oznámit, kolik instancí prostředku každého typu bude během výpočtu potřebovat. Potom při alokaci systém opět kontroluje, zda splněním požadavku nepřejde systém do nebezpečného stavu a případně požadavek odložit na později. Je také možno říci, že aktuální stav je bezpečný, pokud existuje nějaká posloupnost provádění procesů, při které jsou požadavky všech uspokojeny. Zjištění, zda aktuální stav procesu je bezpečný, znamená porovnat aktuálně volné prostředky s aktuálně přidělenými a maximálními požadavky jednotlivých procesů a postupně si „odškrtávat“ ty procesy, u kterých je dostatečný počet prostředků k dokončení. Prostředky ve vlastnictví odškrtnutých procesů připojujeme do seznamu volných, na závěr musíme mít všechny procesy odškrtnuté. Pokud takové pořadí neexistuje, stav je nebezpečný a do takového stavu systém nesmí přejít. Formální popis bankéřova algoritmu najdete např. v [SGG05, Tan01].

8.4 Dvojfázové zamykání a další témata

Problematika deadlocků patří ke „zlatým“ tématům operačních systémů. Zatímco v počátcích této disciplíny se vědci deadlocky rozsáhle a hluboce zabývali, v současnosti je situace poměrně klidná – většina operačních systémů neřeší deadlocky nijak, některé úzce specializované aplikace řeší deadlocky v jisté míře samy a vědci se jimi nezabývají prakticky vůbec. Příkladem řešení deadlocku mohou být SQL servery, kde se setkáváme především s dvoufázovým zamykáním a u těch lepších i s transakčním zpracováním. Díky tomu jsou deadlocky nějakým způsobem řešeny alespoň tam, kde by jejich výskyt nejvíc bolel; zároveň to dokládá fakt, že řešení deadlocků je snazší u úzce specializovaných systémů než u obecných operačních systémů.

Dvoufázové zamykání je protokol vyžadující, aby práce s prostředky probíhala dvoufázově: V první fázi proces prostředky pouze alokuje, ve druhé fázi s nimi pracuje a uvolňuje je. Žádá-li proces o prostředek, který není volný, je celá první fáze restartována, tj. všechny prostředky jsou odebrány a začíná se odznova. U databázových serverů jsou prostředky obvykle části databáze, se kterými proces pracuje, a restartování procesu znamená znovuprovedení nějaké databázové operace či transakce sestávající z několika operací. Výhodou transakčního zpracování je, že

systém dokáže bez újmy na datech obnovit předchozí stav databáze k okamžiku, kdy byla zahájena transakce, která skončila nezdarem a byla zrušena (rollback). Dvoufázové zamykání a transakční zpracování je výhodné pro SQL databáze, ale samozřejmě nemusí být stejně vhodné pro ostatní typy úloh. Běžné operační systémy je nemohou obecně využívat už z principu interaktivní práce uživatele, která dnes převládá.

Situaci kolem vědeckého výzkumu v oblasti deadlocků shrnuje např. Tanenbaum [Tan01] tak, že „this research has died out“, s dodatkem k výzkumu distribuovaných řešení: „Its main function seems to be keeping otherwise unemployed graph theorists off the streets.“

Shrnutí

Deadlock čili uváznutí je potenciální problém každého systému, operační systém nevyjímaje. V této kapitole jsme si nejprve přesně popsali, že deadlock může nastat jen při splnění čtyř podmínek současně. Poté jsme se věnovali různým způsobům, jak se s problematikou deadlocku vypořádat.
Nepočítáme-li tzv. pštrosí algoritmus „strčit hlavu do písku“, pak máme tři možnosti, co dělat. První možností je pomocí alokačního grafu deadlocky detekovat až nastanou a řešit je posléze odstřelením jednoho či více procesů. Další možností je zamezení vzniku neboli prevence deadlocku, kdy systém nutí procesy chovat se tak, aby deadlock nenastal. Poslední alternativou je vyhýbání se deadlocku tím, že systém vyžaduje od procesů informace o budoucích alokacích a nechá procesy pracovat v určitém pořadí tak, aby bylo vždy zajištěno, že všechny procesy v konečném čase budou moci dokončit svůj výpočet.
Dále jsme si uvedli několik obecných charakteristik deadlocku, jako je jeho vztah k předchozí kapitole o synchronizaci, pozice tématu ve studiu operačních systémů nebo princip dvoufázového zamykání hojně používaný v SQL databázích.

Pojmy k zapamatování

- výlučné vlastnictví prostředků
- cyklické čekání
- alokační graf prostředků
- graf čekání
- bezpečný stav [systému]
- bankéřův algoritmus
- dvojfázové zamykání

Kontrolní otázky

1. *Jmenujte čtyři podmínky, za kterých nastává deadlock. Jsou to podmínky nutné, nebo dostačující?*
2. *Jmenujte tři situace, kdy vzniká deadlock a nesouvisí to s počítači.*
3. *Popište dvoufázové zamykání. Kde se obvykle používá? Proč jej nelze použít v obecném systému.*
4. *Popište bankéřův algoritmus.*
5. *Jak lze detekovat deadlock v systému?*
6. *Předpokládejme, že systém je v nebezpečném stavu. Ukažte na příkladu, že i tak je možné, že všechny procesy dokončí činnost bez výskytu deadlocku.*
7. *Předpokládejme, že prostředky lze alokovat a uvolňovat kdykoliv. Když proces A žádá o prostředek, který je právě ve vlastnictví procesu B, kontroluje se, jestli B není ve stavu čekání. Pokud ano, daný prostředek je mu odebrán a přidán do seznamu prostředků, na které B čeká. A dostane tento prostředek. Může v takovém systému dojít k deadlocku? Vysvětlete.*

8. *Může se vyskytnout deadlock na pouze jednom procesu? Dokažte.*

Cvičení

1. Implementujte bankéřův algoritmus pro jeden prostředek podle [Tan01].
2. Implementujte obecný bankéřův algoritmus podle [SGG05] nebo [Tan01].

9 Správa operační paměti

Studijní cíle: V úvodu našeho studia jsme se seznámili s von Neumannovým modelem počítače, který sestává ze tří základních částí. V této kapitole se dostáváme ke studiu druhé z nich – operační paměti. Tato sekce je opět rozdělena do několika kapitol. Toto je první z nich a seznámíme se s úvodními pojmy a základními způsoby organizace paměti, kterými je přidělováním souvislých úseků a stránkování. Řeč bude také o ochraně, která je neméně důležitou součástí správy operační paměti.

Klíčová slova: operační paměť, ochrana paměti, segmentace, stránkování, copy–on–write

Potřebný čas: 150 minut.

9.1 Úvod

Operační paměť je vedle procesoru druhou nejvýznamnější součástí každého počítače. Jak jistě každý ví z vlastní zkušenosti, paměti na počítači není nikdy dost, proto se operační paměť fyzicky skládá z různě velkých části pamětí různých typů. Především na moderních počítačích se na operační paměti účastní nejen klasická paměť RAM či ROM v mnoha variantách, ale také paměti externí, převážně v podobě pevných disků a externích flash modulů. Všechny dohromady pak za spolupráce operačního systému a hardwaru tvoří operační paměť, kterou může používat centrální procesorová jednotka a další inteligentní součásti počítače, jak jsme si již uvedli v kapitole 2. Připomeňme, že v sekci 2.2.3 jsme se seznámili s pojmy adresace paměti, báze, posunutí (offset), lineární adresa apod. S těmito pojmy budeme v této kapitole pracovat.

S operační pamětí souvisí především tyto základní funkce, na kterých se podílí hardware počítače a operační systém:

- Evidence prostoru volného a přiděleného procesům
- Přidělování a uvolňování paměti procesů
- Ochrana přiděleného prostoru
- Realizace virtuální paměti

V minulosti, a především na jednoúlohových systémech, se vesměs používaly velmi jednoduché algoritmy organizace paměti. Jednoúlohový systém, jak si ještě ukážeme později, vystačí s jednodušší správou paměti, protože některé problémy tam z principu nemohou nastat vůbec, nebo působí potíže jen okrajově. Druhým důvodem, proč se dnes tyto jednoduché algoritmy již nepoužívají, je, že při výrazně vyšší složitosti a větší paměťové kapacitě dnešních počítačů pro ně již nejsou vhodné; výrazně nižší cena za jednotku výpočetního výkonu také umožňuje nasazení i takových algoritmů a způsobů organizace, které by v minulosti nebyly ekonomické.

Na počítači je několik různých adresových prostorů.

Na současných počítačích a operačních systémech se obvykle vyskytuje několik nezávislých *adresových prostorů*, tj. samostatných způsobů číslování paměťových buněk. Existuje jich více ze dvou důvodů: Prvním důvodem je, že na víceúlohových systémech je vhodnější, aby každý proces měl svou operační paměť a v ní svůj vlastní adresový prostor. Druhým důvodem je hardwarové řešení počítačů, kdy tyto obsahují řadu nezávislých pamětí různých typů a každý z nich má tedy vlastní adresový prostor. Ve spolupráci hardwaru s operačním systémem pak dochází k mapování fyzické paměti do adresových prostorů jednotlivých procesů a to může být děláno i víceúrovňově, opět čistě z praktických důvodů, čímž vznikají další adresové prostory. Jednotlivé adresové prostory jsou tedy různými „pohledy“ na paměť počítače a představíme si je podrobněji ve zbytku této kapitoly.

Adresový prostor je pohled na paměť.

Průvodce studiem

Na historických sálových počítačích lze vidět, že i v minulosti na nejlepších (a nejdražších) počítačích existovaly sofistikované způsoby organizace paměti. Fakt, že řada počítačů dříve používala spíše hloupé a méně kvalitní způsoby organizace operační paměti, tedy není způsoben tím, že by tato problematika nebyla dostatečně odborně zvládnuta na teoretické úrovni. Řešení zvolené pro praxi je však vždy kompromisem mezi užitkem a cenou. Proto se třeba teprve dnes na běžné počítače dostávají řešení známá již před 30 lety.

Ochrana paměti

Důležitým tématem je také ochrana paměti – musí ji zajišťovat operační systém ve spolupráci s hardwarem. Smyslem ochrany je zajistit, že každý proces má přístup jen ke své části operační paměti a nemůže (ani záměrně, ani omylem) číst, nebo dokonce zapisovat a poškodit paměť jiného procesu. V minulosti se ochrana paměti neřešila; na jednoúlohových systémech nebyla ani tolik potřeba a opět šlo o kompromis mezi cenou a užitkem. Nechvalně proslulý je systém MS-DOS, který ačkoliv jednoúlohový, má v paměti obvykle řadu systémových ovladačů, které snadno mohou být porušeny chybným programem, neboť tento systém žádnou ochranu paměti nepodporuje. Přitom počítače PC AT podporují hardwarově ochranu paměti již od nejstaršího modelu, pouze chyběla podpora v operačním systému. Samotná realizace ochrany přitom přímo souvisí se způsobem organizace paměti, proto ji budeme diskutovat postupně pro jednotlivé případy.

9.2 Přidělování souvislých úseků (bloků) paměti

Přidělováním souvislých úseků (bloků) nazýváme situaci, kdy program je umístěn vcelku na jedno místo fyzické paměti. V zásadě pak lze rozlišit dvě varianty lišící se tím, zda bloky paměti jsou stejné délky, či nikoliv.

První možností je rozdělovat paměť do větších stejně dlouhých bloků. Každý program se pak bez problémů vejde do jednoho takového bloku. Jakmile program (přesněji proces) skončí, jím využívaný blok je vrácen zpět systému a může ho použít další proces. Tento přístup tedy zajišťuje dlouhodobě bezproblémový provoz, ale je zde malá efektivita využití paměti, neboť jednotlivé procesy velmi často potřebují mnohem méně paměti, než je velikost jednoho bloku. Tomuto jevu říkáme *vnitřní fragmentace* – paměť je obsazená, ale není používaná. Tento systém používal např. operační systém IBM OS/360.

Vnitřní fragmentace je u paměti obsazené, ale nepoužívané.

Druhou možností je rozdělovat paměť do bloků proměnlivé délky podle potřeby. Každý program pak může dostat tolik paměti, kolik ve skutečnosti potřebuje, takže nedochází k výše zmíněné vnitřní fragmentaci. Potřebuje-li program další paměť, může si požádat o další úsek libovolné délky. Tento způsob organizace paměti používá např. systém MS-DOS a na jeho příkladu můžeme vidět nevýhody tohoto řešení – efektivita využití paměti je sice zdánlivě velmi vysoká, ale po jisté době provozu se začne využití paměti zhoršovat. Tím, že bloky paměti jsou různé délky, totiž vzniká problém, kdy hodně paměti je volné, ale je rozkouskované do malých bloků, které nelze využít pro větší programy. Paradoxně totiž programy dělí svá data do menších částí, které mohou být umístěny v libovolném místě paměti, ale jednotlivé bloky souvisejících dat (včetně kódu jako celku) musejí být vždy umístěny společně – tj. na jeden požadavek o paměť musí systém najít jeden souvislý úsek požadované velikosti. Výslednému jevu se říká *vnější fragmentace* – paměť je sice volná, ale je nevyužitelná.

Vnější fragmentace je u paměti volné, ale nevyužitelné.

Vnější fragmentaci lze částečně předejít použitím nějakého „chytrého“ algoritmu přidělování paměti. V okamžiku, kdy proces žádá o paměť, totiž musí systém rozhodnout, který z mnoha volných úseků mu přidělí. Nabízejí se přitom nejméně tři možnosti, ze kterých může vybírat:

First fit vezme první vyhovující blok, který najde. Čili nijak se nesnaží rozmýšlet, který blok je výhodnější.

Best fit vezme nejmenší vyhovující blok. Důsledkem tohoto přístupu je, že v paměti zbývá velmi mnoho nevyužitelných bloků, ty jsou však co nejmenší.

Worst fit vezme největší blok. Důsledkem tohoto přístupu je, že v paměti nejsou žádné malé nevyužitelné bloky, ovšem nelze pak spustit velký program, protože velký úsek paměti byl rozebrán na obsloužení malých požadavků.

Bez ohledu na teoretické vlastnosti praxe ukázala, že worst fit je horší než první dvě metody co do efektivity i rychlosti. First fit a best fit jsou zhruba stejně efektivní, vhodnost se může lišit dle konkrétní situace. First fit je přitom podstatně rychlejší. Statistická analýza ukázala, že u first fit je při alokaci N bloků průměrně dalších $\frac{1}{2}N$ nepoužitelných z důvodu fragmentace. Čili zhruba třetina paměti je nevyužitelná. [SGG05]

Fragmentace znehodnotí zhruba třetinu paměti.

Průvodce studiem

Limity zde popsaných způsobů organizace paměti jsou dobře vidět na systému MS-DOS. Pokud má soubor programu 500KB kódu, tak potřebuje jeden souvislý úsek takové velikosti.

Situaci v MS-DOSu znepřehledňuje ještě rozšíření DPMI, které otevřelo možnost snadno používat paměť nad 640KB, čímž se obvykle de facto zbavíme problémů s pamětí, neboť v MS-DOSu obvykle spouštíme jen jeden program a ten si s fyzickou pamětí celkem jistě vystačí. DPMI je přitom jen předpis (standard) a záleží na jeho implementaci, jak je organizace paměti řešena. Ve skutečnosti tedy často stále existuje problém vnější fragmentace, ovšem nepůsobí už tolik problémy, neboť velikost fyzické paměti výrazně převyšuje potřeby jednoho programu v MS-DOSu.

Při přidělování souvislých úseků je fragmentace vždy přítomná. Můžeme se ale pokusit s ní bojovat a situaci alespoň trochu zlepšit. Pokud např. program potřebuje o 2 bajty méně, než je velikost volného bloku, pak tyto 2 bajty zůstanou v samostatném volném bloku a budou do budoucna již prakticky nepoužitelné. Operační systém navíc spotřebuje daleko více paměti pro evidenci tohoto bloku. První možností, jak zmenšit fragmentaci, je alokovat paměť po větších kusech než 1 bajt. Např. MS-DOS (za podpory hardwaru) alokuje paměť po 16bajtových blocích. Další jeden 16bajtový blok přitom spotřebuje pro evidenci každého paměťového bloku, přiděleného i volného. Při tomto systému pak proces žádající např. 30 bajtů dostane vždy minimálně 32. V paměti je sice opět fragmentace, tentokrát vnitřní, ale celkové využití je efektivnější než při přidělování po jednotlivých bajtech.

Průvodce studiem

Pro zajímavost dodejme, že v jazyku C/C++ se v systému MS-DOS používá vlastní správa paměti, kdy jazyk alokuje ze systému co největší blok pro sebe najednou a z něj pak dává programu menší kusy podle potřeby. Z každého alokovaného bloku přitom odebírá 4 bajty pro evidenci, což je mnohem lepší než oněch 16 bajtů, které pro svou evidenci využívá MS-DOS. Výhodu jazyk získává tím, že evidenci prostoru vede přímo v samotných blocích paměti, nepotřebuje tedy žádnou další paměť. Systému by sice taky stačily 4 bajty, ale nemůže si dovolit ukládat svá data v paměti, kterou přiděluje procesům, takže pro každý blok musí rezervovat jeden 16bajtový úsek navíc. Pokud však program v jazyku C chce alokovat např. 16 bajtů, ve skutečnosti potřebuje $16 + 4 = 20$ a spotřebuje tedy 32 bajtů, stejně jako by oněch 16 bajtů alokoval přímo z operačního systému.

Chování MS-DOSu z dnešního hlediska již není podstatné. Uvádíme jej zde proto, že díky své jednoduchosti je tento model snadno představitelný.

Ochrana paměti

Přidělování souvislých úseků je základním modelem organizace paměti operačním systémem a existuje k němu také související základní model ochrany paměti. Jak už víme z kapitoly 2.2.3, aby se proces dostal ke „své“ paměti, musí znát adresu počátku neboli bázi svého bloku. Tato je obvykle uložena přímo v procesoru ve speciálním registru. Proces pak pracuje pouze s posunutím (offsetem) od počátku tohoto bloku. Ochrana paměti pak může fungovat tak, že ke každému bázovému registru je přidán ještě druhý registr, a ten určuje velikost daného bloku paměti. Operační systém se stará o správné nastavení těchto dvou registrů a hardware kontroluje, že přístupy do paměti jsou jen v mezích [báze, báze+limit] a zajišťuje ochranu těchto dvou registrů – kdyby sám proces mohl nastavit libovolné hodnoty do těchto registrů, pak by o ochraně nemohla být řeč. Hardware tedy zajišťuje i nějaký druh odstínění úrovně procesů, čili dokáže rozlišit, zda je kód vykonáván systémovým, nebo uživatelským procesem.

Jak už bylo zmíněno výše, i když léta populární systém MS-DOS ochranu paměti neposkytoval, procesory řady x86 ano a to už od varianty 80286 použité v prvním modelu PC AT.

Průvodce studiem

Počítače IBM PC existovaly v několika variantách a používaly různé modely procesorů Intel řady x86. Varianta PC AT je ta, která se během let nejvíce rozšířila a dnes jsou všechny prodávaná „písíčka“ potomky právě tohoto typu. Již původní počítač IBM PC/AT (1984) měl zmíněný procesor Intel 80286 (1982), který podporuje hardwarovou ochranu paměti. Původní operační systém tohoto počítače byl však PC-DOS (přejmenovaný MS-DOS) 3.0, který žádnou ochranu paměti nepodporoval.)

9.3 Stránkování

Alternativou k přidělování souvislých úseků paměti je stránkování, kdy místo operační paměti naopak „kouskujeme“ programy. Řeší se tím problém vnější fragmentace paměti, o kterém byla řeč výše, přitom se zároveň otvírá cesta k využití pomocné cache na disku, kam lze odkládat méně využívané části operační paměti.

Stránkování řeší vnější fragmentaci.

Průvodce studiem

Zastavme se u otázky, proč výše popsaný model souvislých bloků nedovoluje odkládat část operační paměti na disk. Pokud bychom část adresového prostoru měli na disku, tak musíme nějak řešit problém, že při samotném běhu potřebuje proces mít svou paměť fyzicky v paměti, tedy ne na disku. Na disk se paměť totiž skutečně jen dočasně odkládá, program tam běžet nemůže. Odkládání na disk bychom tedy museli řešit přesouváním bloků paměti mezi různými adresami – to by bylo potřeba dělat kopírováním, což je samo o sobě pomalé, navíc by všechny programy musely být tvořené tak, aby počítaly s tím, že se jim libovolně při běhu mohou měnit adresy dat i kódu. Toto požadované zobecnění by zkomplikovalo kód programů. Pro operační systém by to také bylo mnoho práce navíc, protože by musel při každém přesouvání bloku paměti z a na disk procházet celé programy a měnit všechny adresy. To by počítač ještě více zpomalilo a vyžádalo by si to další paměť, kde by si operační systém evidoval všechny adresy, kde se vyskytují odkazy na jiné adresy, které je třeba při relokaci přepočítat. Výsledkem by byl operační systém, který by byl extrémně pomalý a ušetřenou paměť by navíc z části zabíral svými evidenčními daty.

Stránkování ruší základní příčinu problému vnější fragmentace: Programy nyní již nejsou v souvislých úsecích paměti. Tím odpadá řešení neřešitelného, tedy není již třeba řešit, do

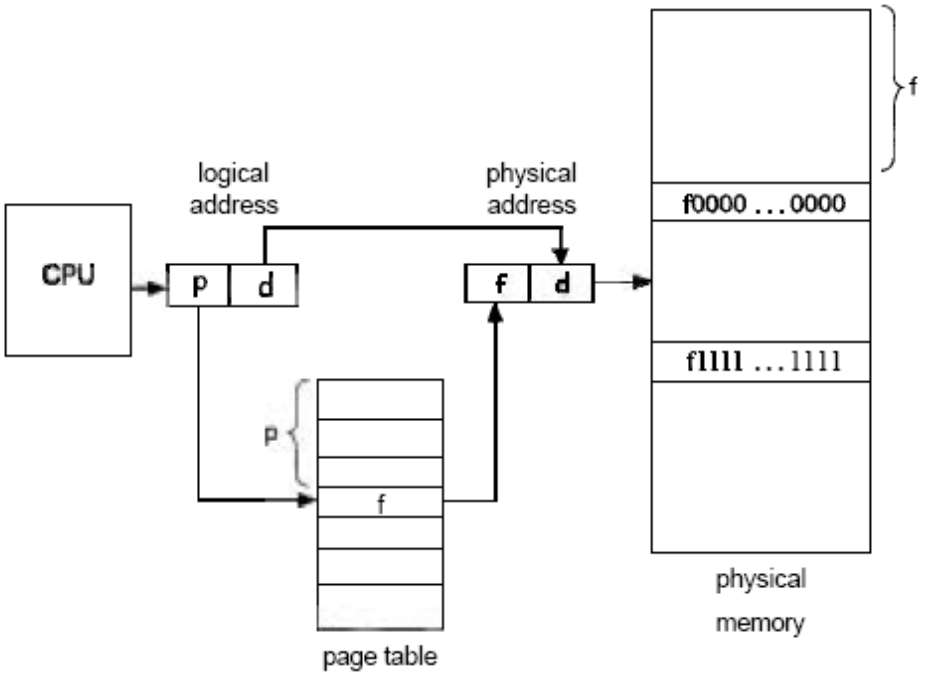
kterého z mnoha volných úseků umístit nový program, aby to někdy v budoucnu nezpůsobilo potíže.

Základní model

Rámec je kus fyzické paměti.

Stránka je kus logické paměti.

Fyzický adresový prostor je rozdělen na úseky stejné délky zvané *rámce*, logický adresový prostor je rozdělen na úseky stejné délky zvané *stránky*. Fyzická i logická adresa má lineární povahu. (Prostory ale mohou být řídké, tj. některé adresy mohou být vynechané, protože celá paměť je složením řady dílčích pamětí v počítači, které na sebe čísly adres nemusejí navazovat.) Stránka a rámec jsou pochopitelně stejně velké. Fyzicky jsou v počítači tedy jen rámce, zatímco procesy vidí jen stránky. Je pochopitelné, že procesy se nestarají o to, kde přesně fyzicky jejich stránky jsou umístěny.



Obrázek 14: Základní model stránkování. [SGG05]

Stránkování je vždy hardwarové.

Stránkování musí být implementováno hardwarově, neboť při každé adresaci paměti je třeba co nejrychleji přeložit logickou adresu na fyzickou. Jak ukazuje obrázek 14, systém udržuje *stránkovací tabulku* neboli tabulku stránek (page table), pomocí které pak hardware adresy převádí. Každá logická adresa se skládá ze dvou částí, tj. několik bitů pro číslo stránky p a několik bitů pro offset. Číslo stránky se vezme jako index do stránkovací tabulky a nahradí se tam uloženým číslem rámce f, čímž je získána fyzická adresa. Velikost stránky je obvykle mocninou dvojky, rozdělení adresy na číslo stránky či rámce a offset je pak velmi jednoduché: je to určitý počet bitů.

Stránkování a efektivita běhu programů

Stránka má nejčastěji velikost 4KB.

Zde popsaný model na tom z hlediska efektivity na první pohled není zrovna nejlépe. Řeší sice vnější fragmentaci, ale vnitřní fragmentace je v průměru rovna polovině velikosti stránky (tj. při typické velikosti stránky 4KB je to 2KB) na blok paměti. Samotný běh programů je pomalejší, protože při každé adresaci potřebujeme udělat dva přístupy do paměti – poprvé k načtení čísla rámce ze stránkovací tabulky, podruhé do samotné adresované paměti. Při použití stránkování jsou tedy programy teoreticky minimálně $2\times$ pomalejší. V moderních procesorech je však překladu adres věnována zvláštní péče a za pomocí cache pamětí TLB (Translation

Lookaside Buffer), kdy hodně používané části stránkovacích tabulek jsou neustále v procesoru, a paralelního vykonávání (viz pipelining, kap. 2.1.3 na str. 15) je dosaženo výsledné rychlosti blížící se nestránkovanému přístupu do paměti. Výhody získané stránkováním tak výrazně převažují a v praxi se skutečně používá.

Průvodce studiem

TLB je zvláštní druh cache paměti fungující jako asociativní pole. Je to tedy obdoba hashovací tabulky, ovšem implementovaná zcela hardwarově – při dodání klíče (čísla stránky) je odpovědí jemu odpovídající hodnota (číslo rámce). Je-li hodnota v tabulce, je odpověď velmi rychlá (cache hit). V opačném případě je hodnota vyhledána ve skutečné tabulce a odpověď je daleko pomalejší (cache miss). S každým cache miss dotazem se TLB aktualizuje doplněním nové hodnoty a vyřazením jedné staré.

Stránkování tedy umožňuje využít veškerou volnou paměť bez vnější fragmentace. Systém již nemusí řešit, který volný blok přidělit, protože všechny jsou doslova „stejně dobré“. Při nedostatku paměti je možno odkládat některé stránky na disk. V tom případě ke stránce neexistuje odpovídající rámec – řeší se to za spolupráce s hardwarem tak, že v okamžiku přístupu do stránky, která nemá přiřazen rámec, dojde k hardwarovému přerušení, operační systém načte chybějící stránku do nějakého rámce a proces adresace se opakuje. Běžící proces tedy ani nepozná, že nebyl v paměti – kdykoliv nějakou paměť potřebuje, tak ji v paměti má. Aby bylo možno načíst stránku odloženou na disk zpět do paměti, je obvykle třeba některou jinou stránku odložit na disk. Stačí tedy chybějící stránku vyměnit s některou jinou. Podrobněji se k této problematice vrátíme později.

Adresáře stránek

V zájmu bezpečnosti (ochrany paměti) má každý proces svou vlastní stránkovací tabulku. (Přesněji řečeno „může mít vlastní“, protože to závisí čistě na operačním systému a např. Windows 95 používá společnou stránkovací tabulku pro všechny procesy.) Podívejme se nyní na konkrétní čísla: Na 32bitových počítačích má jeden záznam v tabulce obvykle 4 bajty, to je 32 bitů, v systému tedy může být až 2^{32} rámců. Při standardní velikosti rámce 4KB je to 32 bitů pro číslo rámce plus 12 bitů pro offset, tj. 44 bitů celkem. V systému tedy lze fyzicky adresovat až 2^{44} bajtů, tedy 16TB.

16TB zní pěkně, problémem však může být samotná velikost stránkovací tabulky. I při omezení na běžnější 4GB limit máme 1M (2^{20}) stránek o velikosti 4KB (dohromady tedy 1M×4KB = 4GB). Jedna položka tabulky má 4 bajty (je to 32bitová adresa), takže celá tabulka zabírá 4MB. Máme-li na počítači např. 50 procesů, pak jejich stránkovací tabulky zabírají 200MB bez ohledu na to, jak velkou paměť vlastně počítač fyzicky má.

Průvodce studiem

Nezapomínejte, že je stále řeč o operační paměti, nikoliv o paměti RAM. Proto jsou nesprávné úvahy typu: „Když mám v počítači jen 512MB RAM, pak mi stačí menší stránkovací tabulky.“ Paměť RAM v podobě DIMM modulů do PC je sice nejznámější, ale v počítači je obvykle také řada další paměti a jednotlivé součásti navíc obvykle nevyplňují adresový prostor souvisle. Např. grafická karta má běžně 256MB nebo i více a mapuje je na adresy D0000000h (3/16 před koncem prostoru, čili od 3328MB výše).

Takové plýtvání pamětí je samozřejmě nežádoucí, proto se v praxi v řadě systémů používá dvojúrovňový model, který zabírá paměti méně. Namísto jedné stránkovací tabulky má každý

Adresáře šetří paměť.

proces řadu stránkovacích tabulek, kde každá popisuje část adresového prostoru. Pokud však nějakou část adres nepoužívá, daná stránkovací tabulka neexistuje, tedy nezabírá paměť. Stránkování je dvouúrovňové – část adresy určuje číslo stránkovací tabulky, další část pak index do této tabulky a zbytek je offset. Při velikosti stránky 4KB může být např. 10 bitů pro číslo tabulky, 10 bitů pro index v tabulce a 12 bitů pro offset. Celkově má adresování stejné schopnosti jako výše, avšak adresový prostor je popsán pomocí 1024 stránkovacích tabulek, kde pouze ty odpovídající využitým adresám existují. Každý stránkovací tabulka má tedy 1024 položek (2^{10}) o 4 bajtech, čili zabírá právě jednu stránku. Adresář stránek má taky 1024 položek o 4 bajtech, tedy také jednu stránku. V tomto případě tedy do sebe vše velmi dobře zapadá, proto se tento model v této podobě používá i v praxi, jak si ukážeme níže.

Ochrana paměti

Ochranu paměti na úrovni stránkování lze realizovat pomocí bitu (oprávnění) přístupu uloženého ve stránkovací tabulce. Tento jeden vyhrazený bit pak určuje, zda je rámec přístupný jen pro čtení, nebo i pro zápis. Při každém překladu adres je pak bit přečtený a použitý dále při práci s touto adresou. Toto lze rozšířit i na více stavů, např. pro zvláštní označení paměti, ve které je povoleno vykonávat kód. Podobně je možno jedním stavem označit stránky, které jsou zcela neplatné (neexistující) a není k nim vůbec povolen přístup. Neexistující stránky lze také řešit pomocí omezení velikosti stránkovací tabulky (či adresáře stránek, je-li používán), nejlépe jednoduše použitím speciálního registru, který ponese počet platných záznamů ve stránkovací tabulce. (Pro označení stránky jedním ze čtyř uvedených stavů nám tedy stačí dva bity.)

Systém IBM/360 používal na úrovni stránkování ochranu typu zámek a klíč – každý proces má svůj 4bitový identifikátor (klíč) a každá stránka má v tabulce také 4bitový identifikátor (zámek). Přístup do paměti je povolen jen tehdy, když se klíč a zámek shodují, nebo když jde o univerzální klíč (číslo 0 – proces jádra). Tímto způsobem mohou procesy (jmenovitě až 15 procesů) sdílet společnou stránkovací tabulku.

NX bit chrání proti zneužití přetečení bufferu.

Dodejme ještě, že procesory typu x86 tento způsob ochrany paměti na úrovni stránek nepoužívají, zatímco procesory novější řady x64 ano. Jeden bit je tam vyhrazen pro NX (no–execute, také značeno XD jako execute disable), jeho nastavením je zakázáno v dané stránce provádět kód. Tato ochrana je zaměřena zejména proti virům – znemožňuje totiž model zavedení škodlivého kódu přetečením bufferu (buffer overflow), kdy se přeplněním bufferu zapíše do oblasti dat (konkrétně zásobníku) procesu škodlivý kód a zároveň se zásobník změní tak, aby se při návratu z volání (ret) přeneslo řízení na adresu obsahující tento škodlivý kód. Je-li v datových stránkách zakázáno provádění kódu, tento typ útoku není možný.

9.4 Princip lokality

Jak jsme si již naznačili výše, složité adresovací mechanizmy moderních počítačů by se neobešly bez TLB, který zrychluje jinak složité výpočty tím, že při opakovaném přístupu do stejné oblasti paměti se nemusí pro překlad adresy vůbec číst skutečná stránkovací tabulka. Využívání tohoto druhu cache současně přináší možnost a zároveň víceméně nutnost používat takové modely programování, které v určitém krátkém časovém úseku využívají data a kód umístěné především na jednom místě v paměti. Pokud totiž toto platí, program běží rychleji.

Tuto vlastnost má používání lokálních proměnných ve funkcích a také objektově orientované programování – v určitém krátkém časovém úseku se tam používají především lokální proměnné umístěné na jednom místě na zásobníku respektive data jednoho objektu, jehož metoda právě běží.

Průvodce studiem

Princip lokality přístupu do paměti je něco podobného jako spojitost funkce v bodě v matematice: Princip lokality znamená, že v nějakém dostatečně malém časovém úseku se program pohybuje jen v malém omezeném prostoru paměti – to se týká jak kódu, tak dat. (Obvykle, tj. ne nutně vždy.)

Podívejme se nyní, kolik času používání TLB ušetří. Přístup do paměti bez TLB znamená vždy dvojí čtení paměti plus hledání v tabulce stránek. Program je tedy více než $2\times$ pomalejší. Rychlost hledání v TLB se liší podle toho, zda je hledaná hodnota nalezena, či nikoliv. Při cache hit se čas potřebný k nalezení blíží nule, při cache miss se opět přistupuje $2\times$ do paměti. Celková rychlost pak závisí na poměru cache hits : cache misses, ten je empiricky zjištěn kolem 99:1, tj. 99% přístupů je cache hits. Tento poměr samozřejmě hodně závisí na zvoleném programovacím paradigmatu – paradigmata podporující princip lokality jsou zde pochopitelně ve výhodě. Je-li např. čas potřebný k překladu adresy při cache hit 1.5T a při cache miss 30T, pak průměrný čas při 99% úspěšnosti je $0.99 \cdot 1.5 + 0.01 \cdot 30 = 1.785T$. Ve skutečnosti pak do hry vstupuje ještě pipelining, protože moderní procesory se snaží vypočítávat adresy v předstihu. Při cache hit je potřebný čas překladu 0T a při cache miss může být cca. 10T nebo i více, ale závisí zejména na podílu miss případů – čím méně jich je, tím je rychlost větší díky pipeliningu. (Tyto odhady berte jen jako čistě přibližné.)

9.5 Stránkování na 64bitových procesorech

V sekci 9.3 jsme si ukázali efektivní způsob implementace stránkování pomocí dvojúrovňových stránkovacích tabulek, kde 32bitová adresa je rozložena na $10 + 10 + 12$ bitů a adresář stránek i jednotlivé stránkovací tabulky obsahují 1024 položek a jsou tedy velké přesně jeden rámec (4KB).

Jak uvádí [SGG05], na 64bitových počítačích tento model nebude fungovat, protože při zachování $10 + 12$ bitů pro stránkovací tabulku a offset zbývá $64 - 22 = 42$ bitů pro adresář stránek. Položky musejí být 8bajtové, takže celková velikost adresáře je až 2^{45} bajtů (32TB) pro každý proces.

Situaci lze řešit zavedením další úrovně tabulek, čili adresu rozdělíme na tři části indexů do tabulek plus offset, tedy např. $32 + 10 + 10 + 12$. První tabulka má stále 32bitové indexy, čili zabírá až 2^{35} bitů (32GB). Při používání trojúrovňových tabulek se také proporcionálně zpomaluje překlad adresy při cache miss. Můžeme samozřejmě přidat ještě čtvrtou úroveň tabulek, pak má největší z nich „maximálně“ 32MB. Ve všech případech předpokládáme stránkovací tabulky o velikosti 8KB ($1024\times$ 8 bajtů).

Průvodce studiem

Výpočty v předchozích odstavcích jsou odvozeny od informací v knize [SGG05], jsou zde však opraveny některé tam se vyskytující chyby. Celá teorie je však poněkud mimo realitu, neboť předpokládá využívání 64bitových počítačů, které používají celý 64bitový adresovací prostor, mají zhruba 2^{64} fyzické paměti a každý proces zhruba (řádově) tolik paměti používá i sám pro sebe. Tato představa však naprosto neodpovídá současné reálné situaci, kdy 64bitové počítače mají sice 64bitové adresování, avšak skutečné velikosti pamětí se blíží těm 32bitovým. Proto lze vyvodit, že také adresovací algoritmy u současných 64bitových procesorů mohou bez problémů být stejné jako u procesorů 32bitových.

Stránkovací systém skutečného 64bitového procesoru typu x64 probereme podrobněji v kapitole 11.3 na straně 119.

Kromě zde diskutovaného hierarchického systému adresovacích tabulek, kdy se pro překlad používá lineární řada tabulek, existují i další alternativy, jako např. hashované stránkovací tabulky nebo clusterované stránkovací tabulky. Tyto metody jsou vhodnější pro řídké obsazení adresovacího prostoru. Současné 64bitové procesory však stále vystačí s lineárním modelem, např. nejběžnější procesory řady x64 používají čtyřúrovňový hierarchický systém tabulek; podrobněji si stránkování na těchto procesorech představíme dále v textu.

Průvodce studiem

Míchání slov hierarchický a lineární by mohlo být matoucí, proto si jej vysvětleme: Řeč je pořád o stejném modelu stránkování paměti. Systém adresovacích tabulek je hierarchický, protože stránkovací tabulky jsou uspořádány v hierarchii (ve stromu). Překlad adres používající tento model je pak lineární, protože při překladu čteme tabulky lineárně – vždy jednu na každé úrovni hierarchie.

9.6 Segmentové adresování (segmentace)

Se segmentovým adresováním jsme se již seznámili v kapitole 2, nyní se k němu vrátíme v kontextu nových znalostí. Připomeňme nejprve, že segmentové adresování funguje tak, že proces má paměť rozdělenou do několika segmentů, obvykle nejméně na kód, data a zásobník. Další segmenty je pak možno použít pro další data.

Pro lepší pochopení systému organizace paměti si popišme souvislost segmentace a stránkování. Používá-li počítač segmentaci i stránkování dohromady, což je běžné např. na procesorech typu x86, pak překlad lineárních adres na fyzické probíhá standardním způsobem, jak bylo popsáno v této kapitole (a podrobnosti budou ještě diskutovány dále). Samotné programy však nepracují přímo s lineární adresou, nýbrž používají logickou adresu ve formě segment + offset. Segment je popsán segmentovým registrem tak, jak jsme si uvedli v kapitole 2 a při běžných operacích program pracuje jen s offsetem. K němu se při práci s pamětí automaticky připočítává báze segmentu a tím teprve z logické adresy vzniká lineární adresa, která se mapuje na fyzickou (buď přímo, nebo pomocí stránkování).

Průvodce studiem

Segmentaci lze používat i samostatně, bez stránkování. Segmentace sama o sobě je totiž možným řešením správy paměti. Tento model však podrobněji nediskutujeme, neboť se jedná jen o konkrétní implementaci modelu přidělování souvislých úseků paměti.

Ochrana paměti

V sekci 9.3 jsme k ochraně paměti na straně 90 uvedli, že procesory x86 tam popsaný způsob ochrany na úrovni stránkování nepoužívají. Jelikož veškerý přístup do paměti jde přes segmentové registry, ochrana paměti je na těchto procesorech realizována právě na úrovni segmentů. Celý segment má tedy vzhledem k paměti stejná práva. Základní ochranou je stanovení dolní a horní hranice segmentu v jeho deskriptoru, takže se vůbec nelze dostat mimo přidělenou paměť. Dolní hranice segmentu může být na libovolném bajtu lineárního prostoru a horní limit paměti může být buď přímo v bajtech (max. 1MB), nebo ve stránkách (max. 4GB). Nižší rozlišovací schopnost u horního limitu je způsobena nižším počtem bitů vyhrazeným pro tento údaj. V deskriptoru lze nastavit ještě několik příznaků, ochranu paměti ovlivňuje DPL (descriptor privilege level), kterým lze nastavit požadovanou úroveň procesu pro přístup k segmentu[3].

[3] Deskriptory mohou popisovat i jiné prostředky používané procesorem než segmenty. DPL tedy chrání přístup k různým prostředkům, ne jen k paměti.

Nastavením nuly povolíme přístup jen jádru operačního systému. Nulováním bitu P (segment present) se označují zcela neplatné segmenty.

Pomocí segmentování lze na 32bitových procesorech x86 využít i více než 2^{32} bajtů (4GB) paměti. V praxi se však tato možnost většinou nepoužívá a hlavní výhodou zavedení segmentů je tedy právě ochrana paměti na nich založená. (Připomeňme, že v 16bitovém režimu, např. v systému MS-DOS, byla hlavní výhodou zavedení segmentů naopak možnost využít větší množství paměti než 2^{16} bajtů.) K problematice využití segmentů při adresování velké paměti se ještě vrátíme v kapitole 11.1.3 na straně 111.

9.7 Copy–on–write

Zastavme se nyní u možnosti sdílet paměť. Z hlediska procesů může být zajímavé, když spolu mohou komunikovat a/nebo spolupracovat pomocí sdílené paměti. Z hlediska operačního systému je však zajímavější dívat se na sdílení paměti jakožto prostředek vedoucí k úspoře: Je-li spuštěn nějaký program vícekrát, pak sdílením kódu lze ušetřit i poměrně hodně paměti. Programy jsou ne náhodou v mnoha systémech členěny na tzv. knihovny, ty jsou umístěné v samostatných souborech a je také výhodné je sdílet. Sdílení stránek paměti realizuje systém jednoduše tak, že tyto stránky naváže na stejné rámce. Systém však samozřejmě má jistou práci navíc s evidencí sdílené paměti – při jakékoliv operaci s rámcem je potřeba změny zanést do všech procesů, které rámec používají. Sdílet přitom lze jak paměť kódu, tak paměť dat.

Dva různé programy mohou sdílet některé knihovny. Příkladem může být Microsoft Word a Microsoft Excel – ty dva kancelářské programy prodávané společně mají mnoho společného a také sdílejí velké množství společného kódu. Díky schopnosti sdílet paměť ušetří operační systém při současném spuštění těchto dvou programů mnoho paměti. A kromě úspory paměti samozřejmě dochází i k úspoře času při startu programu, který používá knihovny, které již jsou v paměti. Pro operační systém to samozřejmě znamená vést si evidenci souborů, které jsou již v paměti, aby mohl sdílení provádět automaticky.

Copy–on–write šetří paměť při sdílení.

Copy–on–write je speciální technika sdílení paměti, kterou nejlépe vysvětlíme na příkladu unixové funkce `fork`. Připomeňme, že tato funkce zdvojí proces a ten pak pokračuje na stejném místě kódu ve dvou exemplářích, které se navzájem dokáží rozlišit pomocí návratové hodnoty této funkce. Při volání fork systém musí zajistit, že každá z kopií procesu bude mít vlastní kód i data tak, aby se dva procesy navzájem při dalším běhu nechtěně neovlivňovaly. Namísto toho, aby systém při volání fork složitě a zdlouhavě duplikoval veškerou paměť procesu, označí paměť kódu jako sdílenou a paměť dat jako copy–on–write. Jelikož se kód při běhu nemění, oba procesy budou sdílet stejnou kopii. Data se měnit mohou, ale nemusí; označení copy–on–write zajistí, že jednotlivé stránky dat se duplikují až při prvním pokusu o jejich změnu.

Copy–on–write je možno implementovat poměrně jednoduše na systémech podporující ochranu paměti před zápisem. Copy–on–write stránky systém označí jako read–only (jen pro čtení) a eviduje si, které stránky to jsou. Při pokusu procesu o zápis do takto chráněné paměti dojde k výjimce a přerušení access violation (nepovolený přístup). Systém v obsluze této výjimky projde seznam evidovaných copy–on–write stránek a pokud najde záznam, tak stránce alokuje další rámec, okopíruje ji tam, zruší aktuálnímu procesu sdílení a povolí mu zápis. Zbyde-li u původní stránky poslední proces, který ji už s nikým nesdílí, systém mu rovněž povolí do stránky zápis. Z pohledu uživatelského procesu je tedy copy–on-write transparentní. Vzhledem k tomu, že po volání fork velmi často načteme do procesu jiný program, je copy–on–write také velmi efektivní z hlediska úspory paměti i výpočetního času. I v případě, že nedojde k načtení nového programu do procesu, se díky copy–on–write ušetří mnoho paměti pro data, která procesy nebudou měnit (např. globální konstanty, ale také prostor zásobníku nad místem rozdělení).

Copy–on–write se používá také při ladění kódu. Při ladění vkládá debugger tzv. break pointy (ladicí body zastavení) do kódu tak, že na příslušná místa vkládá instrukci, která způsobí

výjimku/přerušení a běh procesu je v daném místě zastaven. Máme-li spuštěno více programů používajících knihovnu, kterou takto ladíme, pomocí copy–on–write je vytvořena kopie pro laděný proces. Ostatní procesy tedy běží dále a pouze jeden zvolený ladíme.

Ve Windows NT se ke spouštění procesů (obvykle) nepoužívá **fork**; proces startujeme přímo uvedením spustitelného souboru ve funkci **CreateProcess**. NT při spouštění libovolného souboru (platí pro EXE i DLL) tento mapuje do paměti a pamatuje si, kde v paměti tento soubor je. Při potřebě stejného souboru jiným procesem pak systém již jen odkáže na místo, kam je soubor namapován. Aby bylo možno mapovat i soubory, které potřebujeme měnit, mapování používá princip copy–on–write. Jak si ukážeme v následující kapitole o virtuální paměti, systém mapování souborů navíc načítá do paměti jen ty části souborů, které procesy skutečně používají. Takže máme-li namapovaný velký soubor, ze kterého používáme jen malou část, systém si eviduje, že soubor je mapován do paměti, eviduje si také, kam je mapován, ale ty části souboru, které nepoužíváme, ve skutečnosti vůbec nezabírají paměť a tato může být použita pro jiné účely. Jedná se tedy o propracovaný systém, kde všechno souvisí se vším; na další podrobnosti se ještě podíváme později.

Shrnutí

Tato kapitola uvedla blok výuky zaměřený na organizaci a správu operační paměti. V úvodu jsme se seznámili se základními úkoly správy paměti, kterými je evidence prostoru volného a přiděleného procesům, přidělování a uvolňování paměti procesům, ochrana přiděleného prostoru a realizace virtuální paměti.
Dále jsme si představili model správy paměti využívající přidělování souvislých úseků, který se v různých variantách používal především v minulosti. Jeho alternativou používanou v současných počítačích je stránkování, které odstraňuje vnější fragmentaci – základní nevýhodu modelu pracujícího na bázi souvislých úseků.
Seznámili jsme se také s principem lokality, který vysvětluje, proč programy využívající především lokální proměnné a objektově orientované programy jsou na stránkujících počítačích rychlejší. Dále jsme si připomněli pojem segmentace a umístili ho do kontextu se správou paměti a stránkováním.
V průběhu kapitoly jsme se věnovali také problematice ochrany paměti; pro lepší srozumitelnost jsme ji diskutovali průběžně u každého modelu správy.
V následující kapitole budeme pokračovat dalším modelem správy paměti využívajícím virtuální paměť.

Pojmy k zapamatování

- adresový prostor
- ochrana paměti
- vnitřní fragmentace
- vnější fragmentace
- báze a posunutí (offset)
- stránkování
- rámec
- stránka
- stránkovací tabulka
- TLB (translation look–aside buffer)
- hierarchický systém stránkovacích tabulek
- adresář stránek
- zámek a klíč
- NX bit (no–execute)
- segmentové adresování (segmentace)

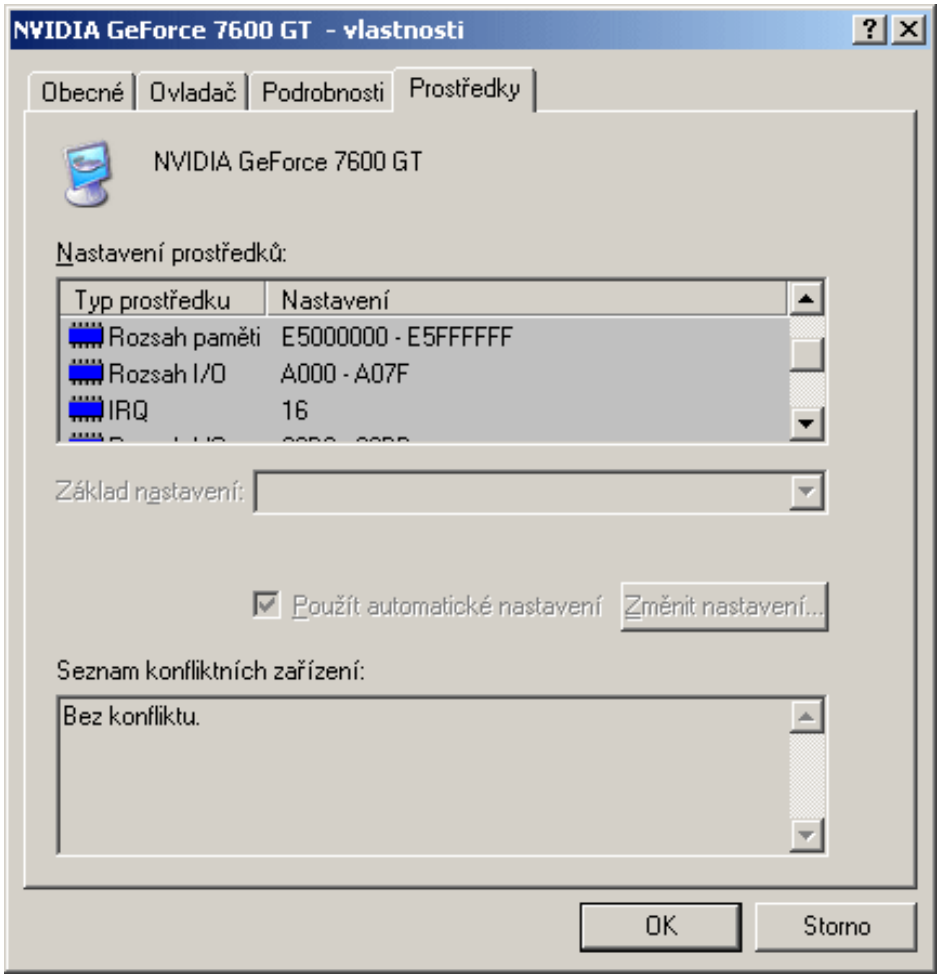
- copy–on–write

Kontrolní otázky

1. *Jaké jsou základní funkce správy paměti v operačním systému?*
2. *Vysvětlete pojmy vnější a vnitřní fragmentace.*
3. *V textu byly popsány tři algoritmy přidělování paměti při používání souvislých bloků: first fit, best fit, worst fit. Který z nich je nejlepší? Proveďte diskuzi.*
4. *Jaká velikost stránek se obvykle používá?*
5. *Vysvětlete, proč při stránkování nevzniká vnější fragmentace.*
6. *Popište vztah segmentace a stránkování paměti.*
7. *Jaké problémy přináší segmentace v programovacích jazycích používajících pointery? Proč se s těmito problémy v praxi dnes již setkáváme málo?*
8. *Jakým způsobem může být implementována ochrana paměti na úrovni stránkování?*
9. *Jakým způsobem může být implementována ochrana paměti na úrovni segmentace?*
10. *Co je to copy–on–write a co k tomu musí hardware umět?*
11. *Proč jsou v praxi používané velikosti stránek vždy mocninou dvojky?*
12. *Co bude výsledkem, když povolíme, aby více stránek odkazovalo na stejný rámec? Popište, jak to může zrychlit kopírování paměti.*
13. *Popište, jakým způsobem mohou dva procesy sdílet stejný segment.*

Cvičení

1. Dokažte, že vnitřní fragmentace při stránkování je rovna jedné polovině velikosti stránky.
2. Ukažte, že při vhodně zvolené velikosti stránek a stránkovacích tabulek všechny datové struktury pasují do celých stránek a všechny bity jsou užitečně využity.
3. Mějme logický adresový prostor velikosti 8 stránek o 1024 bajtech a fyzický adresový prostor o 32 rámcích. Kolikabitové jsou zde logické a fyzické adresy?
4. Máte-li ve svém počítači Windows, prohlédněte si ve správci zařízení (stiskněte Win–Break, potom karta Hardware ⟶ Správce zařízení), jaké rozsahy adres a další prostředky váš počítač používá. Každé zařízení, které má vlastní paměť či mapuje své služby do adresového prostoru, má zveřejněný rozsah použitých adres. Viz obrázek 15. Všimněte si, že u jednotlivých zařízení se tyto prostory většinou nesmějí překrývat.



Obrázek 15: Prostředky použité grafickou kartou

10 Virtuální paměť

Studijní cíle: V této kapitole navážeme na kapitolu předchozí. Řeč bude o virtuální paměti, což je další model správy paměti, jehož cílem je umožnit využití diskového prostoru pro odkládání méně používaných dat z fyzické paměti.

Klíčová slova: virtuální paměť, výměna rámců, výběr oběti, LRU, thrashing

Potřebný čas: 160 minut.

10.1 Úvod

Virtuální paměť zřejmě každý uživatel počítače zná z vlastní zkušenosti. Fyzické paměti RAM, která je hlavním prostředím vykonávání programů, je obvykle málo[4], a tak současné operační systémy hojně využívají tzv. *virtuální paměť* a *swapování*, kdy je k fyzické paměti připojen ještě velký soubor na pevném disku a za pomoci operačního systému se počítač tváří, jakoby fyzické paměti bylo ve skutečnosti více. Zatímco jednotlivé procesy vůbec nepoznají, že jejich kód či data občas cestují mezi fyzickou pamětí a diskem, v pozadí tohoto systému je složitý mechanizmus postavený na spolupráci hardwaru a operačního systému. Jelikož realizace virtuální paměti je úkolem především pro operační systém a úloha hardwaru je jen doplňková, je tomuto tématu věnována celá samostatná kapitola.

Virtuální paměť je operační paměť zahrnující jak fyzickou paměť, tak vyhrazený prostor na pevném disku. (Obecně lze použít i jiné médium, pevný disk je zde jmenován jako nejčastější případ.) Swapování je proces, kdy operační systém vyměňuje úsek (blok) operační paměti mezi fyzickou pamětí a pevným diskem. Často se používá také pojem *stránkovací soubor* nebo také *swapovací soubor* – to je právě onen soubor na disku obsahující data namapovaná systémem do operační paměti. (Pro dosažení vyšší rychlosti to může být i více souborů na různých discích.) Tato data sice obvykle jsou uložena na pevném disku jako běžný soubor, ale z důvodu ochrany paměti operační systém blokuje k tomuto souboru přístup. (Bezpečnostní otazník je pak nad situací, kdy násilně vypneme počítač (simulujeme havárii přerušením dodávky proudu) a pevný disk přeneseme do jiného počítače. Je věcí operačního systému, aby nedovolil swapovací soubor použít po havárii systému (pokud takovou ochranu potřebujeme).)

Stránkovací soubor je součástí operační paměti.

Proč vlastně koncept virtuální paměti funguje? Moderní programy obvykle obsahují části kódu i dat, které používají velmi sporadicky. Čím je program větší, tím více má funkcí a tím menší procento z nich je používáno pravidelně a často. Zbytek není potřeba udržovat neustále ve fyzické paměti. Jestliže jsou v programu málo používané části kódu, pak zřejmě také existují části dat, které jsou také málo používané. Kromě toho některé datové struktury, jako například velmi populární pole, se vždy vytvářejí s jistou rezervou, jsou tedy velmi často větší, než je potřeba. U těchto datových struktur je opět výhodné je „někam uklidit“, aby nezabíraly místo ve fyzické paměti.

Povolíme-li programům tímto způsobem plýtvat s pamětí (tj. například používat převážně jednoduchá velká pole namísto jiných složitějších struktur), programování je také jednodušší a rychlejší, což přináší ekonomickou výhodu. Přitom o plýtvání jde jen zdánlivě, neboť obsazená ale nevyužitá paměť je díky systému virtuální paměti velmi „levný artikl“. Další výhodou virtuální paměti je, že nám umožní spouštět větší množství programů současně, neboť každý potřebuje méně fyzické paměti. Lze tedy říci, že virtuální paměť prospívá jak systému, tak jeho uživatelům.

Virtuální paměť funguje stejně jako stránkování představené v předchozí kapitole, je to de facto jen rozšíření modelu stránkování. Jediné, čím se tedy v této kapitole budeme speciálně zabývat, je problematika přesunu stránek mezi fyzickou pamětí a diskem.

[4] S nadsázkou lze říci, že ať už je paměti libovolně velké množství, je jí vždy málo.

10.2 Startování a běh programu

Dále v textu budeme rozlišovat paměť primární (fyzickou paměť) a sekundární (swapovací prostor na disku). Podívejme se nejprve na startování programu, přesněji procesu. Při spouštění nového procesu můžeme buď umístit celý jeho kód do primární paměti, nebo jednotlivé stránky načítat až v okamžiku, kdy budou potřeba. Druhá varianta (tzv. demand paging, stránkování na žádost) je lepší u větších programů, které obsahují mnoho funkcí či podprogramů, z nichž ale jen některé budou použity. V praxi se proto používá tento přístup nebo nějaký kompromis, protože načítání paměti po jednotlivých malých 4KB stránkách by mohlo působit negativně na rychlost systému jako celku. (Sekvenční čtení disku je výrazně rychlejší než náhodné čtení.)

Systém si musí ve stránkovacích tabulkách evidovat, které stránky jsou v primární paměti a které jsou odswapovány, tj. přesunuty do sekundární paměti. Řešení této evidence je věcí konkrétního hardwaru, protože stránkovací tabulky používá při stránkování přímo procesor (tj. systém si tam nemůže něco pro sebe zapisovat, protože by tomu procesor pak nerozuměl). Konkrétní příklad si uvedeme později. Program běží bez problému až do okamžiku, kdy se pokusí o přístup do stránky, která je odswapovaná. V tom okamžiku nastane přerušení „page fault“ (česky: výpadek stránky), protože používat lze jen primární paměť. Potom je věcí operačního systému, aby správně ošetřil toto přerušení – chybějící stránku tedy načte do primární paměti a aktualizuje stránkovací tabulku. Je-li primární paměť plná, systém musí ještě před načtením chybějící stránky vybrat jinou stránku za oběť, kterou odswapuje. (Odtud slovo swap – výměna – stránky v primární paměti se průběžně vyměňují podle aktuální potřeby.) Po odswapování je rovněž nutno aktualizovat příslušnou stránkovací tabulku.

V dalších sekcích se budeme podrobněji zabývat jednotlivými úkoly, které systém při správě virtuální paměti má; nyní ještě zmíníme dva body, jejichž diskuzi záměrně odsouváme na později:

- Sdílená paměť – při odswapování sdíleného rámce je třeba aktualizovat všechny stránkovací tabulky, které na něj odkazují, ne jen jednu.
- Obrácená (inverted) stránkovací tabulka – při výměnách stránek je třeba umět rychle převádět rámce na stránky, k čemuž normální stránkovací tabulky nestačí.

10.3 Rezervovaná a komitovaná paměť

Jednotlivé stránky paměti mohou být buď *rezervované*, nebo *komitované*. Rezervované stránky jsou takové, které existují v lineárním adresovacím prostoru procesu, ale nikdy se do nich nezapisovalo. Každá stránka je nejprve rezervovaná. Komitovaná (z anglického committed – nepřekládáme pro snazší rozlišení od jiných termínů) stránka je taková, které je přidělen rámec v primární nebo sekundární paměti. Komitovat stránku obvykle může jen operační systém – nestačí tedy jen něco zapsat do rezervované stránky.

Většinu paměti při alokaci rezervujeme a komitujeme současně a programovací jazyky většinou ani tyto dva stavy nerozlišují; paměť pouze rezervovanou lze alokovat příslušnou funkcí operačního systému. Rezervovaná paměť se hodí především pro velká pole, která budeme používat postupně, jestli vůbec. Typickým příkladem je programový zásobník – při startu procesu lze vytvořit obrovský rezervovaný blok paměti pro zásobník. V lineárním prostoru pak existuje souvislá řada stránek, ale nejsou k ní žádné rámce.

Zásobník používá rezervovanou paměť.

10.4 Výměna rámců

Podívejme se nyní na situaci, kdy je třeba provést výměnu rámců (nebo stránek – jde o totéž, neboť stránky a rámce se vyměňují vždy současně). Jak bylo zmíněno výše, při přístupu do stránky, která není v primární paměti, musí systém nejprve najít volný rámec v primární paměti,

kam stránku umístí. Jsou-li nějaké rámce volné, použije obvykle kterýkoliv z nich (může se lišit u některých systémů). Není-li žádný volný, pak systém vybere oběť – rámec, který bude odswapován. (Výběrem oběti se budeme zabývat v následující sekci.)

Systémy si obvykle evidují také příznak změny rámce (anglicky dirty bit). Je to bit, který je nulový, když má rámec přesnou kopii v sekundární paměti, nebo jedničkový, když rámec buď vůbec v sekundární paměti není, nebo byl změněn. Tento příznak musí být podporován hardwarově, nastavuje se na jedničku při každém zápisu do rámce. (Čistě softwarové řešení není možné.) Je-li oběť změněna, pak systém okopíruje oběť do jiného rámce v sekundární paměti. Není-li sekundární paměť volná, systém tam vybere jeden rámec, který má kopii v primární paměti, nastaví příznak změny (v primární paměti) a tento sekundární rámec použije. V každém případě pak systém ještě aktualizuje stránkovací tabulku odkazující na oběť.

Průvodce studiem

Všimněte si, že každá stránka buď nemá žádný rámec, nebo má rámec v primární paměti, nebo má rámec v sekundární paměti, nebo má rámec v primární paměti, který má kopii v sekundární paměti. V každém případě však celkový počet rámců operační paměti musí být o jeden větší než počet skutečně používaných rámců, nebo by swapování nemohlo fungovat. Diskové operace jsou totiž jen čtení a zápis, nikoli výměna.

Nyní je volný rámec, systém do něj tedy okopíruje požadovaný rámec ze sekundární paměti. Eviduje si přitom, kde ve virtuální paměti je původní kopie a nuluje rámci příznak změny. Potom aktualizuje stránkovací tabulku odkazující na příslušný rámec a ukončí obsluhu přerušení page fault. Procesor po skončení přerušení opakuje poslední instrukci, čímž dojde k opětovnému pokusu o dříve nezdařený přístup do paměti. Tentokrát je paměť již aktualizována a proces pokračuje dál, aniž by vůbec zjistil, že jeho část ještě před chvílí nebyla v primární paměti.

Výměna rámce samozřejmě stojí čas. Použitím sekundární paměti lze operační paměť o hodně zvětšit („nafouknout“), měli bychom se však držet v rozumných mezích, neboť příliš časté swapování rámců by velmi zpomalilo celý počítač. Swapování lze zakázat zamknutím stránek, kdy tyto vyřadíme z výběru obětí, takže zůstávají stále v primární paměti. Zamykání by se mělo používat co nejméně, protože zamknutím části paměti způsobíme, že zbytek paměti se swapuje ještě častěji. V praxi se používá obvykle jen na jádro systému nebo ty části programů, které se používají velmi často. Opravdu nutné to však je jen u paměti sdílené s nějakým zařízením, neboť hardwarová zařízení přistupují do fyzické paměti a obcházejí tedy systém virtuální a lineární paměti. Na platformě x86 je zamknutí nutné také u rutin obsluhy přerušení, protože při výskytu jednoho přerušení nemůže nastat přerušení další (page fault) dříve, než je spuštěna obsluha toho prvního. Systém by tedy zkolaboval.

Průvodce studiem

Zpomalení při častém swapování je jediná nevýhoda virtuální paměti. Je také nutno dodat, že swapuje-li počítač hodně, tak bez virtuální paměti by v dané chvíli už to, co právě dělá, vůbec dělat nemohl. V praxi pomáhá princip lokality představený v kapitole 9.4 na straně 90 – proces totiž v krátkém čase obvykle pracuje stále se stejnou pamětí, takže nedochází k tomu, že by operační systém trávil většinu času neustálou zběsilou výměnou velkého množství rámců. Alespoň ne často.

Hardwarové limity

Při implementaci výměny rámců si skutečně vystačíme jen s operačním systémem – žádná podpora hardwaru kromě přerušení page fault a opakování instrukce není třeba. Virtuální paměť by nešlo realizovat, kdyby procesor po skončení přerušení neopakoval instrukci.

Problém však může být i s instrukční sadou. Jelikož se instrukce opakuje, je potřeba, aby částečně provedená instrukce nezpůsobila neopakovatelnost instrukce. Například procesor počítače IBM System/360 obsahuje instrukci pro kopírování až 256 znaků paměti. Pokud se adresa čtení a zápisu překrývá a zároveň je blok na hranici stránky, může nastat situace, kdy během kopírování dojde k výpadku stránky. Instrukce se pak opakuje, ale v místě zdroje dat už jsou nakopírována data nová. V tom okamžiku je výsledek chybný. Má-li procesor podobné instrukce, lze to řešit například přidáním (do hardwaru procesoru) přístupu na začátek a konec bufferu před vlastním zpracováním instrukce.

Obecně mohou nastat i jiné případy, tj. procesor může mít i jiné instrukce, jejichž provedení nelze bez újmy opakovat. Např. instrukce čtení I/O portu (hardwarového zařízení), která zapisuje hodnotu přímo do paměti při opakování opět kontaktuje zařízení a to může způsobit chybný výsledek. Proto ne na každém počítači (přesněji ne na každé hardwarové platformě) lze realizovat virtuální paměť, přestože je to funkcionalita zajišťovaná čistě softwarově.

10.5 Výběr oběti

Existuje řada algoritmů výběru oběti. Jde o čistě softwarovou záležitost, je tedy na operačním systému, jak bude postupovat. Za nejlepší považujeme takový algoritmus, kde dochází k nejmenšímu počtu výpadků stránek.

10.5.1 Optimální algoritmus

Ideální oběť je stránka, která nebude v budoucnu používána. Optimální algoritmus tedy za oběť vybere tu stránku, která v budoucnu nebude použitá, nebo pokud taková není, tak vybere tu, která bude použitá až po nejdelší době. Toto však není exaktně algoritmizovatelné, proto se v praxi používají algoritmy jiné.

Ideální oběť' nebude v budoucnu používána.

10.5.2 FIFO – first in, first out

FIFO algoritmus („první dovnitř, první ven“) za oběť vezme stránku, která je nejdéle v primární paměti. Systém udržuje evidenci o tom, která stránka byla kdy natažena do primární paměti. Stačí přitom udržovat jen frontu stránek, přesný čas totiž není podstatný. Obětí je tedy první prvek ve frontě, novou stránku dáváme na konec fronty. Tento algoritmus je tedy velmi jednoduchý, není však kvalitní. Minimálně jádro systému, které je často používáno, bude trpět tím, že jednou za čas bude přesunuto do sekundární paměti (a nejspíše hned zase zpět).

FIFO algoritmus výběru oběti dokonce trpí *Beladyho anomálií* – v některých případech se může stát, že dá-li systém procesu k dispozici víc fyzické paměti, dojde při běhu k většímu počtu výpadků stránek. K této anomálii dochází u některých algoritmů; není tedy pravda, že když máme více fyzické paměti, snížíme automaticky počet výpadků stránky.

Příklad Beladyho anomálie na FIFO algoritmu [SGG05]: Proces přistupuje ke stránkám v pořadí 123412512345. Máme-li tři rámce, pak dojde k devíti výpadkům stránky (označeny svislou čarou): |1|2|3|4|1|2|512|3|45. Máme-li čtyři rámce, pak dojde k desíti výpadkům stránky: |1|2|3|412|5|1|2|3|4|5.

10.5.3 LRU – least recently used

LRU algoritmus vezme za oběť stránku, která je nejdelší dobu nepoužívaná. Vychází tedy z předpokladu, že co platilo v nedávné minulosti, to bude platit i v budoucnosti. Tím se snaží napodobit optimální algoritmus popsaný v sekci 10.5.1.

Systém udržuje čas posledního použití u každého rámce a podle toho pak vybírá oběť. LRU můžeme považovat za dobrý algoritmus, ovšem je otázka, jak jej implementovat, protože není jednoduché sledovat časy všech přístupů do paměti. Každopádně to vyžaduje hardwarovou implementaci.

První možností implementace je použít počítadlo v procesoru, které bude inkrementováno při provedení každé instrukce, a jeho hodnota bude uložena jako příznak do stránky při každém přístupu k ní. V okamžiku hledání oběti pak projdeme všechny stránky a za oběť vybereme tu s nejmenším číslem. Druhou možností je použít pseudo–zásobník stránek, kde při každém přístupu dáme dotyčnou stránku na vrchol zásobníku a za oběť bereme stránku na dně zásobníku. (Zásobník je jen „pseudo“ proto, že odebíráme prvky z prostředku zásobníku, což normálně nejde.)

Podobně jako optimální algoritmus, LRU také netrpí Beladyho anomálií. (Dokažte!) Jeho implementace kterýmkoliv ze dvou popsaných způsobů by ovšem vyžadovala složitou hardwarovou podporu a vedlejším efektem by bylo zpomalení každého přístupu do paměti z důvodu udržování evidence.

10.5.4 Přibližné LRU

Málokterá hardwarová platforma podporuje LRU. Některé platformy nepodporují výběr oběti nijak, potom si musíme vystačit například s algoritmem FIFO. Některé systémy nabízejí alespoň částečnou podporu, potom můžeme realizovat algoritmus výběru alespoň přibližně odpovídající LRU. Ona částečná podpora má podobu bitu přístupu (reference bit), který je u každé stránky a je nastaven na jedničku při každém přístupu ke stránce. (Vzpomeňte na bit dirty (viz sekce 10.4 na straně 99, toto je obdobné.) Bit přístupu je nejprve nastaven u všech stránek na nulu a při běhu procesu se u používaných stránek postupně nastavuje na jedničku. Nevíme tedy sice, kdy byla která stránka naposledy použitá, díky hardwaru ale víme, jestli byla použitá. Toto je pak základem různých algoritmů výběru oběti.

Algoritmus dalších bitů přístupu

Tento algoritmus zvyšuje rozlišovací schopnost mezi použitými stránkami tak, že pracuje s vícebitovým údajem o použití, čili např. „přístupový bajt“ (reference byte), kde hardware nastavuje nejvyšší bit a systém pravidelných časových intervalech provede bitový posun doprava u každé stránky. V okamžiku oběti tedy je k dispozici statistika použití stránek za 8 posledních časových úseků. Přístupový bajt lze nyní chápat jako neznaménkové číslo a za oběť je vybrána ta stránka, která má nejmenší číslo. Je-li takových stránek nalezeno více, lze z nich vybrat např. FIFO algoritmem.

Počet přídavných bitů, které algoritmus používá, samozřejmě může být různý. V extrémním případě, kdy to je nula bitů, se z tohoto algoritmu stává algoritmus druhé šance.

Algoritmus druhé šance

Tento algoritmus nepoužívá žádné přídavné bity. Základem je FIFO algoritmus. Před odswapováním stránky na začátku fronty však testujeme její bit přístupu. Je-li jedničkový, pak dáme stránce druhou šanci: nulujeme bit přístupu a dáme ji na konec fronty. Odtud tedy název algoritmu: je to modifikace FIFO algoritmu, kde každá v poslední době používaná stránka dostane druhou šanci.

Tento algoritmus se implementuje pomocí kruhové fronty. Stránky zůstávají stále ve stejné frontě a posouvá se jen ukazatel. Kandidátem na oběť je stránka, na které je ukazatel. Má-li však tato stránka jedničkový bit přístupu, tak se tento bit vynuluje a přejde na další stránku a

to je nový kandidát. Nejpozději po projití celé kruhové fronty tedy najdeme skutečnou oběť. Ukazatel se pak posune na další stránku v řadě.

Algoritmus druhé šance lze rozšířit ještě o použití bitu dirty. Získáme tak vlastně dvoubitovou hodnotu reference+dirty a za oběť vybíráme přednostně ty stránky, které mají v těchto dvou bitech nejmenší číslo (bráno jako dvoubitová celočíselná hodnota). Nevýhodou je, že v extrémním případě musíme několikrát projít celou kruhovou frontu, než oběť určíme. Výhodou je, že oběť je brána přednostně ze stránek, které nebyly modifikovány, takže ušetříme čas tím, že nebudeme oběť kopírovat do sekundární paměti.

10.5.5 LFU – least frequently used

LFU algoritmus počítá počet přístupů ke každé stránce a za oběť bere tu, kde počet přístupů byl nejmenší. Algoritmus stojí na předpokladu, že aktivně používané stránky, které odswapovat nechceme, budou mít napočítáno hodně přístupů. Tento algoritmus typicky selhává v situacích, kdy na začátku proces provede inicializační kód, díky kterému se napočítá v určitých stránkách hodně přístupů, ale nikdy později se tyto stránky už nebudou používat. Pomůckou pak může být obdoba výše uvedených postupů, kdy systém v pravidelných intervalech prochází stránky a počítadla přístupů jim posouvá o jeden bit doprava.

Jinou alternativou může být i zcela opačný postup MFU (most frequently used). Ten stojí na předpokladu, že nejméně používané stránky byly zřejmě teprve nataženy do paměti a teprve se začnou používat. Pokud bychom je odswapovali, způsobíme tak vlastně neustálé swapování stejných stránek pryč z paměti a zpět, protože v nich poběží kód, ale budou mít stále málo přístupů a při každém odswapování se jim znovu vynuluje počítadlo.

Je vidět, že LFU i MFU mají jisté nepříjemné nedostatky, ve skutečných operačních systémech se používají varianty přibližného LRU.

10.5.6 Buffer volných rámců

Algoritmy pro výběr oběti lze doplnit o další pomocné činnosti. Např. je výhodné udržovat seznam volných rámců. Při výpadku stránky je tato stránka načtena do volného rámce a proces nemusí čekat, než je oběť zapsána na disk. Oběť je ale i tak vybrána a s jistým zpožděním odswapována. Její rámec je pak přidán do seznamu volných. Rozšířením této myšlenky je udržování seznamu modifikovaných stránek. Jakmile nemá procesor nic na práci, může modifikované stránky zapisovat na disk a mazat dirty bit. Tímto způsobem systém zvyšuje šanci, že až bude daná stránka vybrána za oběť výměny, ušetří čas, protože ji nebude muset znovu zapisovat na disk.

Další variantou téhož je udržovat seznam volných rámců spolu s čísly stránek, které v nich naposledy byly. Jelikož ani volný rámec, ani stránka na disku nemůže být změněna, při potřebě vrátit odswapovanou stránku zpět do paměti se vůbec nemusí číst disk – systém jen opět aktivuje původní rámec. Tento princip používá systém VAX/VMS spolu s obyčejným FIFO algoritmem výběru oběti (VAX nemá k dispozici dirty bit). Zatímco FIFO nevhodně vybírá za oběť i často používané stránky, díky bufferu volných stránek se každá oběť může vrátit zpět k použití bez čtení disku, pokud je použita dostatečně brzo poté, co byla odswapována. Tuto pomůcku používají i některé implementace Unixu jako doplnění algoritmu druhé šance. [SGG05]

10.6 Přidělování rámců

Připomeňme, že stránkování na žádost, které jsme popsali v předchozích sekcích, přiděluje rámce stránkám až v případě potřeby. Při běhu systému jako celku tedy systém postupně řeší

požadavky procesů tak, jak přicházejí přes výpadky stránky. Tento jednoduchý princip je jistě funkční a zvláště na jednoúlohových systémech ani nic lepšího nepotřebujeme hledat. Lze jej kombinovat i s bufferem volných rámců popsaným v předchozí sekci. Shrnuto do jedné věty: Všechny rámce jsou si rovny a každý proces může v případě potřeby dostat kterýkoliv z nich.

V dalších odstavcích se podíváme na algoritmy pracující s tezí, že systém bude fungovat lépe, když si nebudou všechny rámce rovny.

10.6.1 Minimální počet rámců

Pokud klesne počet rámců jednoho běžícího procesu pod rozumnou hranici, bude docházet k velkému množství výpadků stránky a to zpomalí chod celého systému. Počet rámců musí také u každého procesu být alespoň roven počtu rámců, které může použít jedna instrukce, jinak by systém mohl skončit v nekonečné řadě výpadků stránky. (Na x86 např. pro instrukci `movsd` a hranici stránek je potřeba 2 stránky pro instrukční kód, 2 pro čtení a 2 pro zápis dat, celkem tedy až 6 stránek. Navíc potřebujeme přístup do stránek se stránkovacími tabulkami, kde mohou vniknout i vícenásobné odkazy (samotná stránkovací tabulka je ve stránce, kterou potřebujeme najít pomocí jiné stránkovací tabulky). Je proto žádoucí neustále hlídat, aby každý proces měl alespoň jisté minimální množství rámců pro sebe.

10.6.2 Alokační algoritmy

Rámce můžeme dělit mezi procesy buď rovnoměrně – každý proces dostane stejný počet rámců, nebo poměrně (proporcionálně) vzhledem k velikosti virtuálního adresovacího prostoru každého procesu – procesy používající větší virtuální paměť pak dostanou víc rámců. V obou případech se aktuální počty přidělených rámců průběžně mění podle toho, jak v systému přibývá nebo ubývá běžících procesů.

Všimněte si také, že procesy s vyšší prioritou mají stejné podmínky jako procesy s nižší prioritou. V praxi však může být žádoucí, aby procesy s vyšší prioritou měly také jakousi de facto vyšší prioritu co do přidělování rámců. To lze realizovat např. přidělováním rámců poměrně vůči prioritě, namísto velikosti virtuálního adresovacího prostoru.

Další možností je *globální alokace rámců*, kdy procesy sice na začátku dostanou určitou množinu rámců pro sebe (dle nějakých pravidel, jak bylo uvedeno výše), ale při výpadcích stránky je možno za určitých podmínek přesouvat rámce mezi procesy. Např. stanovíme, že proces s vyšší prioritou může dostat rámec na úkor procesu s nižší prioritou. Při výpadku stránky je pak oběť hledána mezi rámci aktuálního procesu a všemi procesy nižší priority.

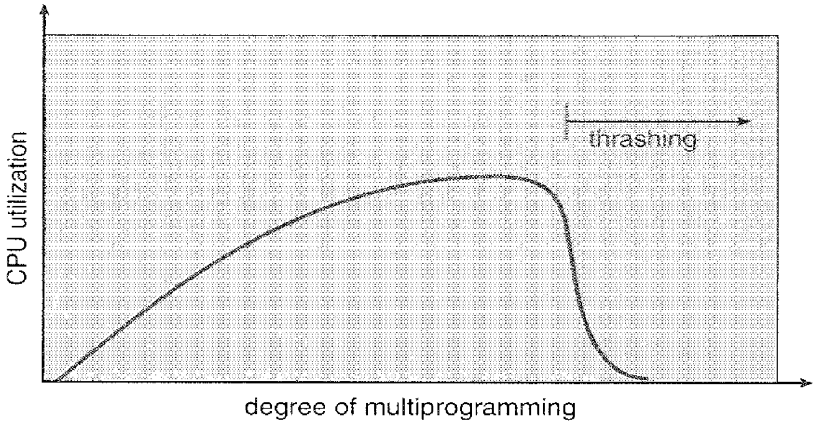
Při globální alokaci dochází k tomu, že opakovaně spouštěný program může běžet různě dlouho, protože pokaždé na něj vyjde jiné množství rámců. (Čím méně má rámců, tím je jeho výpočet pomalejší.) Při lokální alokaci k tak velkým rozdílům nedochází, ale výpočetní výkon systému jako celku je omezen tím, že systém nedokáže efektivně využívat všechny rámce dle aktuálních potřeb procesů. Globální alokace je proto z hlediska systému jako celku výhodnější a je v praxi preferována.

10.7 Thrashing

Anglické slovo thrashing označuje situaci, kdy zvýšení počtu prostředků snižuje výkon systému. Tento paradox si nejlépe představíme pomocí příkladu. Trvá-li postupný výpočet 10 procesů 10T, pak spustíme-li je současně na počítači o deseti procesorech, měl by trvat 1T. Pokud však všechny procesy pracují s CD diskem, pak výsledný čas bude ve skutečnosti mnohonásobný – při přístupu k CD bude docházet k neustálému přesunu čtecí hlavičky na jiná místa. Každý proces si přečte malý kousek a jiný proces mu opět hlavičku přesune jinam.

Při stránkování thrashing nastává, když má proces velký virtuální adresní prostor, ale jen malý počet rámců. Při práci tak neustále vyměňuje své stránky ve svém malém fyzickém prostoru. Nastane-li taková situace, běh procesu se kvůli neustálému swapování extrémně zpomalí a obvykle to zpomalí i systém jako celek. Thrashing je tedy název pro situaci, kdy čím více paměti má program, tím běží pomaleji.

Předchozí příklady jsou extrémní a lze se jim často vyhnout jednoduše tak, že nebudeme dělat „hloupé věci“. Thrashing však může vzniknout i při běžném provozu jako důsledek přetížení systému. Má-li procesor na serveru malé vytížení (což operační systém umí sledovat), pak systém startuje další úlohy čekající ke zpracování. Jak se počet běžících procesů zvyšuje, stoupá i vytížení procesoru. Pokud velikost fyzické paměti neodpovídá výpočetnímu výkonu procesoru, pak při startování dalších procesů postupně každý z nich má méně a méně rámců, až dojde k thrashingu – procesy začnou hodně swapovat a vytížení procesoru opět klesá. Systém opět ve snaze o vytížení procesoru startuje další úlohy a tím dále klesá vytížení procesoru. Tato situace vede k paradoxu: Systém odvádí téměř nulovou práci, neboť je přetížen prací. Viz obrázek 16. Aby procesor začal víc pracovat, je paradoxně třeba mu od práce odlehčit, čili snížit počet současně prováděných úloh.



Obrázek 16: Thrashing – s rostoucím počtem procesů klesá vytížení procesoru. [SGG05]

Thrashing lze částečně řešit použitím lokálního alokačního algoritmu zmíněného v předchozí sekci. Zakážeme tím, aby jeden přetížený proces „kradl“ rámce ostatním procesům a tím je také přetížil. Nevyřešíme tím však samotný problém – má-li jeden proces málo rámců, bude hodně swapovat a ovlivní tak celkový chod systému minimálně tím, že bude zatěžovat swapovací systém svými požadavky a ostatní procesy budou muset déle čekat na vyřízení těch jejich.

Skutečným řešením thrashingu je přidělovat procesům tolik rámců, kolik skutečně potřebují. Potřeby procesů se samozřejmě během času mění, je proto potřeba je nějakým způsobem průběžně zjišťovat.

10.7.1 Pracovní množina rámců (working set)

Kolik rámců proces potřebuje lze průběžně zjišťovat např. skrze jejich tzv. pracovní množiny rámců. Tento algoritmus je postaven na *teorii lokalit*, která je odvozená od principu lokality, viz kapitolu 9.4 na 90. Platí-li princip lokality, pak program během času pracuje v různých lokalitách, které se mohou překrývat. V každém okamžiku je tedy proces v nějaké lokalitě a pracuje s množinou stránek patřící do této lokality. Má-li program k dispozici tolik rámců, kolik je stránek v aktuální lokalitě, pak nedochází ke swapování a potažmo thrashingu. Má-li program rámců méně, pak dochází k thrashingu a výpočet se velmi zpomalí. Má-li jich naopak více, pak je snížena efektivita využití paměti v systému jako celku (důsledkem je, že méně procesů může běžet současně).

Pracovní množinu lze v praxi zjišťovat či odhadovat algoritmem využívajícím kruhovou frontu obdobně jako algoritmus druhé šance (kapitola 10.5.4 na straně 101). Systém s pomocí časovače v pravidelných intervalech prochází stránky procesu a zjišťuje, kolik jich má nastaveno bit přístupu, což je de facto velikost pracovní množiny. Zároveň bit přístupu nuluje, takže je vždy nastaven jen u stránek, které byly použity od předchozí kontroly. Podle zjištěné velikosti pracovní množiny pak systém upravuje počty rámců přidělených jednotlivým procesům. Potřebuje-li proces další volné rámce, tak je buď dostane (jsou-li k dispozici), nebo je naopak celý proces pozastaven a odswapován (není-li dost rámců). V případě, že by hrozil thrashing je tedy celý proces raději odsunut do sekundární paměti, než aby thrashingem zatěžoval celý systém. Zároveň tím systém získá volné rámce pro jiné procesy.

Algoritmus pracovní množiny dokáže eliminovat thrashing a tím zefektivnit využití procesoru, protože ten nebude často čekat na vyřízení výměny rámců. Nevýhodou je nebezpečí hladovění – procesy s velkými pracovními množinami budou pravděpodobně jako první odswapovány a bez dalšího doplnění algoritmus nezaručuje, že při neustálém zatížení systému každý proces vůbec v konečném čase skončí.

10.7.2 Frekvence výpadků stránky

S thrashingem lze bojovat i jednodušší cestou než výpočetně náročnými výpočty pracovních množin. Thrashing se projevuje velkým množstvím výpadků stránek (v podstatě v řadě za sebou). Lze jej tedy detekovat pouhým sledováním počtu výpadků za jednotku času. Můžeme tedy stanovit dolní a horní mez rozumného počtu výpadků za jednotku času a pomocí časovače pravidelně sledovat, zda všechny běžící procesy jsou uvnitř tohoto intervalu. Pokud některý nebude, lze počet jemu přidělených rámců upravit tak, jak bylo popsáno v předchozí sekci (rámec přidáme či odebereme, případně pozastavíme proces). Tato metoda je výpočetně méně náročná než sledování pracovních množin.

10.8 Mapování souboru do paměti

O mapování souboru do paměti jsme se již zmínili v souvislosti s copy–on–write v kapitole 9.7 na straně 93. Jde o funkci, která zpřístupní soubor (z disku) tak, že se „objeví“ někde v paměti. Práce s mapovaným souborem je jednodušší než alokovat paměť a ručně tam načítat soubory. Hlavní výhodou však je, jak už bylo zmíněno, že mapované soubory lze snadno sdílet mezi procesy, a to dokonce i tehdy, když některý z nich chce svou kopii částečně měnit. Práce s pamětí je také algoritmicky jednodušší než práce se souborem (proces vidí mapovaný soubor jako obyčejné pole bajtů), takže mapování zjednodušuje programy a práci programátorům.

Mapovaný soubor nezabírá paměť, dokud s ním proces nezačne pracovat. Teprve po výpadku stránky se načítají ty části souboru, které proces ve skutečnosti používá. Podobně zápis do mapovaného souboru nejde přímo na disk, takže proběhne velmi rychle a skutečný zápis pak je proveden se zpožděním (buď v okamžiku, kdy systém má volný čas, nebo při ukončení mapování). Jedinou nevýhodou mapování je, že jej z principu nelze použít pro soubory větší, než je maximální velikost virtuální paměti (na 32bitových systémech typicky 2–4GB).

Realizace v NT

Podívejme se nyní, jak se mapování souboru dělá ve Windows NT. Mapování zde probíhá dvoufázově: Nejprve vytvoříme mapování souboru do paměti, a pak vytvoříme pohled na tento soubor. Více procesů si může vytvořit pohledy na stejný soubor, čímž získají sdílenou paměť. Tato metoda je jediným způsobem, jak lze v NT sdílet paměť. (Sdílení kódu, které systém zajišťuje automaticky, funguje také na bázi mapování souboru.) Používají-li procesy mapovaný

soubor pro komunikaci, je vhodné doplnit to ještě o mutex či jiný synchronizační prostředek pro sledování, kdy má kdo číst nebo zapisovat.

Soubor nejprve otevřeme či vytvoříme pomocí **CreateFile**. Handle, který takto získáme, použijeme k vytvoření mapování funkcí **CreateFileMapping**. Tím získáme pojmenovaný sdílený objekt, na který se pomocí jména mohou odkazovat libovolné procesy. Můžeme tedy vytvářet pohledy na soubor pomocí **MapViewOfFile**. Díky rozdělení mapování na dvě fáze můžeme do adresovacího prostoru procesu namapovat jen část souboru. Systém přitom zajišťuje sdílení mapovaného souboru, bez ohledu na to, kolik procesů jej používá a které jeho části si každý z nich mapuje. (Tento princip je tím efektivnější, čím více procesů pracuje s jedním souborem a čím menší pohledy tyto procesy používají.)

Po skončení práce nejprve každý proces zruší svůj pohled pomocí **UnmapViewOfFile** a zavře všechny handly pomocí **CloseHandle**. (Zavření handlů samozřejmě lze provést i dříve – kdykoliv, kdy už nebudou potřeba.)

10.9 Další pasti a pastičky

Ačkoliv tato a předchozí kapitola nejsou zrovna stručné, stále popisují systém správy paměti jen rámcově. Pojďme se na závěr stručně seznámit s několika dalšími zajímavými tématy.

10.9.1 Převod rámců na stránky

Například jsme nezmínili problém převodu rámců na stránky. Systém musí umět převést číslo rámce na číslo stránky, protože hardwarová zařízení pracují s fyzickými adresami (pochopitelně, nepatří přece žádnému procesu, takže nemají virtuální adresový prostor, ale vidí paměť v její fyzické podobě). Tento převod rámců na stránky je vlastně inverzní operací ke stránkování – kdyby měl systém hledat každé propojení ve stránkovacích tabulkách, bylo by to velmi pomalé. Navíc při sdílení paměti je třeba umět namapovat rámec na všechny stránky, které se na něj odkazují. Operační systém proto ve skutečnosti musí udržovat ještě tzv. obrácenou (invertovanou) stránkovací tabulku.

Potřebuje-li proces pracovat s hardwarovým zařízením (a to není vůbec okrajové, protože minimálně systémové procesy a ovladače zařízení to dělají velmi často), vyžaduje od systému nejen fyzické adresy, ale také souvislost přidělených bloků paměti. Potřebujeme-li např. pro zvukovou kartu 64KB paměti, pak je musíme dostat v jednom souvislém bloku, ne jako sadu 16 náhodně nalezených volných 4KB rámců. Operační systém tedy musí umět také vyhovět takto specifickým požadavkům.

10.9.2 Paměť pro objekty jádra

Samotné jádro operačního systému také používá nějakou paměť, nejčastěji potřebuje alokovat poměrně malé bloky pro systémové struktury. (Řada systémových funkcí pracuje s datovými strukturami o délce několika desítek či stovek bajtů.) Bylo by značně nehospodárné alokovat tyto struktury přímo stránkovacím systémem, protože většina z 4KB stránek by byla nevyužita (některé systémy navíc mají stránky ještě větší). Operační systém tedy používá pro své jádro jiný algoritmus správy paměti, který může být podobný např. haldě dynamické paměti jazyka C, i když toto řešení není zrovna nejefektivnější z hlediska rychlosti. Jelikož jádro systému alokuje paměť pro datové struktury, které velmi připomínají objekty, může to dělat v zásadě libovolným způsobem, který se osvědčil v objektově orientovaných programových prostředích. Proto toto téma nebudeme dále diskutovat.

10.9.3 Velikost stránky

Dalším tématem, které jsme nediskutovali, je volba velikosti stránky. U některých počítačů nemají tvůrci operačních systémů na výběr a musejí používat tu velikost, kterou hardware počítače podporuje. Při návrhu zcela nových počítačů nebo také na flexibilních platformách (kam spadají i nejrozšířenější x86, x64 a PowerPC) je však možnost minimálně z několika možností vybírat. V literatuře se obvykle jako výchozí nebo vzorová velikost používá 4KB, protože to je nejčastěji používaná velikost v praxi (rozšířeno díky procesoru Intel 80386 a kompatibilním). Zvětšíme-li velikost stránky, zmenšíme stránkovací tabulky. Jelikož každý proces má svou sadu stránkovacích tabulek, můžeme tím ušetřit i slušné množství paměti. Naopak efektivita využití paměti s rostoucí velikostí stránek klesá. Z hlediska (vnitřní) fragmentace je výhodnější mít stránky spíše menší, aby při alokaci mohlo být jednotlivým požadavkům vyhověno pomocí bloků paměti o velikosti co nejbližší požadované velikosti. Vždy platí, že průměrná fragmentace je rovna polovině velikosti stránky (na blok).

Průvodce studiem

Otázka volby velikosti stránky platí zcela stejně i pro volbu velikosti sektoru či clusteru na externím paměťovém zařízení (hard disku apod.). Toto téma budeme probírat v kapitole 12.6 na straně 133.

S rostoucí celkovou velikostí paměti je vhodné zvětšovat i stránky. Z hlediska swapování ovlivňuje chod systém často více přístupová doba než rychlost čtení či zápisu. Běžná přístupová doba je kolem 8ms, zatímco rychlost čtení je minimálně 20MB/s. Načtení jedné 4KB stránky tedy trvá přibližně 0.2ms, což je 40× méně, než je průměrná přístupová doba. Každé načtení jedné stránky z disku tedy trvá zhruba 8ms bez ohledu na to, zda má stránka 4KB nebo třeba 64KB. Z tohoto důvodu by bylo z hlediska swapování vhodnější používat stránky větší. Pro menší stránky naopak hovoří menší fragmentace a také lepší rozlišovací schopnost lokalit (jsou-li stránky menší, pak jsou menší i lokality, takže proces vystačí s menším množstvím fyzické paměti bez thrashingu). Celkově však skutečně s rostoucí celkovou velikostí paměti roste vhodnost větších stránek.

Velikost paměťových stránek souvisí i s rychlostí systému. Jelikož procesor má jen omezenou velikost TLB (viz kapitolu 9.3 na straně 88), paměť přímo přístupná bez cache miss je přímo úměrná velikosti stránky. Jsou-li tedy stránky větší, máme větší pravděpodobnost, že se celá aktuální lokalita procesu vejde do TLB. V opačném případě pak dochází k častému cache miss a kvůli prohledávání stránkovacích tabulek se zpomaluje výpočetní výkon. Některé platformy a jejich operační systémy umožňují používat různé velikosti stránek, podle potřeby. Např. procesory UltraSparc podporují stránky o velikosti 8, 64, 512 a 4096KB. Operační systém Solaris z toho používá 8KB a 4MB. Kód jádra systému je umístěn v 4MB stránkách, takže přístup do jádra významným způsobem šetří TLB. Stejným způsobem funguje i Windows NT na běžných „písíčkách“.

10.9.4 Swapování stránkovacích tabulek, zamykání stránek

Další zajímavostí je swapování stránkovacích tabulek. Jelikož stránkovací tabulky jsou umístěné zase jen ve stránkách, mohou být při nepoužívání také odswapovány. Dojde-li potom k výpadku stránky, může se stát, že prohledávání stránkovací tabulky může vést k dalšímu výpadku. A jelikož jsou stránkovací tabulky víceúrovňové, výpadků může být při jednom přístupu do paměti i několik (až o jeden více, než je počet úrovní tabulek). Operační systém musí však se stránkovacími tabulkami nakládat opatrně, aby nedošlo k situaci, kdy stránkovací tabulka popisuje sama sebe a je odswapována. V tom okamžiku by takový proces nemohl pokračovat. Systém tedy musí umět klíčové stránkovací tabulky, které nesmí být odswapovány, zamknout.

Zamykání, na druhé straně, má tu nevýhodu, že může vést ke zhroucení systému. Stačí menší chyba v kódu procesu, nebo dokonce v kódu nějaké části samotného systému, a zamknutá paměť už nebude nikdy odemknuta. Taková paměť je nevyužitelná. Stane-li se to na systémech, kde pracuje současně více uživatelů, jednotliví uživatelé se budou ovlivňovat (jeden zamkne paměť a ostatním tak znepříjemní práci). Některé operační systémy primárně počítající s víceuživatelským nasazením (jako Solaris a další serverové systémy) proto vůbec zamykání paměti na úrovni uživatelských procesů nepodporují, nebo pouze umožňují, aby procesy doporučily zamykání. Nemohou ho však po systému vyžadovat.

Průvodce studiem

Zde vidíme typický rozdíl mezi operačními systémy unixového typu a systémy Windows. Zatímco ve Windows si může proces sám nastavit nejvyšší prioritu a zamknout celou paměť, čímž efektivně znemožní další chod počítače, na unixových serverech toto není možné. Na obranu Windows je však třeba dodat, že pomocí správného nastavení úrovně oprávnění procesů lze jednotlivým procesům zakázat provádění potenciálně destruktivních operací.

Shrnutí

Tato kapitola byla věnována virtuální paměti. Navázali jsme na povídání o správě paměti z předchozí kapitoly. Model virtuální paměti je pouze rozšířením modelu stránkování, jeho implementace však vyžaduje řešit celou řadu důležitých detailů.
Studium tématu virtuální paměti sestává de facto z mnoha střípků, více či méně nezávislých, které dohromady skládají mozaiku moderního efektivního způsobu organizace a správy paměti.
V této kapitole jsme probrali problematiku výběru oběti při swapování, rozdíly mezi rezervovanou a komitovanou pamětí, globální a lokální přidělování rámců, thrashing, mapování souborů do paměti a v závěru jsme ještě zmínili několik dalších doplňkových témat a problémů, se kterými se tvůrci systémů potýkají.
V následující kapitole si ukážeme, jak funguje správa operační paměti v praxi.

Pojmy k zapamatování

- virtuální paměť
- swapování
- stránkovací či swapovací soubor
- stránkování na žádost (demand paging)
- obrácená (inverted) stránkovací tabulka
- paměť primární a sekundární
- rezervovaná a komitovaná paměť
- výpadek stránky
- výměna rámců
- výběr oběti (mezi rámci)
- ideální oběť
- Beladyho anomálie
- algoritmy výběru oběti: FIFO, LRU, přibližné LRU, LFU
- lokální a globální alokace rámců
- thrashing
- pracovní množina rámců
- teorie lokalit
- mapování souboru do paměti

Kontrolní otázky

1. *Co musí hardware podporovat, aby bylo možno realizovat virtuální paměť a demand paging?*
2. *Co je příčinou thrashingu? Jak jej systém může detekovat?*
3. *Za jakých okolností nastávají výpadky stránek? Popište, jak na ně systém reaguje.*
4. *Mějme log obsahující posloupnost čísel stránek, ke kterým daný proces přistupoval. Proces má k dispozici a rámců, na začátku prázdných. Log má b položek obsahujících c různých hodnot. Uveďte dolní a horní mez počtu výpadků stránek.*
5. *Uveďte, které z následujících programovacích technik a struktur jsou výhodné při stránkování na žádost (demand paging):*
   - *zásobník*
   - *hashovací tabulka*
   - *sekvenční hledání*
   - *binární hledání*
   - *čistý (lineární) kód*
   - *operace nad vektory*
   - *nepřímé odkazy (práce s pointery atp.)*
6. *Seřaďte následující algoritmy výběru oběti od nejlepšího po nejhorší: LRU, FIFO, optimální oběť, druhá šance.*
7. *Popište různé formy přibližného LRU.*
8. *Mějme dvojrozměrné pole* `byte A[] = new byte[100][100]`, *které začíná na adrese 200. Dále předpokládejme stránkovací systém s velikostí stránky 200, algoritmem LRU a celým programem ve stránce 0.*
   Předpokládejme, že stránka 0 je načtena v paměti a ostatní rámce jsou prázdné. Kolik výpadků stránky zaznamenáme u následujících dvou variant inicializace pole A?

   (a)
   ```
   for each i {
     for each j {
       A[i][j] = 0;
     }
   }
   ```

   (b)
   ```
   for each j {
     for each i {
       A[i][j] = 0;
     }
   }
   ```
9. *Předpokládejme systém používající demand paging a pevný počet souběžných procesů, u kterého bylo změřeno využití CPU a stránkovacího disku. Pro následující tři alternativy uveďte, co se v systému děje. Lze přidáním dalších procesů zvýšit využití CPU? Je možnost stránkování přínosem?*
   (a) *Využití CPU 12%, aktivita disku 95%*
   (b) *Využití CPU 88%, aktivita disku 4%*
   (c) *Využití CPU 15%, aktivita disku 5%*
10. *Paměť se používá pro kód a data. Je některá z těchto forem využití lepším kandidátem pro stránkování a odswapování než ta druhá? Svou odpověď zdůvodněte.*
11. *Nejběžnějším uživatelem rezervované paměti je programový zásobník. Vysvětlete, proč právě zásobník.*
12. *Je možné, aby přidáním dalších procesorů (CPU) do serverového počítače systému klesl jeho celkový výkon? Vysvětlete.*

11 Správa paměti v praxi

Studijní cíle: V této kapitole si ukážeme, jak funguje správa paměti v praxi. V předchozích dvou kapitolách jsme kompletně probrali téma správy operační paměti, nyní se podíváme na technické detaily a řešení na běžných počítačích a operačních systémech.
Seznámíme se jmenovitě se správou paměti v systému Windows NT na nejrozšířenější platformě x86. Jelikož správa paměti hodně závisí na hardwarové platformě, řada věcí platí analogicky i pro jiné operační systémy. Podíváme se také na překlad virtuálních adres na fyzické (ten probíhá zcela hardwarově).

Klíčová slova: překlad adres, stránkování, PAE, AWE, x64, ochrana paměti

Potřebný čas: 230 minut.

11.1 NT na platformě x86

Podobně jako jsme v kapitole 6 na straně 49 diskutovali konkrétní řešení správy procesoru na nejběžnějších platformách, podíváme se nyní na to, jak je v praxi řešena správa paměti. Na rozdíl od správy procesoru, kde třeba systém priorit a plánování jsou čistě softwarové záležitosti, správa paměti se hodně týká i hardwaru, neboť jedním z úkolů, které operační systém má, je příprava stránkovacích tabulek, které pak používá hardware pro automatický překlad virtuálních adres na fyzické při běhu programu. Proto se do jisté míry správa paměti na stejné platformě ve všech operačních systémech podobá. My se zaměříme především na Windows NT a platformu x86.

11.1.1 Základní fakta

Procesory x86 existují v několika verzích a podporují různé režimy práce. Systém Windows NT je však možno provozovat pouze v 32bitovém chráněném režimu flat, který je dostupný na procesorech 80386 a výše.

Režim je 32bitový, čili procesor má 32bitové registry a to jak pro výpočty, tak pro adresaci. Přímo adresovatelný prostor je tedy 32bitový. Režim je chráněný, tj. instrukční sada je rozdělena na instrukce běžné, které lze vykonávat každým procesem, a privilegované, které mohou vykonávat jen privilegované procesy (tedy jádro systému). Připomeňme také, že procesy, či přesněji vlákna, při běhu samy vykonávají kód jádra a to tak, že se jim průchodem přes speciální brány mění úroveň privilegií. Na problematiku bran se podíváme později, zatím vystačíme s tím, že privilegia vlákna při běhu se mění podle toho, zda vykonává kód v segmentu patřícím jádru systému, nebo v segmentu uživatelského procesu.

Jak jsme právě zmínili, systém používá segmentaci paměti, avšak pouze minimálně. Segmenty de facto slouží pouze k ochraně paměti – každý segment má nastavenou úroveň požadovaných privilegií procesoru. Z aktuálního segmentu kódu lze volně volat jen ty segmenty, které jsou na stejné úrovni. Volání do vyšší či nižší úrovně probíhají zvláštním způsobem. (K tématu ochrany se vrátíme později.)

2GB uživatel, 2GB systém.

Běžný proces tedy může adresovat 32 bitů lineární virtuální paměti, tj. 4GB. Horní polovina je vyhrazena pro operační systém, takže pouze dolní 2GB jsou pro uživatelský proces. (Vyplývá z toho, že horní 2GB paměti se vždy sdílí mezi všemi procesy, tedy pokud mají oprávnění k jednotlivým oblastem paměti přistupovat.) Do horní poloviny paměti se např. také mapují hardwarová zařízení (paměť grafické karty atp.).

Systémy Windows NT 4.0 Enterprise, Windows 2000 Advanced Server, Windows XP a novější mohou být při startu spuštěny také ve zvláštním režimu, kdy aplikace mají až 3GB a systém jen 1GB. V tomto řežimu sice mohou některé aplikace a hlavně ovladače zařízení mít problémy s kompatibilitou, ale umožní nám to pohodlně využít o 50% více paměti (což je pro servery

velký rozdíl). (Pro hladký chod 3GB režimu je však obvykle nutné vyladit ještě řadu dalších systémových hodnot, protože jinak bude mít systém příliš malé systémové tabulky pro celou řadu důležitých prostředků, takže pravděpodobně vůbec nebude schopen užitečně pracovat. Toto je možné udělat na Windows Serveru 2003.)

11.1.2 AWE

AWE zpřístupní procesu více než 2GB paměti.

Dalším rozšířením je rozhraní AWE (Address Windowing Extensions), které umožňuje pomocí mapování do okének zpřístupnit více než 2GB paměti pro proces. Toto rozhraní musí jednotlivé aplikace použít explicitně, tj. je to sada speciálních systémových funkcí, které dělají obdobnou věc jako mapování souboru. Do vymezeného úseku paměti, „pohledu“, zobrazí kus paměti, který by jinak nebyl přístupný.

Celkovou dostupnou paměť pro aplikaci však i při použití AWE limituje ještě schopnost systému adresovat fyzickou paměť. Běžné verze Windows pak i s AWE poskytují aplikacím jen 4GB paměti. Microsoft navíc používá obchodní politiku, kdy blokuje přístup do větší paměti dle edice Windows, přestože jádra všech těchto společně vydávaných edic jsou stejná.

Průvodce studiem

Okénková metoda přístupu do paměti AWE nápadně připomíná paměť EMS používanou v 80.letech MS-DOSem. Šlo o technologii umožňující používat paměť nad 640KB v DOSu, která fungovala stejným způsobem jako AWE. Přitom kromě možnosti koupit hardwarové čipy EMS pamětí pro libovolné písíčko byla na novějších procesorech 80386 možnost použít i levnější klasickou paměť a softwarovou emulaci EMS paměti. Emulátor paměti EMS pojmenovaný EMM386 patřil svého času k nejpopulárnějšímu vybavení každého počítače a obsahuje jej i emulátor DOSu zabudovaný ve Windows NT.

11.1.3 PAE

Chce-li aplikace využívat paměť nad 4GB přes AWE, systém každopádně musí hardwarově i softwarově takovou paměť podporovat. Hardwarově podporují paměť nad 4GB procesory řady x86 počínaje modelem Pentium Pro (čili jejich poslední generace známá také jako 686 či P6), softwarově to podporují ty edice Windows, které mají podporu PAE (Physical Address Extension).

PAE zpřístupní více než 4GB fyzické paměti.

PAE je systém umožňující použití více fyzické paměti než 4GB. Procesory mají konkrétně o 4 bity širší adresovací sběrnici, takže mohou adresovat až 64GB fyzické paměti. To si vyžádalo úpravu segmentových deskriptorů a stránkovacích tabulek.

PAE je možno používat buď dohromady s AWE, nebo samostatně. Při použití samostatného PAE umí systém adresovat až 64GB paměti, přičemž každý uživatelský proces má stále k dispozici jen 4GB adresovací prostor (kde 2GB je vyhrazeno pro jádro systému). Běží-li více takových procesů, tak systém může každý z nich mapovat jinam do fyzické paměti (takže teoreticky teprve 32 procesů využije 64GB paměti). Se současným použitím PAE a AWE pak může oněch až 64GB využít každý jednotlivý proces. Hardwarový limit 64GB je dán 36bitovou adresovou sběrnicí procesorů x86 počínaje modelem Pentium Pro.

Průvodce studiem

Zatímco systém Windows NT umožňuje pomocí AWE a PAE využívat celou paměť libovolným jedním procesem, 32bitové unixové systémy nemají ve standardu žádnou obdobu

AWE. Před příchodem a rozšířením 64bitových počítačů tedy mělo Windows NT značnou výhodu u serverů používajících hodně paměti [SGG05] (např. největší světové SQL databáze běží většinou na Windows).

Microsoft však limituje celkovou paměť dle zakoupené edice Windows. 32bitové edice Windows XP a Vista limitují paměť vždy na 4GB z důvodu dosažení vyšší kompatibility se staršími ovladači zařízení. Windows 2000 podporuje dle edice 4GB až 32GB, Windows Server 2003 dle edice 4GB až 64GB (128GB na x64 procesorech) a Windows Server 2008 dle edice 4GB až 64GB. Některé speciální zlevněné edice Windows dokonce podporují výrazně méně (např. 512MB apod.). 64bitové edice mají tyto limity vyšší, například Windows XP 128GB, Vista dle verze 8GB až 128GB a serverové systémy v nejvyšších verzích až 2TB.

PAE režim je potřeba explicitně aktivovat, protože se tím změní deskriptory a stránkovací tabulky. Je to přitom změna jen pro operační systém a ovladače zařízení v režimu jádra, uživatelské procesy by neměly žádnou změnu pozorovat. Pro zajímavost dodejme, že procesory x64 navíc umějí v režimu PAE blokovat spouštění kódu v datových oblastech paměti (NX bit – no execute), takže Windows počínaje verzí XP-SP2 automaticky startuje v režimu PAE na procesorech, kde tím získá možnost vyšší ochrany paměti NX bitem. Procesory x64 v režimu PAE také umějí použít i více než 64GB RAM (podporuje-li to operační systém).

11.1.4 Paměťové stránky

Jak už bylo zmíněno v kapitole 10.3 na straně 98, v NT je paměť (každý kousek adresního prostoru) v jednom ze tří stavů:

Free (volná) je nevyužitá.

Reserved (rezervovaná) má přidělenou adresu ve virtuálním prostoru, ale neexistuje mapování na rámec.

Committed (komitovaná čili přidělená) má stránku a má přidělený rámec ve virtuálním prostoru a je připravena k použití.

Jak už jsme také zmínili, příkladem použití tohoto systému je programový zásobník. Při kompilaci programů lze v překladači nastavit u velikosti zásobníku dvě hodnoty: reserved a committed. První číslo je celková velikost zásobníku, kterou při naplnění zabere v adresním prostoru, druhé číslo je startovací velikost, kterou zásobník má při startu programu.

Existence rezervované, ale nekomitované paměti také doplňuje samotnou podstatu principu demand–paging. Máme-li rezervovanou paměť, program při startu alokuje jen pár stránek pro zásobník a o další si požádá jen v případě potřeby. Kdybychom rezervovanou paměť neměli, při startu programu se alokuje velké množství stránek pro obrovský zásobník. Tato paměť se vynuluje a dále se nejspíše nikdy nepoužije. Systém to po chvíli zjistí a odswapuje ji do sekundární paměti. Běh počítače bude tedy zpomalen o počáteční nulování velkého bloku paměti, potom bude mít menší volnou paměť a nakonec situaci vyřeší odswapováním, což opět zpomalí běh celého systému.

NT používá model pracovní množiny (working set), který jsme si představili v kapitole 10.7.1 na straně 104. NT také používá shlukování (clustering), kdy při výpadku stránky automaticky načítá několik sousedních stránek najednou, což obvykle vede ke zrychlení kvůli ušetření času při práci s diskem (jak již víme).

11.1.5 Pracovní množina rámců

Každý proces startuje s pracovní množinou minimálně 50 a maximálně 345 rámců [SoRu05]. Tyto hranice jsou však jen doporučené, při extrémním množství výpadků stránek v procesu

mu systém přidělí další rámce i nad maximum pracovní množiny a naopak pokud proces nemá žádné výpadky a jinde v systému je paměti nedostatek, systém může i snížit počet rámců pod minimální hranici. Rámce se přidělují pouze na bázi počtu výpadků, jak je zde popsáno, od verze XP. Od verze 2003 je dokonce možno i u jednotlivých procesů nastavit pevné limity pracovní množiny, systém je pak dodržuje vždy. Proces každopádně nikdy nedostane více rámců, než je globální systémové maximum. To je spočítáno při startu systému jako počet všech volných stránek – 512. Dále existují pevné horní meze přidělitelné fyzické paměti závislé na platformě: U 32bitových systémů je to přibližně 2GB, u 64bitových systémů IA-64 a x64 je to několik TB.

Při výpadku stránky proces dostane další rámec, pokud je dost volné paměti. V opačném případě systém vybere oběť a velikost pracovní množiny nemění. Oběť není odswapována ihned, místo toho je nový rámec vzat ze seznamu volných rámců a stará stránka je pouze přesunuta do seznamu stránek čekajících na zapsání na disk a vlastní zápis je proveden, jakmile má procesor čas. Je-li systém zahlcen požadavky o další rámce, pak dojde k poklesu počtu volných rámců pod minimální mez a systém hledá, kterým procesům je vhodné dále zmenšit pracovní množinu. (Pokud není systém zahlcen, postupně ukládá stránky ze seznamu použitých na disk a uvolňuje jejich rámce k dalšímu použití.) Přednostně to přitom zkouší u procesů s velkým počtem nepoužívaných rámců nebo u velkých procesů, naopak proces aplikace běžící na popředí zkouší až nakonec, atp. Najde-li systém proces, který má více rámců, než je minimální počet, tak mu odebere nějaké nepoužité. Pokud to nestačí k tomu, aby byl v systému alespoň minimální počet volných rámců, systém pokračuje v hledání a zmenšování pracovních množin u dalších procesů.

Výběr konkrétních stránek k odswapování funguje na bázi bitu accessed, čili přednostně jsou vybírány stránky, ke kterým se nepřistupovalo. Systém průběžně sleduje tento bit u každé stránky, kterou prochází, a udržuje si u ní počítadlo, jak dlouho nebyla použita. Při výběru oběti pak bere v potaz toto počítadlo. Stránkám, které mají nastaven access bit tento pouze vynuluje a nezvyšuje počítadlo.

Úpravu pracovních množin, jak byla popsána, provádí working set manager a řídí ji systémové vlákno balance set manager. To je aktivováno buď časovačem jednou za sekundu, nebo jinou systémovou komponentou, která usoudí, že je třeba změnit velikosti pracovních množin. (Při aktivaci časovačem se mimochodem stará také o boostování priorit, které jsme diskutovali v kapitole 6.2.3 na straně 53.) Vláknům, která dlouho na něco čekají (15 sekund ve Windows XP), je odswapován zásobník. Jakmile jsou odswapovány zásobníky všech vláken procesu, je odswapován celý proces. (Např. služba winlogon se používá jen pro přihlášení do systému a má běžně nulovou pracovní množinu.)

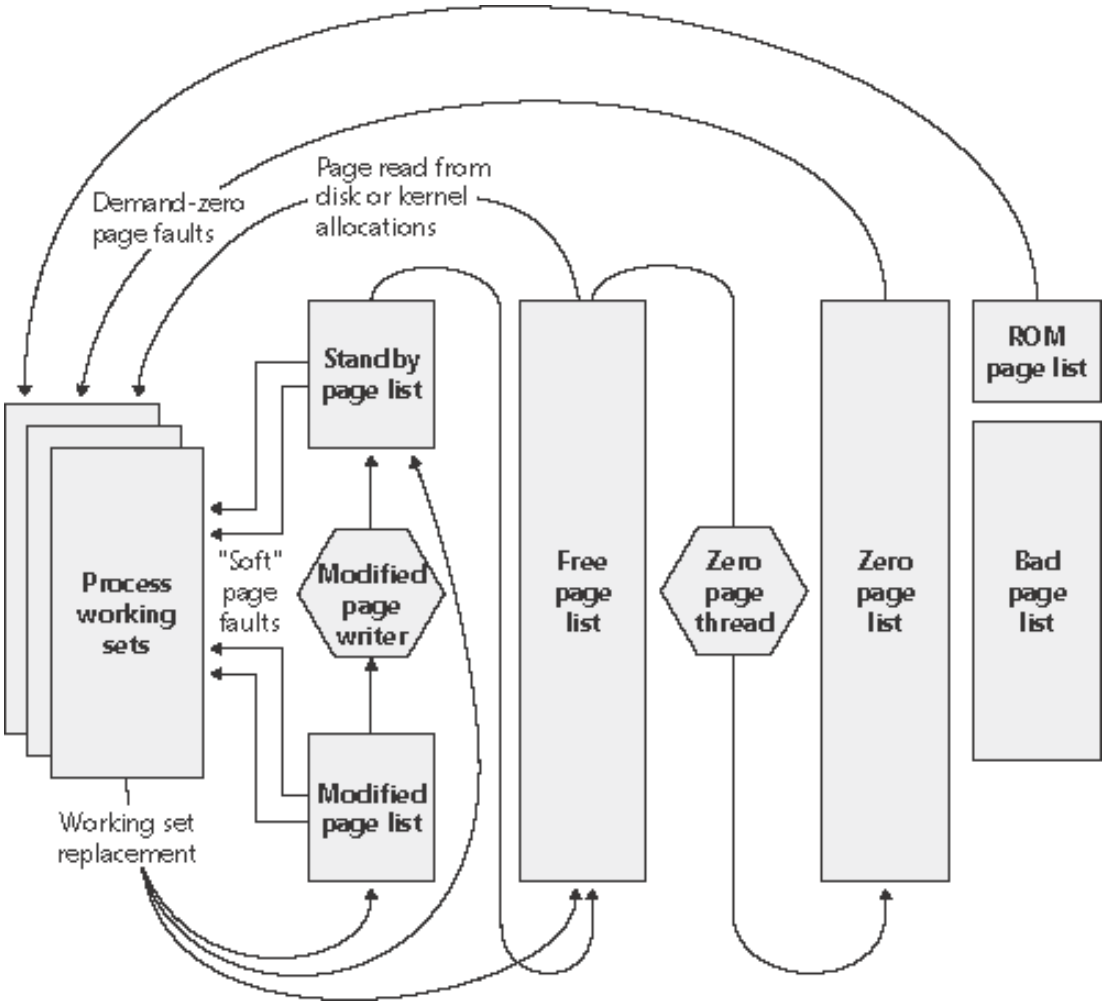
Na závěr ještě složme celou mozaiku stavů stránek a rámců: Stránka je nejprve rezervovaná (reserved). Při komitování je nejprve ve stavu zero demand – to je stav, kdy stránka už byla komitována, ale nebylo do ní nic zapsáno. Taková stránka se ještě chová jako reserved a teprve při prvním zápisu dostane rámec. Používaná stránka je committed. Na konci je stránka opět reserved a pak free.

Stavy rámců ukazuje obrázek 17. Každý rámec je nejprve volný a čeká na vynulování. Po vynulování je nulový. Po přidělení do pracovní množiny je do něj umístěna nějaká stránka. Po odstranění z pracovní množiny je rámec použitý, tehdy čeká na zápis dat na disk. Po zapsání na disk je rámec opět volný, před zapsáním ale může znovu vejít v použití, pokud je výpadek stránky, která v něm byla uložena (viz algoritmus druhé šance, kap. 10.5.4 na straně 101). Systém tedy udržuje seznamy rámců v pracovních množinách a dále seznam rámců použitých, volných a nulových. Kromě toho systém udržuje seznam stránek vadných (ten je obvykle prázdný) a stránek paměti ROM – ty samozřejmě mají zvláštní péči. Jak je také vidět na obrázku, jádro systému má povoleno alokovat paměť i bez nulování.

11.1.6 Logical prefetcher

Logical prefetcher zrychluje start systému a procesů.

Logical prefetcher je komponenta systému, zavedená od verze XP, zajišťující rychlejší start systému (boot) a procesů. Fakt, že Windows XP startuje mnohem rychleji než Windows 2000,



Obrázek 17: Stavy stránek. [SoRu05]

je možno velmi snadno pozorovat v praxi.

Prefetcher pracuje v několika fázích. Nejprve při startu systému či procesu sleduje, které části kterých datových či kódových souborů jsou při spouštění načítány. Nebýt prefetcheru, start systému a každého procesu by provázela řada téměř souvislých výpadků stránek, při kterých by se do paměti natahovaly především soubory s kódem. Největší problém při demand–paging je, že spouštěná aplikace vyžaduje své stránky obvykle v náhodném pořadí a hodně času se tedy ztratí neustálými přesuny čtecí hlavičky z místa na místo. Prefetcher je napojen na správce paměti a sleduje všechny výpadky stránek po dobu 10 sekund od startu aplikace a po dobu 30 sekund od spuštění procesu explorer (při startu systému). Prefetcher sleduje přístupy k MFT na NTFS discích (bude probíráno v kapitole 14.4 na straně 161), přístupy k souborům a přístupy k adresářům.

Po nasbírání dat je prefetcher analyzuje a zapíše log do adresáře Windows\prefetcher, kde je pro každý proces log obsahující seznam objektů, ke kterým se při startu aplikace či bootu přistupovalo. Boot systému má v adresáři prefetch vlastní soubor. Vlastní logy mají také úlohy spouštěné v rámci hostitelských procesů, jako jsou např. MMC (Microsoft Management Console) nebo svchost (service host). Seznam aplikací fungujících jako hostitelské procesy lze nastavit v systémovém registru.

Při opětovném startu systému či aplikací, které již mají záznam v adresáři prefetch, se tento načte a projde se jeho obsah. Jsou načteny všechny položky MFT a soubory kódu, které jsou v seznamu, a jsou otevřeny všechny soubory, se kterými se bude pracovat. Tím se minimalizuje počet následných výpadků stránek.

Systém ještě dále optimalizuje spouštěcí časy tím, že každých pár dní prochází údaje v prefetch adresáři a při nízkém využití procesoru spouští na pozadí defragmentaci disku s požadavkem o umístění souborů z prefetch seznamu za sebe. Při defragmentaci jsou všechny označené soubory přesunuty do jednoho místa disku za sebe do pořadí, ve kterém budou při bootu či startu aplikace načítány. To minimalizuje přesuny čtecí hlavičky.

11.1.7 Ochrana paměti

Windows NT používá několik mechanizmů ochrany paměti. Uveďme si zde alespoň stručný přehled.

1. Každé vlákno má v deskriptoru segmentu nastavenu aktuální úroveň oprávnění. Uživatelské procesy mají vždy oprávnění nejnižší (ring 3). Data jádra systému jsou chráněna tak, že jejich segmenty mají naopak nejvyšší úroveň oprávnění (ring 0). Hardware x86 zajišťuje, že vlákno běžící v ring 3 nemá přístup do segmentů ring 0, přestože jejich selektory má přístupné v GDT.

2. Každý proces má vlastní adresový prostor a vlastní sadu segmentů ve vlastní LDT. Procesy na sebe nijak nevidí, výjimkou jsou společně mapované soubory.

3. Systém využívá všechny dostupné prostředky ochrany na úrovni hardwaru (procesoru).

4. Bloky sdílené paměti mají své ACL (access control list). Systém kontroluje, že běžící proces má oprávnění k přístupu do paměti. Tím je zajištěno, že nemůže přijít libovolný „cizí“ proces a přidat se ke sdílení paměti mezi dvěma spřátelenými procesy. Pomocí ACL systém kontroluje všechny sdílené prostředky (pojmenované roury, mutexy atp.)

5. Přidělované stránky jsou vždy nulované. V nově alokované paměti nikdy nejsou zbytky dat jiného procesu. Zvláštní proces s prioritou 0 neustále prochází vyřazené stránky a nuluje je. Systém si udržuje dva seznamy volných stránek: zvlášť „čisté“ a „špinavé“. Není-li čas nulovat stránky v předstihu, každá se vynuluje nejpozději v okamžiku alokace. Výjimkou je jádro systému, kde lze používat i nenulované stránky. (Nulování stránek je jedním z požadavků americké armády na (pro ně dostatečně bezpečný) operační systém.)

11.2 Adresace na procesorech x86

Nyní se budeme věnovat tématům souvisejícím s adresací na procesorech typu x86. Toto je nezávislé na operačním systému; systém ale musí být napsán tak, aby jeho manažer paměti akceptoval vlastnosti hardwaru.

11.2.1 Typy adres

Na platformě x86 rozlišujeme tři typy adres:

Logická – tu vidí program. Má 48 bitů, z toho 16 bitů je selektor segmentu a 32 bitů je offset. Běžně se pracuje pouze s offsetem, protože segmenty jsou doplněny automaticky.

Lineární (virtuální) – ta má 32 bitů, je to již finální adresa v adresním prostoru procesu.

Fyzická – má opět 32 bitů (při PAE 36 bitů) a je to již fyzické číslo bajtu v primární paměti. Čísla mimo skutečnou paměť může operační systém využít pro značení dalších informací, včetně swapovací oblasti. Fyzický adresový prostor je společný celému systému, včetně hardwarových zařízení.

Zastavme se ještě krátce u segmentů. U adresující instrukce může být uveden segmentový prefix označující segmentový registr. Adresování se pak váže k danému segmentovému registru. Není-li segment uveden prefixem u instrukce, pak se použije výchozí segment. Každý registr použitelný pro adresaci má vlastní výchozí segmentový registr, se kterým je asociován. Většina registrů je asociována se segmentovým registrem DS, pouze ESP a EBP jsou asociovány se SS (adresují zásobník) a EIP je asociován s CS (adresuje kód). Každý segmentový registr má kromě viditelného 16bitového selektoru, kterým ukazuje do GDT nebo LDT (viz kap. 2.2.3 na straně 20) také skrytou část obsahující deskriptor, čili především adresovací bázi a limit.

11.2.2 Logické adresy

Jak už víme z kapitoly 2.2.3, x86 pracuje s pojmy segment, deskriptor a selektor. Deskriptor je 8bajtový záznam v tabulce deskriptorů. Systém má jednu globální tabulku GDT, na kterou ukazuje registr GDTR, a každý proces má svou lokální tabulku LDT, na kterou ukazuje registr LDTR. Deskriptor popisuje jeden segment paměti, udává především jeho počátek s přesností na bajt, délku (limit) s přesností na bajt, úroveň oprávnění (ring 0–3), typ (data/kód atp.), povolení čtení/zápisu/spuštění atp. Proces může používat GDT i LDT současně, slouží k tomu segmentové registry.

Každý segmentový registr má 16 bitů, do kterých vložíme selektor – je to index do tabulky GDT nebo LDT plus identifikace, kterou tabulku adresujeme, plus požadovaná úroveň zabezpečení (ring 0–3). Index do tabulky je v horních 13 bitech (každý záznam má 8 bajtů, takže je to de facto offset do tabulky), bit 2 rozlišuje GDT/LDT a bity 1–0 určují požadovanou úroveň oprávnění. (Oprávnění probereme později.) Načtením hodnoty do segmentového registru (instrukcí **mov**) se kromě uložení této 16bitové hodnoty také načte příslušný deskriptor do zbytku segmentového registru. Tato část není vidět a nelze ji nijak přečíst (jen při přepínání vláken se uloží do TSS, viz kap. 5.1.2 na straně 43).

Při nastavení hodnoty 0 odkazujeme na první záznam v GDT, který je vždy neplatný. Je to tedy jakýsi „null“ deskriptor. Tuto hodnotu lze nastavit do selektoru a procesor nevyhodí výjimku, dokud se nebudeme snažit použít tento selektor pro přístup do paměti.

Uložíme-li hodnotu segmentového registru někam do paměti, pak se uloží jen 16bitový selektor. Při načtení této hodnoty zpět do segmentového registru se znovu načte příslušný deskriptor do skryté části registru. Výhodou tohoto řešení je mj. to, že segmentové registry jsou stejně velké jako na původním procesoru Intel 8086. Programy navíc používáním 2bajtových selektorů šetří paměť, nemusejí s sebou totiž vláčet celé 8bajtové deskriptory. Při každém přístupu do paměti (čtení/zápis čehokoliv) se však používá celý deskriptor ve skryté části segmentového registru. Při přístupu do paměti nelze nepoužít selektor, protože paměť adresujeme vždy přes segmenty a ty jsou popsány v deskriptorech. Ačkoliv je s každým přístupem do paměti mnoho výpočtů, většinou je to rychlé, protože převod adres z logických na lineární a dále na fyzické zajišťuje samostatná adresovací jednotka procesoru, která běží paralelně s dalšími částmi procesoru a překládá adresy pokud možno v předstihu. V ideálním případě tedy i přes svou složitost trvá překlad adresy 0T.

Jak bylo řečeno, tabulku deskriptorů zpřístupňuje segmentový registr LDTR. Ten samozřejmě musí odkazovat do GDT (neboť LDT nemůže popisovat sama sebe). Na GDT odkazuje registr GDTR, toto však už není segmentový registr (selektor by neměl kam ukazovat), ale menší registr přímo obsahující bázi a limit v bajtech fyzické paměti. Tabulky GDT i LDT jsou velké max. 64KB, čili každá může obsahovat až 8192 deskriptorů. Segmentace paměti je přitom zcela nezávislá na stránkování, jednotlivé segmenty mohou začínat i končit na libovolném bajtu paměti, bez ohledu na velikosti stránek. V praxi se však pro jednoduchost a přehlednost preferuje systém, kdy jsou segmenty zarovnány na hranice stránek (počátkem i koncem). Každý proces přitom ve Windows používá jen několik málo svých segmentů. Velikost každého je totiž až 4GB, takže se segmenty většinou používají jen pro rozlišení dat, zásobníku a kódu (a s tím související ochranu paměti).

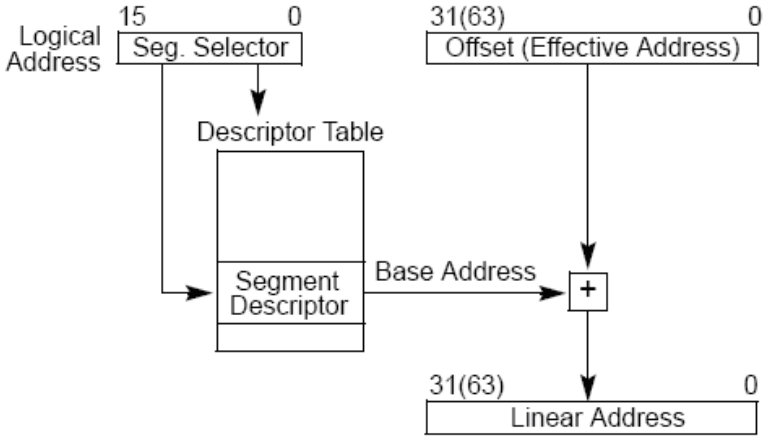
Segmentace je nezávislá na stránkování.

Pro úplnost dodejme, že v 8bajtovém (64bitovém) deskriptoru je uložena 32bitová adresa báze a pouze 20bitový limit segmentu. Zbylých 12 bitů je použito pro různé příznaky (flagy). Jedním z nich je G (granularity), který určuje, zda se 20bitová hodnota limitu bude brát jako přímá hodnota určující velikost segmentu 0–1MB, nebo jako počet stránek určující velikost segmentu až 4GB, ale ve skocích po 4KB. Další příznaky slouží např. k rozlišení 16/32/64bitového segmentu a typu segmentu. Typ segmentu je vícebitová hodnota, do které je zakódováno rozlišení kódu a dat a povolení přístupu. V kódovém segmentu je vždy povoleno vykonávání kódu a lze povolit také čtení. V datovém segmentu je vždy povoleno čtení a lze povolit také zápis. Procesor rozlišuje také dalších 6 systémových typů deskriptorů, které neukazují na segmenty. O nich se zmíníme později.

Další bit určuje, zda datový segment roste nahoru (data), nebo dolů (zásobník). U kódového segmentu tento bit značí konformitu (přizpůsobivost), která ovlivňuje oprávnění (k tomu se vrátíme později).

11.2.3 Segmentace (překlad logické adresy na lineární)

Segmentaci, čili překlad logické adresy na lineární, ukazuje obrázek 18. Báze segmentu je sečtena s offsetem a tím je získána lineární adresa. Je-li mimo limit nebo neodpovídá-li přístup oprávnění ke čtení/zápisu také uloženému v segmentovém registru, pak je vyvolána výjimka access violation (neoprávněný přístup).



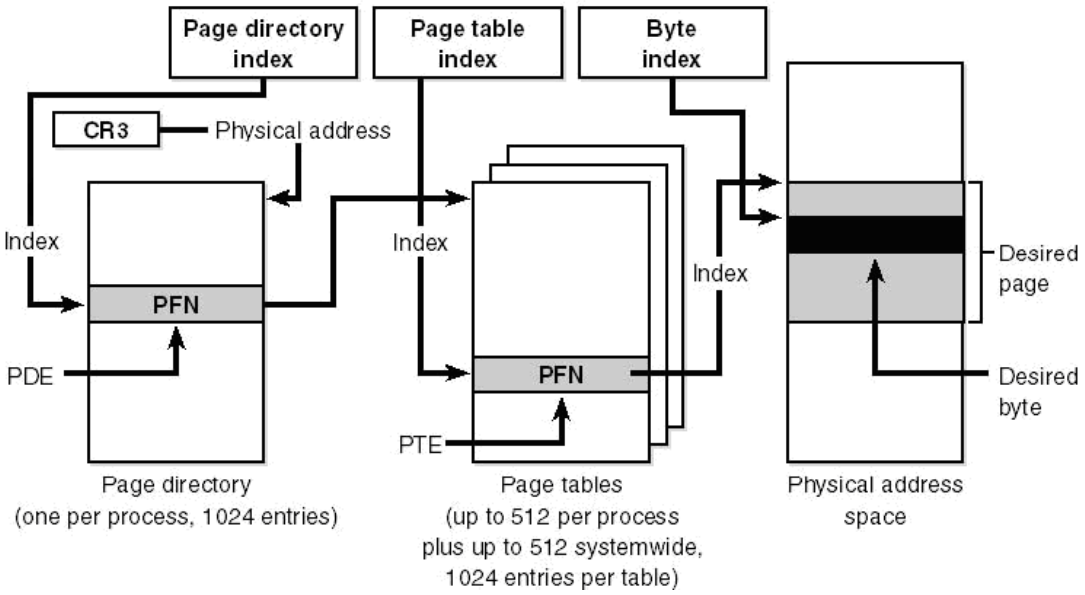
Obrázek 18: Překlad logické adresy na lineární. [IA32-3A]

11.2.4 Stránkování (překlad lineární adresy na fyzickou)

Stránkování, čili překlad lineární adresy na fyzickou, ukazuje obrázek 19. Lineární adresa se skládá z čísla stránky a 12bitového offsetu. Fyzická adresa se skládá z čísla rámce a stejného 12bitového offsetu. Překlad lineární adresy na fyzickou tedy znamená převést číslo stránky na číslo rámce. Je-li zcela vypnutá virtuální paměť (což u starších verzí Windows bylo možné udělat), pak je fyzická adresa totožná s lineární a žádný překlad se neprovádí. Podívejme se tedy na situaci, kdy virtuální paměť používáme.

Jak už víme z kapitoly 9.3 na straně 89, stránka i rámec mají na x86 standardně velikost 4KB a stránkovací tabulka obsahuje 4bajtová čísla rámců pro jednotlivé stránky. V zájmu úspory místa používají procesory x86 hierarchický systém tabulek, díky čemuž nevyužitá paměť nezabírá prostor ve stránkovací tabulce.

Stránkovací tabulka má vždy velikost jedné stránky (4KB), obsahuje tedy 1024 čtyřbajtových záznamů, které popisují 4MB paměti. Index do stránkovací tabulky má 10 bitů a tvoří prostřední část adresy (vlevo od 12bitového offsetu). Nejvyšších 10 bitů adresy tvoří index do adresáře



Obrázek 19: Překlad lineární adresy na fyzickou (PFN = číslo rámce, PDE/PTE = záznam v tabulce). [SoRu05]

tabulek, což je další tabulka o velikosti 4KB obsahující opět 1024 čtyřbajtových záznamů. Každý záznam stránkovacího adresáře udává číslo rámce obsahujícího příslušnou stránkovací tabulku. Celkově tedy adresář popisuje $1024 \times 1024 \times 4\text{KB} = 4\text{GB}$ paměti.

CR3 obsahuje fyzickou adresu adresáře stránek.

Převod adresy tedy probíhá takto: Fyzická adresa aktivního adresáře tabulek je uložena v registru CR3 (platí jen horních 20 bitů, protože začíná vždy na hranici rámce). Horních 10 bitů lineární adresy je indexem do adresáře tabulek. Hodnota na příslušném místě je číslem rámce stránkovací tabulky. Indexem do této tabulky jsou bity 12–21 lineární adresy. Číslo na příslušném místě této tabulky je pak číslo rámce, které tvoří bity 12–31 fyzické adresy. Bity 11–0 jsou stejné jako u adresy lineární.

Připomeňme, že fyzická adresa je nakonec nastavena na adresové sběrnici a na řídicí sběrnici je nastaven příslušný příkaz. Na dané adrese může být nejen paměť RAM, ale třeba paměť grafické karty nebo cokoliv jiného.

11.2.5 Velikost stránky

Standardní velikost stránky je 4KB.

Standardní velikost stránky je 4KB. Od procesoru Pentium je možno používat také 4MB stránky. Horních 10 bitů adresy je pak indexem do stránkovacího adresáře, na jehož začátek ukazuje CR3. Zbylých 22 bitů je už přímo offset (rozsah 0–4MB). NT toto používá pro jádro systému, stejně jako další moderní operační systémy. Jak už jsme zmínili v závěru kapitoly 10, použitím 4MB stránek pro statický kód jádra systému se ušetří práce adresovací jednotce procesoru a především se ušetří velmi cenný TLB. Ten se navíc dělí na dvě části, zvlášť pro 4KB a zvlášť pro 4MB stránky. Obě velikosti stránek lze libovolně kombinovat, neboť velké stránky se zapínají přímo v záznamech stránkovacího adresáře. Nastavením příslušného bitu tedy určíme, zda adresář ukazuje na stránkovací tabulku, která popisuje mapování 4MB paměti, nebo přímo na 4MB fyzické paměti. Záznamy ve stránkovacím adresáři mají 32 bitů, z toho horních 20 je číslo fyzického rámce a dolních 12 bitů jsou další flagy. K dispozici jsou také 3 bity pro potřeby operačního systému. Mezi dalšími bity najdeme i accessed a dirty, které jsou automaticky nastavovány na jedničku při přístupu (čtení/zápis) či změně (zápis) stránky. Tyto dva bity pochopitelně používá systém při správě paměti.

Existuje ještě třetí velikost: Při zapnutí PAE a vypnutí stránkovacích tabulek přepneme procesor do režimu 2MB stránek.

11.2.6 Úrovně oprávnění

x86 rozlišuje čtyři úrovně oprávnění 0–3.

Procesory x86 rozlišují čtyři úrovně oprávnění (privilege level), označované jako ring 0 až 3. Ring 0 je nejvyšší úroveň a používá ji jen jádro systému. V této úrovni lze vykonávat všechny instrukce procesoru. Ring 3 je nejnižší úroveň a běží v ní všechny uživatelské procesy. Ring 1 a 2 lze použít např. pro ovladače zařízení, ale většina operačních systémů používá jen tyto dvě úrovně, pro rozlišení běžných procesů a jádra systému.

Každý segment má nějakou úroveň oprávnění. Volání kódu v segmentu s jinou úrovní oprávnění by mělo probíhat jen přes brány. Brána je záznam v tabulce deskriptorů označený jako systémový segment. Ve skutečnosti je v tomto deskriptoru uložena adresa, kterou lze volat mezi segmenty. Brána je tedy jakýsi popisovač (deskriptor) vstupního bodu do nějaké jinak nepřístupné části kódu. Význam z hlediska zabezpečení je jasný – pomocí brány je zajištěno, že uživatelské procesy volají systémové funkce jen na správných místech. (Provedením **call** někam mimo skutečný začátek systémové funkce by bylo v některých případech možno nabourat nebo napadnout operační systém a získat kontrolu nad počítačem.) K úrovním oprávnění se ještě vrátíme v sekci 11.4 věnované ochraně paměti.

11.2.7 Stránkování v režimu PAE

V režimu PAE (physical address extension) lze používat 36bitovou adresovou sběrnici (funguje od procesoru Pentium Pro). V tomto režimu používá procesor stránky 4KB nebo 2MB a jiný formát stránkovacích tabulek. Každá tabulka má opět velikost 4KB, obsahuje ale 512 osmibajtových záznamů. Stránkování je trojúrovňové – 12 bitů pro offset, 9 bitů pro index do stránkovací tabulky, 9 bitů pro index do adresáře stránek a 2 bity pro index do tabulky ukazatelů na adresáře tabulek (page–directory–pointer table[5] (PDPT)). PDPT obsahuje jen 4 osmibajtové záznamy, celkem tedy má 32 bajtů. Registr CR3 nyní ukazuje na fyzickou adresu PDPT, přičemž 27 jeho horních bitů adresuje horních 27 bitů na 36bitové adresové sběrnici. PDPT tedy leží na 32bajtové hranici a až 128 procesů může mít PDPT ve společném rámci paměti.

2MB stránky v režimu PAE fungují analogicky 4MB stránkám v normálním režimu, tj. záznam v adresáři stránek může mít nastaven bit, který vyřadí ze stránkovacího algoritmu stránkovací tabulku a místo toho se použije 21bitový offset. (Stránky jsou tentokrát jen 2MB velké, protože index do stránkovací tabulky má jen 9 bitů.)

Způsob, jakým se nyní překládají 32bitové lineární adresy do 36bitového fyzického prostoru, je již jasný z informací v předchozích dvou odstavcích a nepotřebuje další komentář. Organizace záznamů v tabulkách také umožňuje v budoucnu rozšířit PAE z původních 36 bitů až na 64bitů adresového prostoru, což už dnes používají procesory x64, kde platí už minimálně 40 bitů.

PSE–36

Procesory Pentium 3 a novější nabízejí pod názvem PSE–36 ještě další variantu, jak zpřístupnit 64GB paměti. Podrobnosti lze najít v [IA32-3A].

11.3 Adresace na procesorech x64

Procesory x64 (známé také jako AMD64 či x86–64) jsou rozšířením procesorů typu x86. Aktuálně (v roce 2007) je již naprostá většina procesorů na trhu PC typu x64 a pouze nejlevnější modely jsou ještě typu x86 (jde o podtyp P6 čili 686). Platforma x64 je velmi chytrým rozšířením x86 a z hlediska uživatelských procesů těžko vůbec vidět nějaké rozdíly, samozřejmě kromě

[5] Tento krkolomný název je skutečně používán v originální dokumentaci [IA32-3A].

toho základního, že všechno je 64bitové. Registry v procesoru x64 jsou značeny písmenem R, čili např. RAX je 64bitové rozšíření registru EAX. Kromě toho má procesor 8 nových registrů pro běžné použití, čímž získává několikanásobně větší výpočetní sílu už jen díky svým registrům (je jich 2× více a každý je 2× větší). Procesory se navíc dnes už běžně vyrábějí i vícejádrové. Strůjcem architektury x64 je AMD, nicméně Intel dnes již vyrábí kompatibilní procesory.

Průvodce studiem

Autor této učebnice zde musí přiznat svůj prohřešek: Informace o procesorech AMD64 jsou až na malé výjimky čerpány pouze z materiálů firmy Intel. Ta totiž vyrábí skutečně kopie ze softwarového hlediska téměř k nerozeznání. Značky AMD64, x86–64, Intel 64, EM64T a IA-32e všechny označují totéž, jde jen o jakýsi „vývoj zkratek v čase“. Mimo firemní materiály výrobců je nejvíce zažité označení x64, kterého se budeme držet i my.

Podívejme se však na 64bitovou adresaci paměti. Procesory x64 mohou být provozovány buď v režimu 32bitovém, tedy s klasickým 32bitovým operačním systémem, kdy velká část čipu doslova leží ladem, nebo v režimu 64bitovém, kdy procesor má úplně jinou adresaci paměti a jinak kódovanou instrukční sadu. Jména instrukcí jsou sice stejná, ale registry jsou 64bitové a instrukce mají jiná čísla v paměti, takže automatický převod programů není možný. Je však pochopitelné, že v 64bitovém režimu lze přejít do režimu kompatibility, kde lze spouštět většinu 32bitových aplikací. Rozdíly oproti opravdovému 32bitovému režimu jsou totiž jen na úrovni jádra operačního systému – např. stránkovací tabulky jsou jiné, ale běžný 32bitový program se k takovým technickým detailům stejně nedostane, takže běží, jakoby byl na skutečném 32bitovém počítači.

Segmenty

V 64bitovém režimu se nepoužívají segmenty. Existují sice všechny segmentové registry a lze do nich načíst selektory, ale při samotné adresaci jsou ignorovány báze a limit. Platné jsou pouze ostatní položky (sloužící ke kontrole typu, oprávnění přístupu atp.). Paměť je tedy „flat“ – zcela lineární. Výjimkou jsou segmentové registry FS a GS, jejichž bázi lze při adresaci použít jako displacement adresy. (Nebudeme rozpitvávat.) Všechny ostatní segmenty, ať už do nich nastavíme cokoliv, mají při adresaci bázi nula a neomezený limit. (Segmentové registry zůstávají, protože se používají v režimu kompatibility.)

Každý deskriptor kódového segmentu má bit L určující, zda je kód 32bitový, nebo 64bitový. V systému lze 64bitové a 32bitové segmenty libovolně kombinovat, přechod mezi takovými segmenty je samozřejmě možný jen pomocí brány.

Kanonické adresy

Zajímavé je, jakým způsobem je řešeno rozšíření adresového prostoru. Současné procesory AMD používají 40bitový fyzický prostor (tj. mají 40 bitů na adresové sběrnici) a 48bitový logický prostor (tj. stránkovací tabulky operují se 48bitovými adresami). Procesory Intel byly v minulosti trochu pozadu, nicméně nyní mají stejné parametry. Systém adres používá tzv. *kanonické adresy*, díky kterým je možno velmi jednoduše v budoucnu rozšířit adresovací prostor na více bitů.

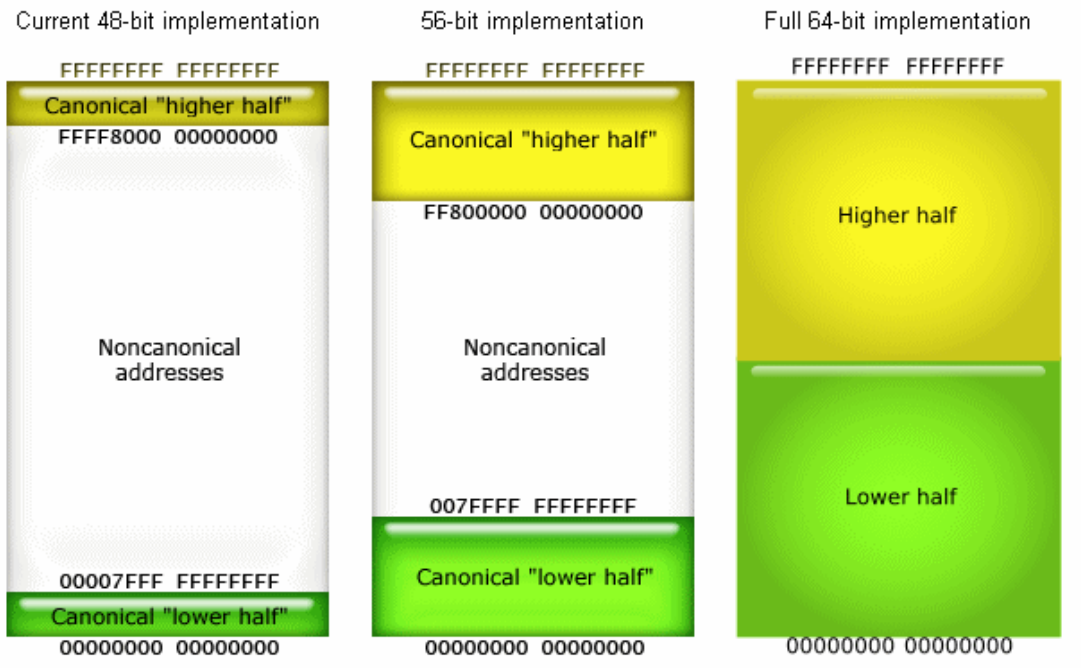
Průvodce studiem

Rozšiřitelnost adresového prostoru je velmi důležitá vlastnost, protože vzhledem k tomu,

že kompatibilita je u procesorů důležitější (či žádanější) než samotná kvalita jejich designu, lze odhadovat, že procesory x64 vydrží na trhu velmi dlouho (minimálně v nějakém režimu kompatibility dalších novějších procesorů). Současné 32bitové procesory x86 se vyrábějí od roku 1985, čili už více než 20 let, a ještě minimálně několik let budou. 64bitové x64 je nahrazují kvůli tomu, že 32bitová sběrnice už není dostačující. Přitom 64bitová sběrnice zřejmě vydrží mnohem déle než 20 let, než bude nedostačující. Ačkoliv je zřejmě předčasné dělat v tomto okamžiku nějaké velké odhady, platforma x64 může sloužit třeba i dalších 100 let.

Ještě 5 až 10 let (od nynějška, tj. roku 2007) bude zřejmě s x64 koexistovat také dožívající x86. Ačkoliv výrobci procesorů jsou již dávno na jiné metě, lze říci, že na poli softwaru je x86 dokonce stále jednoznačně dominující platformou.)

Princip kanonických adres ukazuje obrázek 20. Je odvozen od způsobu, jakým Windows (nebo jiné systémy) organizuje paměť: Dolní 2GB pro aplikace, horní 2GB pro jádro. x64 tento princip rozšiřuje na libovolně velký virtuální adresový prostor. Polovina virtuálního prostoru je tedy pro aplikace, druhá polovina pro systém. Procesor používá 64bitové registry a v budoucnu bude možná potřeba přejít z dnešních 48 bitů na více. Z hlediska registrů to nebude problém, z hlediska adresování to umožní právě kanonické adresy. Každá virtuální adresa na x64 musí být kanonická. Pro kanonické adresy platí, že neadresovatelné bity jsou kopií nejvyššího platného bitu. Tj. na současných procesorech majících 48bitové adresy je 16 nejvyšších bitů ze všech adres vždy kopií bitu 47. Výsledkem je totéž, co známe u doplňkového kódu: Horní polovina adresového prostoru se mapuje od konce, tj. jsou to jakoby záporné adresy v 64bitovém prostoru. Na obrázku 20 je zeleně označen prostor pro uživatelské procesy, žlutě prostor pro operační systém a bílá část je pro adresaci neplatná.



Obrázek 20: Kanonické adresy. [Wiki]

Podívejme se nyní, jak funguje stránkování v 64bitovém režimu. Teoreticky platforma podporuje 64bitové virtuální a 52bitové fyzické adresy. Virtuální, jak víme, jsou vázány na velikost registrů, proto 64 bitů. Fyzické jsou vázány jednak na to, kolik je místa ve stránkovací tabulce, to je oněch 52 bitů, a potom na počet bitů fyzické adresové sběrnice, to je v současnosti 40 bitů. 64bitový režim přitom lze provozovat jen v adresovacím režimu PAE. Ten totiž rozšiřuje záznamy ve stránkovacích tabulkách na 8 bajtů a zpřístupňuje tak více než 32 bitů na adresové

sběrnici.

Aresování x64 používá čtyřúrovňové tabulky.

Adresovací tabulky v 64bitovém režimu jsou čtyřúrovňové. Offset je 12bitový a každá úroveň tabulek je 9bitová, čili opět máme 512 záznamů v 4KB stránce. Celkově tedy můžeme mapovat až 48 bitů (a ne více) virtuálního prostoru. V alternativním režimu 2MB stránek se opět vynechává poslední řada tabulek, takže offset je 21bitový. Záznamy ve stránkovacích tabulkách jsou stejné jako v PAE, pouze je rozšířena adresa tak, aby bylo možno adresovat až 52 bitů fyzického prostoru. Je k tomu využito bitů, které byly dříve rezervované. (Původní stránkování umožňovalo adresovat 32 bitů. PAE rozšířilo záznamy v tabulkách ze 32 na 64 bitů, tím přibylo 32 bitů v každém záznamu. 4 bity použilo PAE pro 36bitové adresování, ostatní byly rezervované pro budoucí rozšíření. V 64bitovém režimu je adresování rozšířeno až na 52 bitů, nejvyšší bit je NX a mezilehlých 11 bitů je k dispozici operačnímu systému.) Jedinou významnou změnou je tedy zavedení čtyřúrovňové hierarchie stránkovacích tabulek.

11.4 Ochrana paměti na procesorech x86 a x64

11.4.1 Přehled

Jak už jsme mnohokrát zmínili v předchozím textu, segmentace a stránkování mohou být kombinovány s ochranou paměti. Pokud to hardware podporuje, jedná se o snadno použitelný prostředek ochrany přístupu k paměti. Konkrétně procesory x86 hardwarově podporují ochranu na několika úrovních, několikrát jsme se také o ní již zmínili v předchozím textu. Podívejme se nyní na kompletní seznam prvků ochrany paměti, které na x86 najdeme.

Kontrola přístupu do paměti je v x86 jak na úrovni segmentů, tak na úrovni stránkování a nejde vypnout. (Můžeme samozřejmě nastavit segmenty a stránky tak, aby každý proces měl přístup všude, ale kontrola stále probíhá.) Kontrola probíhá při každém přístupu do paměti, provádí ji však samostatná jednotka v procesoru, která běží paralelně, a kontrola tak nemá žádný vliv na rychlost vykonávání instrukcí. Je prováděno šest druhů kontrol:

- Kontrola limitu (velikosti) segmentu
- Kontrola typu segmentu
- Kontrola úrovně oprávnění
- Omezení adresovatelné domény
- Omezení vstupních bodů procedur
- Omezení instrukční sady

Pokud nějaká kontrola neprojde, je vyhozena výjimka vedoucí k příslušnému přerušení. Jeho obsluha je již věcí operačního systému. Některé výjimky slouží k řízení stránkování virtuální paměti, takže se nejedná o chybové stavy. (Ty ostatní pak třeba ve Windows obvykle končí nechvalně proslulou hláškou s tlačítkem „Neodesílat“.)

11.4.2 Kontrola limitů segmentu

Každý deskriptor obsahuje údaj o délce segmentu. U zásobníkových segmentů rostoucích dolů je to velké (nebo chcete-li záporné) číslo. Limity segmentů jsme již diskutovali výše.

Procesor také kontroluje limity GDT, LDT a IDT. Velikost GDT a IDT je uvedená v registrech GDTR a IDTR, LDT je chráněno jako segment (LDRT je standardní segmentový registr).

V 64bitovém režimu procesor nekontroluje ani bázové adresy, ani limity segmentů. Ve skutečnosti je přímo ignoruje – do segmentových registrů sice lze standardně načíst deskriptor, ale při adresaci se jeho bázová adresa a limit ignorují.

11.4.3 Kontrola typu deskriptoru a segmentu

Kontrolu typu deskriptoru a segmentu můžeme chápat podobně jako kontrolu typu ve vyšších programovacích jazycích. Jeden bit v deskriptoru slouží k rozlišení systémových a ostatních deskriptorů. (Systémové deskriptory neukazují na segmenty.)

Další 4 bity v deskriptoru slouží k dalšímu upřesnění typu. U systémových deskriptorů se rozlišuje celá řada bran, LDT a TSS. Celkem je tedy 16 možných typů systémových deskriptorů. Nejčastěji to je právě nějaký typ brány sloužící k volání kódu mezi segmenty. Procesor velmi omezuje volání kódu pomocí klasických instrukcí (**call**, **jmp**, **ret** apod.) do jiných kódových segmentů. Místo toho by se právě mělo používat bran. Brána je totiž zapsaná v tabulce deskriptorů, takže sám proces ji nemůže vytvořit, ani změnit. Pomocí brány je specifikováno konkrétní místo kódu, které lze zavolat. Procesor tak nemůže volat libovolný kód, který si zamane, má možnost volat pouze správné vstupní body procedur pomocí bran v tabulce deskriptorů. Je pochopitelné, že tento způsob ochrany volání se používá především u jádra systému, kde by volání na náhodné adresy mohlo významně uškodit celému systému.

U segmentových deskriptorů slouží ony 4 bity k rozlišení jejich dalších vlastností. Jeden bit rozlišuje datové a kódové segmenty. Datové segmenty mají vždy povoleno čtení, další dva bity slouží k blokování zápisu a rozlišení zásobníku rostoucího dolů od ostatních datových segmentů. Kódové segmenty mají vždy povoleno spouštění kódu, další dva bity slouží k povolení čtení a nastavení konformity. Konformní (přizpůsobivý) segment přijme při volání od volajícího nižší úroveň oprávnění. Nekonformní (či nonkonformní) segment se nepřizpůsobuje volajícímu a vykonává kód ve své úrovni oprávnění. (Budeme ještě diskutovat později.) Poslední bit u všech slouží jako příznak přístupu používaný algoritmem výběru oběti při správě paměti.

Těchto pět bitů se pak používá při běhu v různých okamžicích. Ihned při načtení selektoru se kontroluje: CS může ukazovat jen na kódový segment, SS jen na segment s povoleným zápisem, ostatní segmentové registry jen na segment s povoleným čtením. LDTR může ukazovat jen na segment LDT. TR může ukazovat jen na segment TSS.

Při přístupu do segmentu se kontroluje: Číst lze jen ze segmentu s povoleným čtením. Zapisovat lze jen do datových segmentů s povoleným zápisem. Instrukce jako call a jmp jsou kontrolovány na správný typ segmentu dříve, než jsou provedeny. Procesor také dokáže díky deskriptoru poznat volání mezi procesy a namísto obyčejného volání provede přepnutí kontextu procesu (ve Windows vlákna).

Null segment (první položka GDT) nelze načíst do CS nebo SS. Do ostatních registrů ano, ale nelze je pak použít k přístupu do paměti. V 64bitovém režimu procesor null segment nepodporuje, tj. první položka GDT obsahuje standardní deskriptor.

11.4.4 Úrovně oprávnění

Jak už víme, jsou 4 úrovně oprávnění (tzv. ring). Nejvyšší je ring 0 a používá ji jádro systému. Nejnižší je ring 3 a používají ji uživatelské procesy. Úrovně 1 a 2 mohou být využity např. pro ovladače zařízení či jiné součásti systému, většina současných operačních systémů je však nevyužívá (tj. rozlišují jen jádro a zbytek). Ve stručnosti lze říci, že systém úrovní slouží k tomu, aby omezil přístup běžných procesů do jádra systému. Úroveň oprávnění je tedy 2bitová hodnota, kterou najdeme na různých místech.

CPL (current privilege level) je aktuální úroveň běžícího procesu. Je uložena v CS a SS a obvykle je rovna úrovni oprávnění aktuálního kódového segmentu. Výjimkou je volání konformního segmentu vyššího oprávnění, kdy CPL zůstává na původní hodnotě (kódový segment tedy pak má vyšší oprávnění, než je CPL v CS).

DPL (descriptor privilege level) je úroveň oprávnění uložená v deskriptoru brány či segmentu. DPL se používá při přístupu k bráně či segmentu.

Datový segment, TSS a volací brána (call gate): DPL je nejnižší úroveň, pro kterou je povolen přístup, tj. vyžaduje se CPL≤DPL.

Nekonformní kódový segment při přímém volání: DPL je požadovaná úroveň oprávnění, tj. vyžaduje se CPL=DPL.

Nekonformní kódový segment při volání přes bránu a konformní kódový segment: DPL je nejvyšší úroveň oprávnění, pro kterou je povolen přístup, tj. vyžaduje se CPL≥DPL.

RPL (requested privilege level) je požadovaná úroveň oprávnění, kterou můžeme libovolně nastavit do spodních dvou bitů segmentového selektoru. RPL se používá k omezení oprávnění k segmentu, aniž bychom měnili tabulku deskriptorů. Podrobněji v sekci 11.4.5.

Do datových segmentů lze tedy přistupovat, pokud CPL≤DPL a zároveň RPL≤DPL. Jak jsme již zmínili výše, do datových segmentů lze také načíst kódový segment, pokud má povoleno čtení. Podobně lze použít segmentový prefix SEGCS ke čtení dat přímo přes CS, opět ale pouze pokud je tam povoleno čtení. Dodejme, že kromě nastavení hodnoty do segmentových registrů je také možno používat vzdálené pointery, kdy se selektor nenačítá do registru, ale je uveden přímo u jedné instrukce.

SS lze nastavit jen na segment, kdy CPL=RLP=DPL.

11.4.5 Privilegia u kódových segmentů

Jak už bylo vidět výše, u přechodu mezi kódovými segmenty je situace poněkud složitější než při práci s daty.

Přechod na jiný kódový segment je možný přímým voláním, přes volací bránu, přes segment TSS nebo přes bránu úlohy (task gate). Druhé dva případy vedou k přepnutí kontextu procesu, což je specifický případ. První dva případy použijeme k běžnému volání, jak jsme již zmínili výše.

Procesor rozlišuje čtyři druhy bran: call, trap, interrupt, task. Kromě prvního případu z nich jde o specifické brány pro obsluhy výjimek, přerušení a přepínání kontextů. Popišme si tedy pouze fungování běžné volací brány (call gate).

Deskriptor volací brány obsahuje selektor a offset kódu, který má být zavolán. Procesor zásadně nesdílí zásobník mezi různými úrovněmi privilegií, tj. každé vlákno má čtyři zásobníky, pro každou úroveň jeden. Pokud zavoláním přes volací bránu dojde ke změně CPL, dojde také ke změně zásobníku (z TSS se načte jiná hodnota do SS). Deskriptor volací brány proto také udává, kolik dat ze zásobníku volajícího segmentu se má překopírovat do zásobníku volaného segmentu (max. 32 buněk zásobníku). Volací bránu lze také použít pro přechod mezi 16bitovým a 32bitovým segmentem. (Procesor může sdílet 16bitový a 32bitový kód v jednom systému. Je ostatně známo, že Windows umí spouštět programy pro MS-DOS – ty jsou právě načítány do 16bitových segmentů.)

V 64bitovém režimu se používají 16bajtové deskriptory, ve kterých je místo pro celý 64bitový offset. Předávání parametrů přes zásobník zde však podporováno není (procesor však má dvojnásobný počet obecných registrů, ty lze využít). 16bajtové deskriptory 64bitových bran mohou být míchány v jedné tabulce s 8bajtovými deskriptory 32bitových bran, systém podle typových bitů dokáže chybný přístup k horní polovině velkého deskriptoru poznat. Míchání 32bitových a 64bitových bran v jedné tabulce je nutné, protože to je jediný způsob, jak přecházet mezi 32bitovým a 64bitovým kódem – ten může koexistovat v jednom systému a právě jen pomocí bran lze přecházet např. mezi 32bitovým kódem staršího programu a 64bitovým jádrem systému. Připomeňme také, že 64bitové adresy musejí být v kanonickém tvaru a platí to i pro offset ve volací bráně. 64bitová volací brána může směřovat jen do 64bitových kódových segmentů.

Volání brány probíhá pomocí libovolné standardní volací instrukce. Směřuje-li selektor na bránu, pak je offset ignorován a celé volání proběhne na bázi selektoru brány. Procesor přečte druhý selektor z brány, čímž získá skutečný cíl volání. Pak kontroluje úrovně privilegií. Při volání přes bránu vystupují dva DPL – DPL brány a DPL kódového segmentu, obojí se kontroluje. DPL brány určuje požadované minimální oprávněni pro přístup k bráně. DPL segmentu se kontroluje standardním způsobem vůči RPL a CPL (záleží také na konformitě segmentu, viz výše). Výjimkou je volání skokem (**jmp**), které nesmí jít do segmentu vyšší úrovně privilegií nekonformního segmentu (u konformního ano).

Směřuje-li volání úspěšně do nekonformního segmentu na jiné úrovni oprávnění, dojde k přepnutí zásobníku. Směřuje-li volání úspěšně do segmentu konformního, tak nedojde ke změně CPL a přepnutí zásobníku. TSS obsahuje SS:ESP zásobníku pro ring 0–2. Ring 3 nelze volat přes bránu, takže nepotřebuje zásobník v TSS. Původní hodnoty SS:ESP jsou při volání uloženy na nový zásobník a po skončení volání opět obnoveny. Hodnoty SS:ESP pro ring 0–2 v TSS však nejsou nikdy měněny! Při každém použití volací brány jsou hodnoty z TSS použity jako výchozí. Je věcí operačního systému, aby do TSS připravil dostatečně velké zásobníky pro každý proces.

V 64bitovém režimu se nepoužívají segmenty, proto TSS neobsahuje selektory pro zásobníky v ring 0–2, ale pouze jejich počáteční offsety. Jinak je fungování obdobné, pouze vše probíhá 64bitově. Jak bylo uvedeno výše, data na původním zásobníku nejsou překopírována. Potřebujeme-li předávat data zásobníkem, pak lze jednoduše přečíst adresu původního RSP z nového zásobníku a přečíst si data ručně. (Znovu: 64bitový režim nepoužívá segmenty, takže původní a nový zásobník jsou přístupny současně v paměti, každý má jen vrchol na jiném místě. Proto by bylo zbytečné a neefektivní kopírovat data mezi zásobníky.)

Návrat z volání provedeme instrukcí **ret**. Blízký návrat pouze přečte offset ze zásobníku a nastaví jej do EIP. Kontroluje se limit aktuálního kódového segmentu. Vzdálený návrat mezi segmenty stejné úrovně oprávnění proběhne vytáhnutím dvou hodnot ze zásobníku a nastavením CS:EIP. Načítá se nový CS, takže jsou provedeny všechny příslušné kontroly. Změna úrovně oprávnění je možná jen směrem dolů, čili návrat je možný jen do stejné nebo nižší úrovně. (Pochopitelně: Platí opak oproti volání.) Při návratu do jiné úrovně oprávnění se vrací CS:EIP i původní zásobník (bere se ze zásobníku, tj. musí být uložen hned za návratovou adresou), jsou provedeny všechny příslušné kontroly. Hodnoty CS:EIP před instrukcí návratu jsou ztraceny. Nakonec je provedena kontrola všech ostatních segmentových registrů a ty, které ukazují do segmentů vyšší úrovně oprávnění, jsou nulovány.

Od procesoru Pentium 2 jsou k dispozici také instrukce **sysenter** a **sysexit**, které umožňují volání systémových služeb rychlejším způsobem než přes volací bránu. Rychlejší je toto volání proto, že používá dodatečné registry v procesoru, namísto čtení hodnot z volacích bran. Díky tomu je méně přístupů do paměti a je třeba provádět méně kontrol.

11.4.6 Kontrola na úrovni stránek

Kontrola na úrovni stránek funguje na bázi stránkovacích tabulek. Probíhá při každém přístupu do paměti a zajišťuje ji samostatná jednotka v procesoru pracující paralelně, takže ani tato kontrola nijak nezpomaluje vykonávání instrukcí. Pokud dojde k porušení ochrany, je vyhozena výjimka page fault (výpadek stránky). Operační systém pak musí zkontrolovat, zda je výjimka způsobena chybějící stránkou virtuální paměti, nebo neoprávněným přístupem.

Každý záznam ve stránkovací tabulce, včetně adresářů, má dva ochranné bity. První z nich označuje systémové stránky – lze jej nastavit pro zákaz přístupu z ring 3. Stránky takto nastavené jsou v adresovém prostoru procesu, ale nejsou přístupné z kódových segmentů v ring 3. Voláním funkce operačního systému přes bránu dojde ke změně kontextu procesu a změně privilegií. Tím proces získá bez práce přístup do takto chráněné paměti. Přitom různá vlákna téhož procesu mohou paralelně fungovat v rámci jednoho adresového prostoru a pouze ta z nich, která právě vykonávají systémové funkce, mají přístup do systémových paměťových stránek.

Druhý ochranný bit zakazuje zápis do stránky. Tuto ochranu jsme již zmiňovali např. v souvislosti s copy–on–write. V ring 0 lze vypnout tuto ochranu, takže procesor může zapisovat i do stránek určených jen pro čtení. To je také velmi výhodné, proces díky tomu může sám provádět různé systémové operace nad pamětí. (Uvědomme si, že proces má několik vláken a nešlo-li by vypnout ochranu zápisu, tak by v případě potřeby něco zapsat do takové paměti systém musel přenést řízení do jiného procesu s jinou virtuální pamětí. Systém by se tak velmi zpomalil a zároveň zkomplikoval.)

Jelikož ochrana je vedena jak na stránkách, tak na adresářích stránek, tyto mohou mít různá nastavení. To může pomoci především úsporou času – namísto zdlouhavého nastavování každé stránkovací tabulky stačí nastavit příslušný stránkovací adresář. Je to taky jeden z důvodů, proč operační systémy na x86 používají nelineární organizaci stránek (systém má různé části „rozházené“ po virtuálním prostoru).

Přístup do GDT, LDT, IDT a pomocných zásobníků ring 0–2 při volání přes bránu je vždy kontrolován jakoby CPL=0. Tím je zajištěno, že systém neskončí totálním kolapsem např. při zákazu přístupu do stránek GDT.

11.4.7 NX bit

V režimu PAE je ve stránkovacích tabulkách ještě NX bit (no execute), který zakazuje spouštění kódu v dané stránce. Jak už bylo zmíněno v sekci 11.1.3 na straně 111, je to velmi silný prostředek ochrany před malwarem (viry atp.), proto operační systémy podporující NX bit přímo startují v režimu PAE, i když má počítač méně než 4GB paměti.

Upozorněme, že NX bit není podporován na všech procesorech, které podporují PAE. Dříve to byla výsada dražších modelů, nejnovější procesory již NX podporují všechny. Spolu s no–execute na bázi NX bitu provádí systém mající tuto schopnost také kontrolu „reserved“ bitů. Reserved je rezervovaný bit a měl by být stále nastaven na výchozí hodnotu (obvykle nulu). Reserved bity se vyskytují ve všech systémových strukturách, které mají nějakou nevyužitou část – právě to, co je nevyužité, je označeno reserved. Reserved bity jsou určeny pro další možné rozšíření procesorů dalšími funkcemi. Vyžadováním a kontrolou správných hodnot (tj. obvykle nul) v reserved bitech systém zajišťuje, že současné programy nezpůsobí nějakou nechtěnou nebo i plánovanou nehodu na budoucích procesorech. Schopnost kontrolovat reserved bity je navázána na podporu NX bitu.

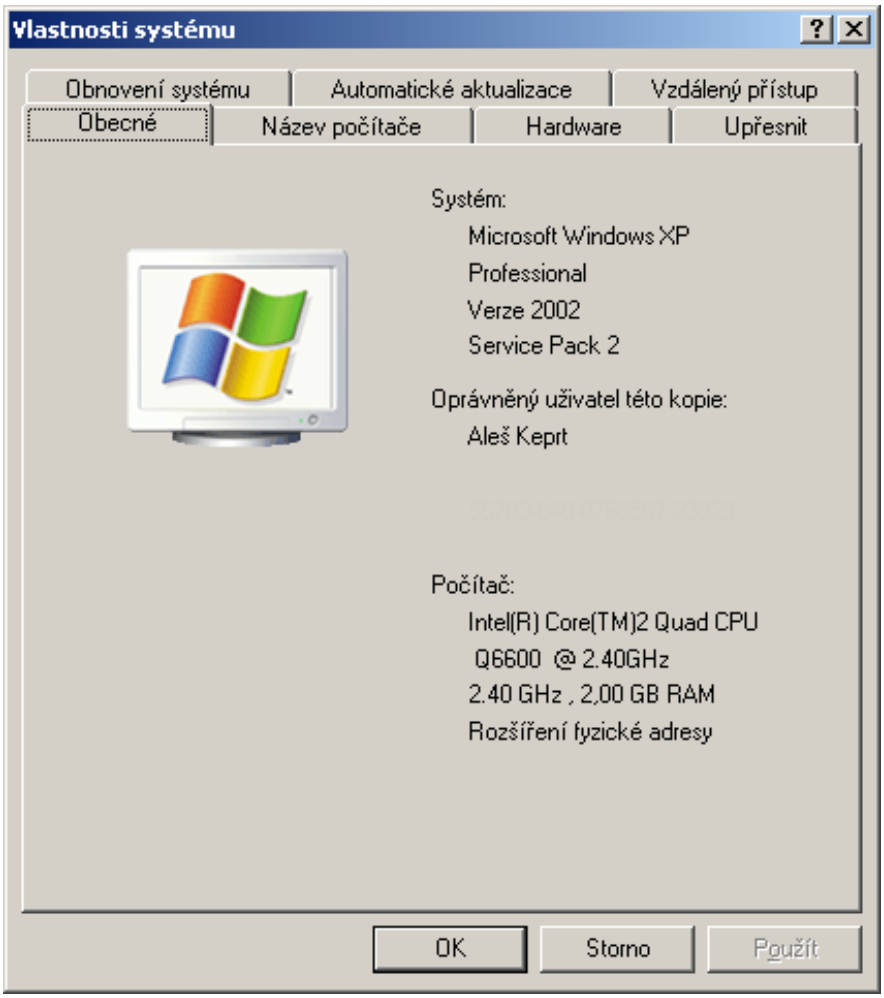
Průvodce studiem

Jelikož NX bit je v nejvyšším bitu 64bitového záznamu stránkovací tabulky, není ve 32bitovém režimu k dispozici. Windows XP SP2 a novější proto u procesorů typu x64 i ve 32bitové edici ve skutečnosti pracuje v 64bitovém režimu, aby NX bit mohl být využit k důležité ochraně paměti před přetečením bufferu.

Windows automaticky detekuje podporu NX bitu na procesoru a přechází do režimu 64bitového adresování (PAE) bez nějaké předchozí domluvy s uživatelem. To, že 32bitová edice Windows jede v PAE, se lze dočíst v ovládacím panelu Systém, viz obr.21.

Shrnutí

V této kapitole jsme se věnovali hodně technickým a praktickým věcem. V první části jsme rozebrali stránkování ve Windows NT na platformě x86. Zmínili jsme AWE, PAE, systém typů stránek a ochranu paměti v NT. Ve druhé části kapitoly jsme probrali adresaci na procesorech x86 a x64. Jsou to dva velmi podobné typy procesorů a řadu vlastností mají společných, proto byly popsány společně. Probrali jsme typy adres, překlady mezi nimi, segmentaci, stránkování, úrovně oprávnění, režim PAE, rozdíly v 64bitovém režimu x64.



Obrázek 21: Titulek „Rozšíření fyzické adresy“ ukazuje, že systém běží v režimu PAE.

Zvláštní pozornost jsme věnovali také ochraně paměti navázané na segmentaci a stránkování. Téma již hodně zmiňované v předchozích dvou kapitolách jsme nyní rozebrali z hlediska procesoru x86 a x64, které provádějí kontrolu na úrovni segmentů i na úrovni stránek.

Pojmy k zapamatování

- AWE (address windowing extension)
- PAE (physical address extension)
- ACL (access control list)
- logical prefetcher
- deskriptor a selektor
- tabulky deskriptorů GDT a LDT
- registry GDTR a LDTR
- registr CR3
- úroveň oprávnění (privilege level) / ring
- CPL, DPL, RPL
- x64
- limit segmentu
- NX (no execute) bit

Kontrolní otázky

1. *Popište, jak systém Windows NT používá segmentaci paměti.*
2. *Kde a jak funguje AWE?*
3. *Popište, jak funguje PAE. V čem se liší implementace na procesorech x86 a x64?*
4. *Je stránkovací režim PAE vlastností Windows, nebo mikroprocesoru?*
5. *Vysvětlete výhodu rozlišování rezervované a komitované paměti.*
6. *Co je hlavním důvodem toho, že Windows přiděluje paměť až po vynulování?*
7. *Popište, jak Windows NT implementuje systém pracovních množin rámců.*
8. *Co je smyslem logical prefetcheru? Popište jeho funkci.*
9. *Které mechanizmy ochrany paměti používá Windows NT?*
10. *Vysvětlete pojmy logická, lineární a fyzická adresa.*
11. *Vysvětlete, k čemu slouží GDT, GDTR, LDT a LDTR. Dále přesně popište, kterou část práce má na starosti operační systém a kterou hardware.*
12. *Vysvětlete algoritmus segmentace a smysl deskriptorů a selektorů na procesorech x86.*
13. *Procesory x86 a x64 používají jedno až čtyřúrovňovou hierarchii stránkovacích tabulek. Vysvětlete, které všechny možné režimy se dají použít, kdy a jak. Uveďte také, jaké jsou adresovací možnosti či omezení v jednotlivých režimech, jak se tyto různé režimy dají v jednom systému kombinovat a jaký má takový kombinovaný způsob stránkování smysl.*
14. *Jaké velikosti stránek nebo obecně řečeno bloků paměti používaných při stránkování podporují procesory x86 a x64? Uveďte všechny možnosti a pomocí souvislosti s formátem stránkovacích tabulek vysvětlete, proč to jsou právě tyto velikosti.*
15. *Vysvětlete, jak přesně se používají hodnoty úrovní oprávnění CPL, DPL a RPL. Uveďte také odkud se ty hodnoty při vykonávání kódu berou, kde jsou uloženy a kdy vlastně se provádějí kontroly na bázi těchto hodnot.*
16. *Vysvětlete, jak lze na procesorech x86 volat kód mezi různými kódovými segmenty a různými úrovněmi oprávnění. Jak je to při takových voláních s programovým zásobníkem?*
17. *Vysvětlete pojem kanonické adresy. Na jakých procesorech se používají a proč? Jak kanonický systém adres souvisí se strukturou a organizací stránkovacích struktur? (Nápověda: Stránkování musí být funkční a se stejným algoritmem bez ohledu na to, kolikabitové jsou ve skutečnosti adresy na konkrétním modelu procesoru.)*
18. *Vysvětlete, proč přes volací bránu nelze volat kód v ring 3.*
19. *V operačních systémech, ve kterých si jednotlivá vlákna sama vykonávají kód systémových služeb, je potřeba, aby kód i data jádra systému byly mapovány do adresového prostoru procesu, ale zároveň tato paměť musí být chráněná, protože proces z bezpečnostních důvodů nesmí mít přístup k paměti jádra. Jak je toto řešeno na platformě x86?*
20. *Proč NX bit pomáhá chránit počítač proti útoku typu přetečení bufferu (buffer overflow)?*

Cvičení

1. Vyzkoušejte si spustit Windows v režimu /3GB.

12 Souborový systém

Studijní cíle: Tato kapitola zahajuje další celek, tentokrát věnovaný správě diskového prostoru. Cílem je seznámit se se základními pojmy, jako je soubor a proud, které používáme na vyšší úrovni abstrakce při práci s diskem či přesněji vnější (sekundární) pamětí obecně. V následujících kapitolách se pak podíváme na nižší úrovně abstrakce.

Klíčová slova: soubor, adresář, proud, sdílení souborů, ochrana souborů, ACL, DACL

Potřebný čas: 140 minut.

Úvod

V předchozích kapitolách jsme probrali správu procesoru a paměti, nyní se dostáváme ke třetí a poslední části počítače dle von Neumannova modelu – externím zařízením. Namísto obecné diskuzi o zařízeních se však budeme nejprve věnovat jen tématu volně ale plynule navazujícímu na správu operační paměti, kterým je správa diskového prostoru.

Vnitřní paměť počítače není nikdy dost velká, aby v ní byly uloženy všechny informace dohromady a neustále, proto počítače používají také tzv. *vnější paměť*. Tato oblast prošla během vývoje počítačů také svým dramatickým vývojem a dnes jsme v situaci, kdy pro běžnou práci používáme v převážné míře disková zařízení. Studium externí paměti tedy můžeme omezit na disková zařízení a stále pokryjeme téměř 100% případů.

Další omezení, které si můžeme směle dovolit, je předpokládat, že data na disku jsou organizována souborovým systémem. Těžko si dnes někdo dokáže představit, že by svá data na disku měl uložena jinak než v souborech a adresářích. Z hlediska studia operačních systémů je souborový systém de facto představitelem disku jako takového. Je to totiž softwarová podoba tohoto hardwarového zařízení, s diskem pracujeme pouze prostřednictvím souborového systému a všechno, co je za tím, už se týká spíše hardwaru a do studia operačních systémů patří jen okrajově.

12.1 Soubor

Průvodce studiem

Pojem *soubor* má jednu nedobrou vlastnost – máte-li vysvětlit člověku, který nikdy neviděl počítač, že tam bude muset pracovat se soubory, zjistíte, že některé věci prostě vysvětlit nejdou. Na otázku: ”Co je to soubor?” pro některé lidi neznalé počítačů prostě odpověď neexistuje.

Počítače obvykle používají k ukládání dat disková či pásková zařízení, nejčastěji s magnetickým nebo optickým médiem. Operační systém definuje abstrakci nad těmito zařízeními, čili jakýsi společný hardwarově nezávislý pohled, kde logickou jednotkou ukládání dat je *soubor*. Procesy pak používají soubory k trvalému či dočasnému ukládání a načítání dat, přičemž úkolem operačního systému je zajistit mapování souboru z podoby, v jaké ho vidí procesy, do fyzické podoby na externím zařízení.

Soubor můžeme definovat jako pojmenovanou kolekci souvisejících informací uloženou ve vnější paměti. Z hlediska procesu je soubor nejmenší jednotka vnější paměti, kterou lze alokovat. (Čili data nelze do vnější paměti dostat jinak, než tak aby byla v souboru). Soubory mohou obsahovat programy i data, mohou to být jen posloupnosti bitů či bajtů, nebo nějakých složitěji strukturovaných záznamů. Říkáme, že soubor má nějakou strukturu, té přitom musí rozumět samotný proces, který soubor používá. Z hlediska operačního systému je soubor pouze

pojmenovaná posloupnost bajtů (či jiných jednotek pevné délky). Říkáme tomu také data či tělo souboru.

Každý soubor má kromě dat také nějaké *atributy*. Je to především jméno, dále pak typ, velikost, nastavení ochrany/oprávnění přístupu, datum a čas, majitel atp. Důležitými atributy jsou také identifikátor souboru a jeho pozice/umístění na zařízení (tyto dva atributy uživatel souboru přímo nevidí).

Atributy o souborech jsou sdruženy v *adresářích*, které jsou také uloženy ve stejné vnější paměti. V adresáři bývá buď celá sada atributů, nebo jen jméno a identifikátor souboru, podle kterého lze najít ostatní atributy.

12.2 Souborové operace

Operační systémy, hlavně ty větší, obvykle nabízejí velké množství souborových operací. Těmi základními jsou:

Create (vytvoření souboru) proběhne tak, že v adresáři je vytvořen záznam. Na disku pochopitelně musí pro soubor být místo.

Write (zapisování souboru) může probíhat buď přidáváním dat na konec souboru, nebo přepisováním existující části souboru. Proces zapisuje data tak, že připraví v paměti větší souvislý blok dat k zapsání, nebo data zapisuje postupně po bajtech či jiných velmi malých částech.

Read (čtení souboru) probíhá tak, že proces připraví blok prázdné paměti a nechá do něj načíst část souboru.

Seek (změna pozice čtení/zápisu) na některých starších či speciálních zařízeních ani nemusí být podporována.

Erase (smazání souboru) proběhne tak, že v adresáři je vyhledán záznam souboru dle jména a je zrušen. Spolu s tím je uvolněno veškeré místo, které soubor na disku zabíral.

Truncate (zkrácení souboru) proběhne nastavením nové velikosti v záznamu adresáře a uvolněním té části disku, která již nebude použitá. (Prodloužit soubor lze jednoduše připsáním dat na jeho konec. Pokud by soubor nešel jednoduše zkrátit, museli bychom jej zrušit úplně a zapsat novou verzi znovu.)

Pomocí těchto šesti operací lze provádět většinu běžných činností se soubory. Princip souboru přitom používá většina operačních systémů nejen pro disky, ale i pro většinu ostatních zařízení používaných ke vstupu či výstupu dat. Zatímco u myši to může působit komicky, u jiných zařízení lze vidět mnoho vlastností velmi podobných souborům na disku, a proto je tento jednotný přístup typu „Všechno je soubor.“ příjemný a dobře se s ním pracuje.

Pro příklad: Klávesnice funguje jako soubor, ze kterého lze jen číst. Nelze provádět write, seek, erase, ani truncate. Při zavolání funkce read může být proces blokován až do doby, než je k dispozici nějaká klávesa. Z minulosti přitom speciálně u klávesnice přetrval zvyk, že data z klávesnice jdou do procesu po celých řádcích, tj. po stisku klávesy Enter se v souboru klávesnice „objeví“ znaky celého řádku najednou. Potom je proces opět blokován až do zadání dalšího řádku. Toto chování má historické důvody a obvykle jej lze i vypnout.

Druhý příklad: Tiskárna funguje jako soubor, do kterého lze jen zapisovat. Nelze provádět read, seek, erase či truncate. Tiskárna tímto způsobem funguje bez ohledu na to, zda je textová (tiskne přímo text), nebo grafická (tiskne jednotlivé malé tečky – doty).

12.3 Otevřené soubory a handly

Aby systém nemusel při každé operaci hledat v adresáři soubor dle jména, používá se operace **open** – *otevření souboru*. Otevření je operace, při které systém najde v adresáři soubor dle jména, načte jeho atributy a začne si udržovat v paměti informace o tomto souboru. Systém vrátí procesu tzv. *handl* (neboli rukojeť, z anglického handle). Při další práci pak proces identifikuje soubor handlem, nikoli jménem, čímž se práce s diskem velmi urychlí. Po skončení práce proces soubor zavře (*close*).

Systém otvírání souborů a handlů se komplikuje tím, že v systému je obvykle souběh více procesů. Ty mohou pracovat se stejným diskem a dokonce i se stejným souborem. Systém si proto vede globální seznam otevřených souborů a pak ještě lokální seznam otevřených souborů pro každý proces zvlášť, kde má např. aktuální pozice čtení či zápisu a samozřejmě odkaz na soubor do globálního seznamu. V globálním seznamu je u každého souboru mj. počet otevření procesy. Soubor zůstává fyzicky otevřen, dokud s ním pracuje alespoň jeden proces. Jakmile počítadlo otevření klesne na nulu, je soubor fyzicky uzavřen.

12.4 Zamykání souborů (lock)

Moderní operační systémy obvykle umějí i zamykat otevřené soubory. Smyslem zamykání je, že proces pracující se souborem si zamknutím zajistí, že jiný proces nebude k souboru přistupovat. Zamykání má význam ve dvou případech:

- Proces bude soubor měnit a nechce, aby jiné procesy četly obsah souboru během zápisu, kdy jsou tam částečně stará a částečně nová data, ale až po sestavení celé nové verze.
- Proces bude soubor číst a nechce, aby mu jej během toho jiný proces začal měnit a způsobil tak přečtení nekonzistentního (částečně starého a částečně nového) obsahu.

Soubor lze obvykle zamknout buď jen pro čtení, nebo pro zápis. Zamknutí pro čtení umožňuje nerušené čtení, ale neblokuje jiné procesy, které chtějí také jen číst. Zamknutí pro zápis umožňuje nerušeně číst i zapisovat a jiné procesy nemají k souboru přístup. V některých systémech se zamykání takto nerozlišuje, de facto tedy existuje jen zamknutí pro zápis. Další skupina operačních systémů neumožňuje současný přístup více procesů k jednomu souboru, tam je tedy de facto zámek pro zápis automaticky aktivován při každém otevření souboru. (Takto se chovají především systémy starší a/nebo jednoúlohové.) Moderní systémy naopak umožňují zamykat soubory i po částech a umožňují tak např. současný chráněný zápis dvou procesů, kdy každý si pro sebe zamkne „svou část“ souboru.

Průvodce studiem

O zamykání byla řeč již v kapitole 7 o synchronizaci. Samostatné typy zámků pro čtení a zápis jsou vlastně zobecněním principu, který známe jako mutex, a toto se může v praxi hodit i při práci s jinými typy objektů než jen se soubory. Proto taky jistě nepřekvapí, že tyto zámky na řadě operačních systémů najdeme i mezi klasickými synchronizačními objekty, obvykle jako „reader writer lock“ (zámek čtenářů a písařů).

12.5 Typy souborů

Typy souborů jsou dobře známým prvkem, takže si jen stručně shrňme, co je jejich smyslem. Řada operačních systémů zavádí souborové typy proto, aby bylo jasné, co je obsahem daného souboru a co tedy se souborem je třeba či možno dělat. V praxi se osvědčily tyto způsoby identifikace typu souboru:

Přípona názvu
: Nejčastějším způsobem, jak identifikovat typ souboru, je přípona názvu, která je součástí názvu (část názvu za poslední tečkou). Kvůli omezením MS-DOSu se nejvíce rozšířily přípony tříznakové, ale dnes již jsou běžné i jiné (kratší i delší). Řada přípon má de facto standardní význam napříč operačními systémy a některé existují v tří- i víceznakové podobě (např. místo jpeg, mpeg či html často vidíme zkrácené tvary jpg, mpg, respektive htm).

Magic number
: Unixové systémy hodně používají tzv. „magické“ hodnoty. Jsou to čísla uložená na začátku souboru (přímo v datech), obvykle v prvních dvou bajtech. Typ souboru se pak pozná přečtením jeho začátku a zjištěním hodnot prvních dvou bajtů.
Tato metoda se používá i mimo Unix, např. MS-DOS a Windows mají ve spustitelných souborech značku „MZ“ (nebo „ZM“); textové soubory v některém z kódování Unicode (např. UTF-8 nebo UTF-16) mají značku vždy a bez ohledu na operační systém. Takové soubory tedy mohou mít nakonec libovolnou příponu a systém stejně dokáže poznat, co v souboru je.

Explicitní metadata
: Typ souboru může být explicitně uložen někam do souborového systému, obvykle někam do metadat souboru. Hovoříme o metadatech, protože jde o informace o souboru, ne samotný soubor. Je mnoho způsobů, jak toto realizovat, vždy je však problém s kompatibilitou a přenositelností, protože tato metadata se obvykle překopírováním souboru na jiné místo ztrácí (často se kopíruje obsah souboru, jméno, datum atp., ale ne speciální metadata). Proto používání metadat není zvláště v prostředí heterogenních počítačových sítí příliš praktické.

Mac OS X běžně používá přípony souborů a stejně jako Windows je také obvykle skrývá před uživatelem (což je jistý způsob ochrany, protože náhodné přejmenování přípony může vést k jistému druhu narušení bezpečnosti). Mac OS však používá také metadata, kde kromě typu souboru ukládá také identifikátor aplikace, která soubor vytvořila. Při pokusu otevřít později tento soubor pak systém ví, ke které aplikaci patří. (Tato „značka“ vzniká automaticky, když proces založí nový soubor na disku.)

Průvodce studiem

Pošlete-li soubor do tiskárny, vytiskne se správně jen tehdy, když jeho obsahem je text. Podporuje-li však operační systém typy souborů, může například automaticky chápat, že soubor JPEG je obrázek a pro jeho tisk použít speciální program pro tisk obrázků. Podobně při spouštění souborů systém může pomocí typů snadno rozlišit, zda spouštíte binární program (například tzv. „exe“ ve Windows) nebo soubor obsahuje jen data, která je pro spuštění potřeba zpracovat jiným programem.

12.6 Struktura souborů

Typy souborů mohou posloužit také k určení vnitřní struktury souboru. V současnosti však u operačních systémů převládá zjednodušený princip převzatý z Unixu, kdy systém podporuje jen jedinou strukturu souborů – každý soubor je obyčejný proud bajtů; význam těmto bajtům může dát jak operační systém, tak jednotlivé programy, ale data souborů jsou plně popsatelná touto posloupností bajtů.

Některé operační systémy podporují různé souborové struktury, např. pro spustitelné soubory může být výhodná jiná struktura, než je proud bajtů. Podporuje-li operační systém více souborových struktur, práce se soubory je pak jednodušší, aplikační programy si však musí vystačit

s omezeným výběrem. Praxe ukázala, že zobecnění na jednotnou a jednoduchou strukturu přináší flexibilitu. Například kopii proudu lze udělat tak, že načteme jeho obsah do pole a zapíšeme jej pak do nového souboru. Soubory složitějších struktur by takto kopírovat nebylo možné.

Mac OS má u každého souboru dva proudy: data a prostředky. V datovém proudu je klasický obsah souboru, např. kód programu, v prostředkovém proudu jsou další prostředky používané tímto programem, např. texty apod. Ostatní operační systémy sice taky často používají systém prostředků u spustitelných programů, umísťují je však obvykle někam do samotného datového proudu (např. na jeho konec, za kód programu).

Souborový systém NTFS používaný na Windows NT zobecňuje přístup známý z Mac OS tak, že umožňuje kromě datového proudu přiřadit k souboru libovolný počet dalších proudů. V praxi je však tato funkcionalita využívána velmi málo (např. Microsoft SQL Server). Samotný systém NT umožňuje do alternativního proudu NTFS ukládat poznámky o souborech (jako jméno autora, popis obsahu atp.), opět jde však o poměrně málo používanou funkcionalitu a některé hloupější náhrady Průzkumníka mohou při kopírování souborů tyto informace dokonce ztrácet (tím, že překopírují jen datový proud). Více o NTFS se dozvíme v sekci 14.4 na straně 161.

Fyzická struktura souboru

Kromě uživatelského pohledu mají soubory ještě svou fyzickou strukturu. Ukládání souborů na disk funguje obvykle po jistých větších blocích (sektorech či klastrech), o velikosti nejčastěji 512, 1024 nebo 2048 bajtů. K proudu tedy přistupujeme po jednotlivých bajtech, ale k disku se přistupuje jen po celých sektorech či klastrech. Velikost sektoru je nejčastěji 512 bajtů (z čistě historických důvodů), klastr je skupina sousedních sektorů a slouží ke zvětšení jednotky přístupu na disk obvykle na zmíněných několik málo KB.

Průvodce studiem

Hardwarová poznámka: Disky se dělí na sektory proto, že přímý přístup k disku by byl velmi složitý, protože by bylo obtížné zjistit, kde přesně v daném okamžiku čtecí hlavička je. Takto jsou na disku uloženy hlavičky sektorů a mezi nimi jejich data. Čím menší jsou sektory, tím méně dat se celkově na disk vejde, protože tam bude více hlaviček. Naopak větší sektory přinášejí vnitřní fragmentaci (viz popis fragmentace v kapitole 9.2 o organizaci paměti na straně 85) a opět tedy nižší využití.

Klastry (clustery) jsou zavedeny proto, že před přečtením sektoru je třeba jej najít. Hlavička čeká, až se neustále otáčející se disk natočí do správné polohy. Čtením více sektorů v řadě za sebou tedy ušetříme čas. Nevýhodou větších klastrů je opět větší vnitřní fragmentace, snižuje se tím však naopak vnější fragmentace a také nároky na evidenci přiděleného místa. Z pohledu fragmentace je tedy fyzická organizace disku podobná správě fyzické paměti.

Potřebujeme-li číst méně než celý klastr, operační systém vždy musí načíst celý klastr a uchovat si jej v bufferu. Při zápisu necelého klastru musí systém dokonce nejprve načíst celý klastr do bufferu, pak provést požadovanou změnu a pak zapsat celý klastr. Zápis tedy může být velmi pomalý, pokud se nepostupuje rozumně; při zápisu je velmi důležitá správná funkce diskové cache.

12.7 Přístup k souborům

Základním způsobem práce s proudem je *sekvenční přístup*. Čteme celý proud od začátku a jediná další operace je rewind (tzv. „přetočení zpět“, odpovídá seek na začátek). Při sekvenčním přístupu je operace seek omezena jen na přesun na začátek a na konec souboru. (Na konec se

přesouváme, chceme-li k němu připisovat další data.) Výhodou sekvenčního přístupu je, že je použitelný pro různá datová média a zdroje, včetně počítačové sítě či páskových zařízení.

Druhým často používaným způsobem práce s proudem je *přímý přístup*. Ten přistupuje pouze k vybraným částem souboru, kdy pomocí operace seek nejprve přesune pozici čtení/zápisu na požadované místo a pak s ním pracuje. Tento způsob práce je vhodný zejména pro diskové soubory, protože právě na disku lze snadno přesouvat pozice čtení/zápisu na libovolné místo.

Průvodce studiem

Říkáme, že disk umožňuje *náhodný přístup*, protože operací seek lze přejít na libovolné místo souboru. Přímý přístup k souboru pak funguje tak, že data v souboru se skládají z mnoha stejně velkých záznamů (datových struktur), kde program nejprve stanoví, kolikátý záznam v souboru potřebuje, tuto hodnotu vynásobí délkou záznamu v souboru a díky možnosti náhodného přístupu získá přímý přístup k požadovaným datům. Pojmy náhodný a přímý přístup tedy pak v praxi splývají a vyjadřují zde de facto totéž. S pomocí náhodného přístupu lze samozřejmě realizovat i řadu dalších vlastních způsobů přístupu k souboru.

12.8 Dělení disku a adresáře

V současné době bývají disky kapacitně již poměrně velké a dokáží pojmout také velké množství souborů, takže nikoho nepřekvapí, že tyto soubory se nějakým způsobem organizují pro lepší přehlednost. Je všeobecně známo, že základním nástrojem k organizaci souborů jsou adresáře, ze kterých lze vytvořit libovolnou stromovou strukturu. Platí jednoduché pravidlo, že existuje vždy právě jeden kořenový adresář a každý adresář může obsahovat soubory nebo další adresáře.

Větší disky však kromě adresářů dělíme ještě na samostatné části zvané partition (z angličtiny, česky jednoduše „část disku“ či oddíl). Toto dělení je na úrovni fyzického disku. Z pohledu operačního systému se disková paměť skládá ze svazků (anglicky volume) – každý svazek je jednou instancí nějakého souborového systému, např. ve Windows jsou svazky označovány klasicky písmeny s dvojtečkou (A:, C:, D: atd.). Je zřejmé, že svazek vytvoříme naformátováním nějaké části disku (oddílu) do nějakého souborového systému. Jednotlivé svazky jsou na sobě navzájem logicky zcela nezávislé, avšak mohou být na stejném fyzickém disku, dokonce může být více svazků v jednom oddílu (avšak tyto nejsou bootovatelné). Podobně některé operační systémy umožňují spojovat více částí (oddílů) dohromady pro vytvoření větších svazků, či připojovat celé svazky jako virtuální adresáře do jiných svazků. Tento poslední způsob je běžný na Unixových systémech, ale je možno jej použít také ve Windows (můžeme se tak zcela zbavit diskových písmen a onoho C: stylu cest, kořenový adresář pak také značíme jen lomítkem).

Připojování svazků do souborového systému se nazývá *mounting* či mountování (anglicky, sloveso mount). Přesné fungování této operace je samozřejmě závislé na konkrétním systému, ve Windows např. lze používat svazky i bez explicitního mountování ve tvaru \\?\Volume{GUID}\, kde GUID je jednoznačný identifikátor svazku.

Průvodce studiem

Unix umožňuje připojit svazek do libovolného adresáře. Všechny připojené svazky pak jsou součástí jednoho adresářového stromu. Windows NT jako výchozí používá písmena (C: aj.), z historických důvodů (pozůstatek po MS-DOSu). Zajímavé také je, že v Unixu se běžně různé části systému a data dělí do více svazků, zatímco Microsoft dává přednost používání jednoho svazku pro vše (C:). NT přitom interně pracuje na podobném principu jako Unix, oba systémy připojují svazky pomocí tzv. bodů připojení (anglicky *mount point*). Mount point je cesta, která se prezentuje jako adresář, ale ve skutečnosti je branou do svazku.

Mount point lze zařadit kamkoliv do adresářového stromu a umožnit tak de facto přístup odkudkoliv kamkoliv. (Ve Windows se používání této funkcionality téměř nerozšířilo.)

Adresářové operace

Běžné adresářové operace jsou všeobecně známé, proto si uveďme jen stručný výčet: nalezení souboru, založení souboru, odstranění souboru, získání seznamu souborů, přejmenování souboru, procházení adresářovým stromem.

Adresářová struktura

Existuje více možností, jak organizovat adresáře. Nejběžnější, jak už jsme zmínili, je stromová struktura. Ta se dnes používá zejména z historických důvodů. Vzhledem k tomu, že pouze obecná grafová struktura, kde jsou povolené libovolné vazby mezi adresáři a soubory, je jednoznačně flexibilnější, ale také mnohem složitější na údržbu systému, adresářová struktura se jeví jako nejlepší z hlediska většiny aplikací. Kromě složitější grafové struktury lze adresáře organizovat i jednodušším způsobem. Nejjednodušší systémy nepoužívají adresáře vůbec, takže všechny soubory jsou na stejné úrovni, případně umožňují jen jednu úroveň adresářů, kde jiný než kořenový adresář sám nemůže další adresáře obsahovat. Jednoduché víceuživatelské systémy pak mohou dělit soubory podle uživatelů, takže na nejvyšší úrovni je skrytá adresářová úroveň, kde každý adresář patří jednomu uživateli systému. Výsledkem pak může být adresářová dvojúrovňová struktura – hlavní systémový adresář slouží k rozdělení dat uživatelů, každý z nich pak může mít jednu svou adresářovou úroveň. Všechny tyto varianty pak lze zobecnit do stromové struktury.

Výše zmíněná grafová struktura je dalším zobecněním všech ostatních typů struktur. Moderní stromové souborové systémy umožňují při zachování stromové struktury definovat libovolné dodatečné vazby, čímž získáváme možnost de facto grafové organizace (tzv. hard link a soft link). Výhodné je to však jen pro některé speciální aplikace či situace.

Průvodce studiem

Hard link je ukazatel na fyzické tělo souboru na disku. Každý pojmenovaný soubor na disku je tedy de facto hard link. Moderní souborové systémy pak umožňují vytvářet více hard linků k jednomu souboru. Unixové systémy k tomu mají příkaz `ln`, NT má příkaz `fsutil hardlink` a od verze Vista také nově `mklink`.

Soft link (často také nazýváno symbolic link či symlink) je odkaz na jiný soubor, který je specifikován jeho cestou. Soft link může odkazovat i na jiný soft link. V Unixových systémech se soft link vytváří opět příkazem `ln` (s jinými parametry). NT podporuje soft linky jen na úrovni shellu, tzn. samotný operační systém je nepodporuje a lze je používat jen v Průzkumníkovi – jedná se o soubory s příponou lnk (zástupce) a url (zástupce internetové adresy). Náhrady shellu (např. různé „commandery“) podporují tyto zástupce také, ale obecně v libovolných programech jej používat nelze bez dalších úprav.

Pro zajímavost dodejme, že systém NTFS ve skutečnosti soft linky podporuje, ale Windows nemá žádný nástroj, jak s nimi pracovat [Wiki]. Od verze Vista již najdeme podporu soft linků přímo na úrovni souborového systému, pracuje se s nimi standardním příkazem `mklink`. Informační zdroje o podpoře linků ve Windows jsou často dosti matoucí, naštěstí se jedná o funkci ve Windows velmi málo používanou. Za zmínku stojí i fakt, že soft link má vlastní ACL (viz níže).

Největší problém při používání linků je omyl uživatele, který ve snaze smazat link ve skutečnosti smaže obsah cílového adresáře či souboru.

Při práci v adresářové struktuře pracuje proces vždy v nějakém *aktuálním adresáři*. Při pokusu o přístup k souborům pak má v daný okamžik přístupné jen soubory aktuálního adresáře, soubory ostatních adresářů pak musí kromě jména identifikovat také určením *absolutní cesty* k jeho adresáři, nebo *relativní cesty* vztažené k absolutním adresáři. Pojem *cesta* přitom obvykle označuje přímo konkrétní soubor, ne jen jeho adresář, takže ještě můžeme používat termíny *absolutní adresář* a *relativní adresář*. Pokud budeme chápat adresář jako speciální typ souboru, pak cesta může označovat také adresáře.

Ve stromové struktuře bývá zvykem cestu označovat jako posloupnost jmen adresářů a souboru, kde jednotlivé názvy oddělujeme lomítkem. Přitom ve Windows obvykle zpětným lomítkem \, ve většině ostatních systémů obyčejným /. Rozlišení absolutní a relativní cesty se dělá také v různých systémech různě, v Unixu je absolutní cestou každá začínající lomítkem, ve Windows pak absolutní cesta začíná dvěma lomítky nebo jménem disku (písmenem s dvojtečkou) a lomítkem. MS-DOS a některé verze Windows používají ještě další zvláštnost: Aktuální adresář je zvlášť pro každý svazek, takže při přechodu mezi svazky tam a zpět se zachovává původní adresář. Kromě relativní a absolutní cesty pak lze používat i relativní cesty vztažené k aktuálnímu adresáři jiného svazku, ty se označují pomocí jména disku nenásledovaného lomítkem. Tento způsob zápisu cest je však poněkud matoucí a v praxi se příliš nepoužívá.

Průvodce studiem

Dodejme ještě, že ve Windows je délka cesty omezena na 260 znaků, přičemž u národních znaků se ještě v jednotlivých případech může lišit, jak přesně se do celkového součtu počítají. Toto omezení není dáno přímo souborovým systémem, ale jen interními funkcemi zpracovávajícími cesty. Ty však můžeme vypnout (přesněji řečeno obejít či přeskočit). Začíná-li ve Windows cesta znaky \\?\, musí to být absolutní cesta a v kódování unicode (předaná do příslušné unicode verze systémové funkce), délka cesty může být ve Windows NT až 32767 znaků. Tento tvar je možno použít pro lokální i síťové cesty, tj. např. \\?\C:\Windows nebo \\?\server\shared.

Hledání spustitelných souborů

V systému obvykle máme řadu programů (a jejich spustitelné soubory). Menší programy používané na příkazové řádce můžeme spouštět zadáním jejich jména. Řada systémů toto řeší pomocí systémové proměnné (jménem path) obsahující seznam adresářů, ve kterých má hledat programy ke spouštění bez zadaného adresáře. Toto chování je společně pro Unix, Linux i Windows.

Macintosh, jak víme, umožňuje otvírat libovolné dokumentové soubory díky tomu, že si u každého pamatuje, ke kterému programu patří. Používá jinou strategii: Systém si uchovává adresy všech spustitelných programů ve zvláštním seznamu a při pokusu otevřít dokument hledá ve svém seznamu, kde má program jména uvedeného v záznamu otvíraného souboru. Windows pak stejnou funkcionalitu zajišťuje jiným způsobem: Programy si mohou zaregistrovat v systému jimi používané přípony souborů. Samy přitom uvedou, jakým programem je chtějí otvírat. Tyto registrace jsou uloženy v systémovém registru. Systém pak při pokusu o otevření dokumentového souboru podle jeho přípony najde potřebný program. (Systémovou proměnnou path Windows používá jen pro programy či příkazy na příkazové řádce.)

12.9 Sdílení souborů mezi uživateli

V sekci 12.4 na straně 131 jsme hovořili o zamykání souborů. Sdílení souborů je de facto opačná operace k zamykání a jedná se o poměrně důležitou a užitečnou funkcionalitu moderních systémů. (Jedno specifické využití sdílení jsou soubory mapované do paměti, které jsme zmínili

již v několika předchozích kapitolách.) Nyní však nahlédneme na sdílení souborů na úrovni uživatelů, nikoliv procesů.

Máme-li víceuživatelský systém, což je dnes již zcela běžné, pak uživatelé zřejmě nějakým způsobem sdílejí společný souborový systém. Unixové systémy používají jako základ model, kde každý soubor nese informaci o majiteli (to je obvykle uživatel, který jej vytvořil) a tři masky přístupových práv zvlášť pro tohoto uživatele, jeho skupinu a ostatní uživatele.

Sdílení souborů je aktuálním tématem také v oblasti počítačových sítí. Základním prostředkem sdílení je dobře známý protokol ftp, nověji pak také (velmi populární) protokol http; ani jeden z těchto prostředků není třeba představovat. Alternativou jim jsou systémy umožňující sdílení souborů pomocí mapování vzdálených svazků do souborového systému. Všechny tyto modely jsou typu *klient–server*, kde jeden počítač (server) nabízí soubory a ostatní počítače (klienti) jej využívají. V posledních letech se velmi rozšířil také model *peer–to–peer* (ve zkratce P2P), který se umožňuje sdílení souborů přímo mezi uživateli bez potřeby serveru. Jako příklad uveďme systémy Napster (již nefunkční), Kazzaa, Torrent či Direct Connect. Tyto souborové P2P sítě nabízejí větší flexibilitu a jsou dnes v některých oblastech nasazení jednoznačně populárnější než sítě klient–server, i když technicky se jedná o daleko složitější systémy. To je právě z důvodu neexistence nějaké centrální autority (serveru), což znesnadňuje například navazování spojení, vyhledávání souborů apod. Různé P2P sítě tyto problémy řeší částečnou centralizací některých úloh, empiricky však bylo dokázáno, že úspěch P2P řešení je mimo jiné nepřímo úměrný úrovni centralizace.

U všech typů síťových sdílení je třeba řešit problémy zabezpečení a ochrany komunikace, kde ve stručnosti lze možné problémy rozdělit do dvou oblastí: ztráta či narušení identity uživatele a ztráta či narušení dat. Tato problematika je však nad rámec této kapitoly a de facto i tohoto studijního kurzu Operační systémy.

Sémantika konzistence

Sémantika konzistence přesně specifikuje chování systému při současné práci více procesů s jedním souborem, jmenovitě popisuje zejména to, kdy jeden uživatel uvidí změny provedené jiným uživatelem.

Např. v Unixu jsou změny souboru okamžitě viditelné ostatními procesy, které používají tentýž soubor. Jedním z možných způsobů sdílení je sdílení aktuální pozice souboru, kdy čtením souboru jedním procesem se posouvá pozice čtení i ve všech ostatních procesech. Jinou alternativou může být sémantika immutable–shared–file (česky: nemutující soubor), kde zahájením sdílení dojde k zákazu změn dat a jména souboru. Sdílený soubor je tedy neměnný, což umožňuje především jednodušší implementaci pro operační systém.

12.10 Ochrana souborů

Soubory zabezpečujeme proti fyzické chybě systému (média, disku atp.), pak hovoříme o *spolehlivosti*, a proti nesprávnému přístupu, pak hovoříme o *ochraně*. Spolehlivost zajišťujeme nejčastěji vytvářením záložních kopií souborů či přímo používáním automatizovaných systémů pro tento účel, např. zálohovacích páskových zařízení či diskových polí RAID (viz kapitola 14.5 na straně 163). Ochranu lze zajistit různými prostředky různě. Silberschatz [SGG05] například uvádí postup: „Vyjmout disketu se souborem a zamknout ji do zásuvky ve stole.“ Podívejme se však spíše na softwarová řešení ochrany.

Typy přístupu

Potřeba ochrany přímo vyplývá z možnosti přistupovat k souboru. Jak je zřejmé z příkladu s disketou zamčenou ve stolní zásuvce výše, ochrany docílíme právě kontrolou či omezením

přístupu. Nejjednodušším typem ochrany je zcela znemožnit, aby uživatelé mohli sdílet soubory mezi sebou – potom žádnou ochranu nemusíme řešit. Tento způsob však není optimální.

Ochrana je obvykle řešena pomocí kontroly přístupu. Je definována množina možných typů přístupů a u každého souboru jsou pak definována pravidla určujících, kdo může či nemůže provádět daný typ přístupu k souboru. Základními typy přístupu jsou čtení, zakládání a zápis, spouštění, připisování na konec souboru (append), mazání, prohlížení vlastností, přejmenování. Jednotlivé operační systémy pak mohou k dílčím operacím přistupovat jinak, když obvykle několik příbuzných typů přístupu je pro jednoduchost zahrnuto dohromady pod jednu položku.

Kontrola přístupu

Máme-li definovány typy přístupu, pak kontrolu přístupu ke každému souboru a adresáři realizujeme pomocí ACL (access control list). ACL je seznam uživatelů a jim povolených typů přístupu k danému souboru (jinými slovy seznam povolených operací se souborem). Při práci se soubory se pak kontrolují jejich ACL.

ACL je flexibilní, ale může být pomalé a paměťově náročné, protože uchovávání velkého množství dat v ACL komplikuje jejich vytváření (Jak vůbec zjistíme všechny uživatele systému, když chceme soubor zpřístupnit všem?) i používání (ACL je třeba uložit do záznamu o souboru v adresáři, tím se adresářové záznamy neúměrně zvětšují a jednotlivé záznamy potřebují proměnlivou velikost.). Unixové systémy tyto potíže řeší používáním zjednodušeného systému oprávnění, kde soubor má jen tři záznamy v ACL pro majitele souboru, jeho skupinu a ostatní uživatele. (Každý uživatel systému pochopitelně patří do jedné z těchto tří skupin.) ACL tohoto typu pak zabírá jen několik málo bajtů paměti (3 bity pro každý typ přístupu). ACL se nastavuje příkazem `chmod` (vyzkoušejte!). Moderní verze Unixových systémů pak umožňují k tomuto základnímu systému přidávat i plnohodnotné ACL u souborů, kde je to třeba. Tím je zajištěna plná flexibilita a úspora místa zároveň.

Windows 95 nepodporuje ochranu souborů vůbec, stejně tak MS-DOS a Mac OS do verze 9.x [SGG05]. Mac OS je specifický tím, že pokrývá jen malý trh a není kompatibilní s jinými systémy v poměrně velkém množství detailů. Některé zdroje dokonce uvádějí, že Mac OS má celou řadu vlastností, které jsou v systému z úplně jiných důvodů (by design), ale v důsledku v kombinaci s malou penetrací Macintoshů příznivě ovlivňují praktickou bezpečnost celého systému. Novější Mac OS X je pak již verzí Unixu (doslova), takže pro něj platí totéž, co pro Unix. Opačná situace je pak u systémů Microsoftu – příliš flexibilní design bez ochran v kombinaci s velkou penetrací těchto systémů způsobil boom virů a jiného škodlivého softwaru.

Windows NT, ačkoliv také od Microsoftu, nabízí propracovaný systém ochrany souborů. Používá plnohodnotné ACL (čili bez zjednodušení) a umožňuje nastavit sdílení ACL s adresářem, ve kterém je soubor umístěn. Ve většině praktických situací tedy nastavíme ACL jen u několika adresářů a všechny jim podřízené součásti adresářového stromu již vlastní ACL nemají, takže to nezabírá žádný diskový prostor. NT také na rozdíl od Unixu definuje velmi mnoho typů souborových přístupů a také umožňuje kromě ACL pro povolení přístupu definovat seznamy zamezení přístupu DACL (*Denied ACL*). V NT se ACL nastavuje obvykle v GUI (vyzkoušejte!) nebo také příkazem `cacls`.

Při kolizi oprávnění, kdy ACL například definuje nějaké oprávnění pro uživatele, ale jiné pro jeho skupinu, se jednotlivé operační systémy chovají různě. Například Solaris upřednostňuje více specifické údaje (tj. zde o uživateli) před obecnými (tj. zde o skupině). NT všechna povolující oprávnění slučuje, ale vyšší prioritu má vždy zákaz – je-li v DACL zákaz na libovolné úrovni, nelze jej pomocí ACL nijak přebít.

Shrnutí

Zahájili jsme blok kapitol věnovaných správě diskového prostoru či obecně sekundární paměti.

Celá kapitola se věnovala seznámení se základními pojmy této oblasti na vyšší úrovni abstrakce, tj. soubor, adresář, proud atp. Probráno bylo zamykání souborů, typy souborů, sdílení souborů a závěr kapitoly byl věnován studiu zabezpečení na úrovni souborů. Řada těchto pojmů je čtenáři či studentovi jistě dobře známá, jedná se však o jedno ze základních témat studia operačních systémů, proto je mu zde věnován patřičný prostor. Běžně používané operační systémy se na úrovni souborového systému dosti podobají, model adresářů a souborů je značně zažitý a systém pracující na jiném principu by asi dnes neměl šanci uspět. K tomu přispívá i rozvoj internetu, kde také řada protokolů pracuje na bázi souborů či proudů. Problematiku souborových systémů v počítačových sítích jsme zde však nijak hluboce nezkoumali, v následujících kapitolách se spíše zaměříme na implementační otázky souborových systémů jednotlivých počítačů.

Pojmy k zapamatování

- soubor
- souborové operace
- zamykání souborů
- typ souboru
- magic number
- metadata souboru
- přístup k souboru – sekvenční, přímý, náhodný
- proud
- oddíl a svazek
- mounting
- adresář
- aktuální adresář
- absolutní a relativní cesta
- hard link, soft link
- sdílení souborů
- klient–server, peer–to–peer (P2P)
- sémantika konzistence
- spolehlivost a ochrana (souborového systému)
- typ přístupu, kontrola přístupu
- ACL (access control list)
- DACL (denied access control list)

Kontrolní otázky

1. *Co to je zamykání souborů, jaké typy zamykání známe a k čemu je to dobré? Má smysl, aby systém umožňoval zamykat i pouze části souborů? Vysvětlete.*
2. *Jmenujte možné způsoby, jak bývají v souborovém systému reprezentovány typy souborů.*
3. *Navrhněte, jak řešit mazání adresářů, které obsahují nějaké soubory?*
4. *Jak systém rozhodne o přístupu, když ACL souboru obsahuje více protichůdných záznamů?*
5. *Co to jsou alternativní datové proudy? K čemu se používají?*
6. *Přímý přístup k souboru je na první pohled dokonalejší, proč tedy vlastně existuje i sekvenční přístup? Je to jen historický prvek, nebo se sekvenční přístup používá i dnes? Vysvětlete.*
7. *Jak souvisejí oddíly a svazky?*
8. *Jak systém NT rozhodne o přístupu, když je stejný záznam v ACL i DACL.*
9. *Popište systém ochrany souborů v MS-DOSu.*

10. *Co je to sémantika konzistence? Uveďte také nějaký konkrétní příklad.*
11. *Jak souvisí sdílení souborů se zamykáním?*
12. *Vysvětlete rozdíl sdílení souborů v sítích typu klient–server a peer–to–peer.*
13. *Popište výhody a nevýhody peer–to–peer sítí.*
14. *Vysvětlete, jak systémy bojují s problémem velké datové náročnosti ACL (mnoho uživatelů $\longrightarrow$ mnoho záznamů $\longrightarrow$ zabírá mnoho paměti).*
15. *Popište detailně základní model kontroly přístupu k souborům v Unixu.*
16. *Jmenujte alespoň jeden příklad, kdy využijete nástroje pro obecné grafové vazby v systémech se stromovou adresářovou strukturou.*
17. *Vysvětlete příkazy open() a close().*
18. *Uveďte konkrétní příklad sekvenčního a náhodného přístupu k souboru.*
19. *Některé systémy umožňují, aby uživatel dostal oprávnění pracovat s adresáři jako s běžnými soubory. Vysvětlete, jaká tím vzniká bezpečnostní díra.*
20. *Mějme systém s 2000 uživateli. Jak v Unixu nastavíte 1990 z nich přístup k souboru?*
21. *K předchozí otázce uveďte, jak lze totéž jednoduše řešit v NT.*

Cvičení

1. Vyzkoušejte si v praxi zamykání souboru pro čtení a pro zápis.
2. Vyzkoušejte si v Linuxu či Unixu změnu ACL pomocí příkazu `chmod`. Vyzkoušejte si totéž nastavit programově.
3. Vyzkoušejte si ve Windows nastavení ACL a DACL. Ověřte si, že při kolizi má DACL vyšší prioritu než ACL.

13 Implementace souborového systému

Studijní cíle: V předchozí kapitole jsme si představili princip souborového systému, nyní si přiblížíme jeho možnou strukturu a především probereme otázku jeho implementace. Tu je možno studovat jak teoreticky, tak prakticky, protože téměř každý z běžně používaných operačních systémů používá svůj vlastní souborový systém. V této kapitole se na implementaci souborového systému podíváme uceleně, především se tedy zaměříme na principy používané při implementaci pojmů, se kterými jsme se již seznámili (adresářová struktura atp.).

Klíčová slova: raménko, MBR, FCB, adresář, soubor, žurnálování

Potřebný čas: 180 minut.

13.1 Struktura disku

Popišme si na úvod strukturu pevného disku. Pro základní představu postačí znalost disket, běžné 3.5" diskety fungují přibližně na stejném principu jako pevné disky. Na rozdíl od diskety, kdy v počítači je disketová mechanika, zatímco samotné diskety jsou přenosné, u pevného disku je celé zařízení napevno v počítači. Přitom „pevnost" disku není dána tím, že je pevně v počítači, ale tím, že magnetický kotouč a mechanika s ním pracující jsou pevně spojeny. Právě díky tomuto řešení je zabráněno vnikání cizích částic (nečistot) do mechaniky, díky čemuž pevné disky mohou nabídnout daleko vyšší kapacity a spolehlivost než přenosné diskety.

Disk obsahuje plotny.

Disk obvykle obsahuje několik *ploten* (anglicky platter), nejčastěji to jsou dvě nebo tři. Plotna je magnetický kotouč (připomínající kotouč v disketě), na kterém jsou uložena data. Každá plotna má pochopitelně dvě strany a vlastní hlavy, které magnetickou informaci čtou/zapisují. Plotny se neustále otáčejí a hlavy jsou připojeny na raménku, které je posuvné a nastavuje tak hlavu nad určitou *stopu* disku. Stopa je de facto kružnice. Na rozdíl od disket, kde na každé stopě obvykle bývá stejné množství dat, u pevných disků obvykle na vnitřních stopách bývá dat méně, aby byla data uložena s rovnoměrnou hustotou. Samotná data, jak víme, jsou uložena po blocích, obvykle o velikosti 512B. Tyto bloky se nazývají sektory a je to vždy část jedné stopy. Tím, že disk má více ploten nad sebou a na všech jsou hlavy, při práci s diskem obvykle pracují všechny hlavy najednou, takže pracujeme s určitou stopou na všech stranách všech ploten. Taková skupina stop se nazývá *cylindr* (anglicky cylinder). U pojmů stopa a cylindr někdy dochází k záměně či splývání, není to však z pohledu našeho studia podstatný problém.

Cylindr je skupina stop.

Podrobnější (tedy hardwarové) informace o fungování disků jsou mimo rámec tohoto předmětu (viz předmět Struktura počítačů), na závěr této sekce se však ještě zastavíme u problematiky rychlosti disků. Rychlost disku je daná několika faktory: Prvním je *přenosová rychlost*, to je rychlost toku dat mezi diskem a počítačem. Jde tedy o rychlost datového kanálu (kudy data „tečou"). Rychlost čtení a zápisu jsou rychlosti, kterými disk skutečně čte či zapisuje data, pokud pracujeme s po sobě jdoucími sektory a stopami. Reálnou rychlost disku velmi ovlivňuje také tzv. *seek time*, což je průměrná doba potřebná k nastavení hlav nad určitou stopu. Dodejme ještě, že rychlost disků je v současnosti velmi malá ve srovnání s rychlostí operační paměti a procesoru, takže se obvykle vyplatí i výpočetně či paměťově náročné algoritmy řešící snížení počtu přístupů na disk, stejně jako algoritmy řešící minimalizaci počtu a vzdáleností přesunů raménka mezi stopami.

13.2 Master boot record

Některé vlastnosti mají reálné systémy společné, obvykle proto, že jsou určeny pro stejný typ počítače, nebo pro stejný typ datového média. Např. první sektor pevného disku se na počítačích PC nazývá *master boot record* (MBR) a obsahuje informace o členění disku (anglicky partitioning) na jednotlivé části (anglicky partitions). Každá část má typ: Je buď primární, nebo sekundární. Primární část přímo odpovídá svazku a může být použita k bootování operačního

systému. Sekundární část lze dále rozdělit na více svazků, které však nejsou bootovatelné. Zde popsaný princip MBR je obvykle používán i na jiných počítačích než PC, důvodem je samozřejmě dominantní postavení PC mezi počítači. Rozdělením disku na více svazků vznikne ve Windows více „písmen“ disků, v Linuxu se pro změnu více svazků používá k přesnějšímu oddělení samostatných částí adresářového stromu. Např. adresáře uživatelů v `/home` jsou na jiném svazku než dočasné soubory v `/tmp` – výhodou i nevýhodou pak může být fakt, že jednotlivé svazky mají vždy pevnou velikost.

Průvodce studiem

Windows při startu systému (a kdykoliv později, kdy objeví nový svazek – např. USB disk) připojí všechny svazky na první volná písmena (počínaje C:, protože A: a B: jsou vyhrazeny pro disketové mechaniky). Písmena je možno pohodlně změnit či úplně zrušit ve správci disků, kde lze také připojit svazky do stromu jako adresáře jiných svazků (ve stylu Unixu). Při odpojení svazku/zařízení si systém pamatuje jeho minulé písmeno a příště opět použije to stejné (týká se pevných i přenosných disků), pokud ho ovšem dokáže podle jména či čísla příště opět identifikovat. (Např. každému USB disku můžete přiřadit jiné písmeno, přestože je budete střídavě zapojovat do stejného USB portu.)

Samotný proces bootování a podrobnější popis struktury MBR jdou nad rámec úvodního kurzu Operační systémy. Opačnou situaci, kdy namísto dělení částí disku na svazky naopak jeden svazek je na více diskových částech, budeme diskutovat v sekci 14.5 na straně 163.

13.3 Obecná struktura svazku

Bez ohledu na konkrétní formát, každý svazek obvykle sestává z vlastní bootovací hlavičky používané při startu operačního systému z tohoto svazku, dále pak obsahuje nějaká řídicí data svazku (tj. informace týkající se svazku jako celku), informace o adresářové struktuře a informace o jednotlivých souborech. Každý soubor je popsán svým FCB (file control blok), adresář pak obsahuje odkazy na tyto FCB. Jméno souboru (a např. také datum) je obvykle v adresáři, zatímco FBC popisuje tělo (data). To vše samozřejmě platí především na běžných počítačích, neboť čistě technicky může disk a potažmo svazek mít libovolný obsah. Některé (hlavně starší a jednodušší) souborové systémy nemají samostatně uložené adresáře, takže FCB pak pochopitelně musejí obsahovat i jméno a další údaje, které by byly v adresáři.

13.4 Data v paměti

Operační systém si v paměti uchovává především informace o tom, který svazek a jak je připojen (namountován) do systému – dohromady se pak obvykle všechny připojené svazky tváří jako jeden společný souborový systém. Operační systém tedy uživatelským procesům poskytuje jakýsi jeden společný virtuální souborový systém, obvykle v podobě adresářového stromu a jeho vlastnosti jsou takové, aby pokryly vlastnosti všech fyzických souborových systémů, které reprezentuje. Dalším důležitým prvkem je disková cache, ve které operační systém uchovává nejčastěji či nejposledněji používané informace o souborech či svazcích za účelem zrychlení práce se soubory.

Průvodce studiem

Disková cache výrazně zrychluje práci s diskem. I kdybychom nikdy opakovaně nečetli stejné soubory, díky cache se práce s diskem zrychlí už už proto, že v ní operační systém může uchovávat informace o adresářové struktuře apod. Při zápisu souborů cache pomáhá

ještě více – moderní systémy umějí kromě jednoduchého principu *write through*, kdy zapisovaná data zůstávají v diskové cache pro případ, že by se měla v blízké době opět číst, také princip *write back*, kdy jsou data zapisována pouze do cache a teprve později jsou zapisována na disk. Jelikož rychlost počítače je výrazně vyšší než rychlost disku, write back cache se projeví velmi zřetelně.

Konkrétní implementace diskové cache je samozřejmě věcí konkrétního operačního systému.

Z principu práce se soubory popsaného v předchozí kapitole je zřejmé, že operační systém si také uchovává seznam informací o otevřených souborech a buffery pro každý z nich. Každý otevřený soubor má tedy v paměti jakousi obdobu FCB. Buffer slouží pro práci s proudem daného souboru – systém si zde uchovává část dat souboru, protože číst z disku lze jen po celých klastrech, ale jednotlivé procesy mohou od operačního systému žádat data po libovolně velkých blocích a načíst opakovaně stejné klastry by bylo značně neefektivní (pomalé). Operační systémy podporující sdílení souborů či duplikaci procesů také obvykle umožňují sdílet tyto seznamy otevřených souborů – systém může například evidovat zvlášť lokální otevřené soubory procesu a zvlášť otevřené soubory sdílené více procesy. Ke každému otevřenému souboru existuje jednoznačný identifikátor (ve Windows file handle, v Unixu file descriptor). Všechny funkce, které lze volat pro práci se soubory, pak požadují jako jeden z argumentů právě identifikátor otevřeného souboru. (Výjimkou je pochopitelně funkce open(), která soubor otevře a identifikátor vrací.)

Jak víme, mnohé operační systémy používají princip souborů a proudů i pro další účely (přístup k hardwarovým zařízením, síťová komunikace aj.). V tom případě musí mít záznam o souboru takový formát, aby dokázal popsat všechny typy otevřených souborů, ať už se odkazují na soubory na disku, či jiné zdroje dat. Je to tedy specifická datová struktura, která se váže více ke konkrétnímu operačnímu systému než k souborovému systému. Operační systémy používající soubor jako „nástroj na všechno“ se obvykle inspirují systémem Unix, kde se tento princip nejvíce rozšířil (či prosadil, či osvědčil). Podobně jako „všechno je soubor“, při práci se soubory nezáleží ani na konkrétním použitém souborovém systému – operační systém pomocí několika úrovní abstrakce zpřístupňuje uživatelským procesům všechny soubory stejně (např. CD-ROM a disk používají zcela odlišný systém ukládání dat, ale s otevřenými soubory se pracuje v obou případech úplně stejně).

13.5 Adresáře

Způsob implementace adresářů může výrazně ovlivnit výslednou rychlost práce se souborovým systémem a také bezpečnost, jde proto jednu z naprosto klíčových věcí.

13.5.1 Lineární seznam

Lineární model implementuje adresáře jako obyčejné pole záznamů pevné délky popisujících jednotlivé soubory. Při potřebě přidat další soubor se na konci přidá další záznam. Při naplnění klastru se přidá další. (Zde je vidět, že adresář skutečně připomíná běžný soubor, protože to je také především blok dat na disku.) Výhodou tohoto řešení je velmi snadná implementace. Nevýhodou je pak časová náročnost operací. Pro vytvoření souboru je nutné projít celý adresář, abychom zjistili, zda se dané jméno souboru již nevyskytuje. Pro smazání souboru je rovněž nutno projít celý adresář a mazaný soubor označit jako smazaný. Po smazaných souborech zůstává v adresáři nevyužité místo. Alternativou může být přesunutí záznamu posledního souboru do uvolněného místa, tím se však mění pořadí souborů v adresáři, což může být nežádoucí. Obecně téměř všechny operace vedou k problému nalezení souboru podle jména, což lze realizovat jen pomalým lineárním prohledáváním celého adresáře (složitost $O(n)$). Operace

s adresáři se opakují velmi často, proto jsou právě adresářová data nejčastějším objektem cachování – klastry obsahující adresáře jsou zřejmě úplně nejčastěji používanými částmi disku, proto je uchování jejich kopií v primární paměti počítače zvláště výhodné. Data adresářů se do cache načtou vždy při prvním přístupu k jednotlivým adresářům, takže první přístup je běžně znatelně pomalejší než přístupy následující.

13.5.2 Hashovací tabulka

Složitější, ale lepší alternativou je model používající hashovací tabulku. K výše popsané lineární struktuře je přidána hashovací tabulka sloužící k rychlému nalezení souboru podle jména. Výhodou tohoto řešení je výrazné zrychlení většiny operací (složitost $O(1)$), nevýhodou jsou pak klasické těžkosti spojené s hashováním: tabulka musí mít určitou pevnou velikost dle hashovací funkce, zatímco různé adresáře mohou obsahovat velmi rozličná množství souborů. Hashovací tabulky lze implementovat různými způsoby a v případě adresářů nejde o nic zvláštního (viz např. učebnice [Knu98] nebo [Več04]).

13.6 Alokace diskového prostoru

Podobně jako u správy operační paměti je alokace prostoru také u správy diskového prostoru jednou z hlavních funkcí a používá i podobné algoritmy.

13.6.1 Souvislá alokace

Souvislá alokace funguje stejně jako správa paměti v systému MS-DOS. Diskový prostor je rozdělen na klastry, které jsou seřazeny (tj. existuje určité lineární pořadí klastrů). Poloha souboru na disku je pak určena počátečním klastrem a počtem zabraných klastrů. (Tyto dvě hodnoty jsou obvykle uloženy v adresářovém záznamu o souboru.)

Implementace tohoto algoritmu je jednoduchá. Výhodou je rychlé sekvenční čtení disku, protože při čtení není třeba přesouvat čtecí hlavu na jiné pozice. Přidání nového souboru však vyžaduje nalezení dostatečně velkého prostoru a zejména pak je potřeba předem znát velikost budoucího souboru, jinak může nastat problém v okamžiku, kdy se alokovaný prostor zcela zaplní a potřebujeme soubor ještě dále zvětšit. Tento problém je identický se situací při souvislé alokaci paměti, včetně alternativ řešení a vnější fragmentace (např. MS-DOS, viz kap.9.2 na straně 85).

13.6.2 Spojové seznamy

Problémy souvislé alokace lze řešit např. použitím spojových seznamů (linked list). Soubor v tom případě může být uložen v libovolné posloupnosti klastrů, kde každý ukazuje na následující klastr v řadě. Pro příklad: Při velikosti klastru 512 bajtů bude 510 bajtů obsahovat data a zbylé 2 bajty budou číslo klastru obsahující následující část souboru. V adresářovém záznamu o souboru pak stačí uložit číslo počátečního klastru. Čtení souboru je v tomto případě opět velmi rychlé, protože načtením klastru ihned zjistíme číslo klastru následujícího. V případě fragmentace souborů může být čtení zpomaleno o přesuny čtecí hlavy, ale to jen v těch případech, které by u souvislé alokace vedly k neřešitelným situacím. Čili technika spojových seznamů je oproti souvislé alokaci lepší.

Nevýhodou spojových seznamů je problematický náhodný přístup. Chceme-li najít určitou část souboru na disku, je třeba sekvenčně projít celý soubor. Další nevýhodou je chování při chybě na disku: Jeden vadný klastr způsobí nečitelnost celého zbytku souboru.

Variaci spojového seznamu používá MS-DOS: Tabulka FAT umístěná na začátku disku obsahuje tolik buněk, kolik je klastrů, kde v každé je výše popsaný odkaz na následující klastr v řetězu

souboru. Samotné klastry na disku pak již obsahují jen data souboru. Nejde tedy o zcela jiný systém, jen o jinou implementaci spojového seznamu. Výhodou tohoto řešení je, že nalezení kterékoliv části souboru je mnohem rychlejší, neboť nám k němu stačí načíst FAT a projít odkazy, zatímco klastry obsahující nepotřebné části dat načítat nemusíme. Další výhodou je snadné hledání volného místa. Výše uvedená souvislá alokace, ani klasický spojový seznam totiž nedokáží nějak efektivně najít volné místo na disku bez toho, aby načetli data všech souborů. Alternativou (platnou pro každý systém) může být tzv. bitová mapa volných klastrů, což je bitové pole, kde jednička či nula na každé pozici značí obsazení či uvolnění příslušného klastru. Nevýhodou bitové mapy je, že její udržování stojí výpočetní čas, způsobuje dodatečné čtení a zápisy disku.

Mapa volných klastrů i FAT tabulka nesou jisté informace o klastrech. Pokud dojde k selhání systému (např. v důsledku vypnutí proudu) v okamžiku zápisu, pak FAT tabulka či bitová mapa neodpovídají skutečnému obsahu disku, přičemž důsledkem je dle konkrétní situace téměř libovolné možné poškození dat na disku. Toto je proto nutné řešit kontrolou disku příslušným programem po každém takovém selhání systému. (V MS-DOSu a Windows 95 je k tomu příkaz `scandisk`.)

Nevýhodou FAT tabulky je také pomalejší přístup k souborům, protože systém musí kromě vlastních souborů průběžné načítat i různé části FAT do paměti. Toto zpomalení může být dosti znatelné, proto je FAT jedním z hlavních kandidátů na cachování. Cachování FAT tabulky obvykle probíhá podobně jako u adresářů: Při prvním načtení určité části tato zůstává v cache pro další použití.

13.6.3 Indexovaná alokace

Další alternativou je vedení seznamu použitých klastrů pro každý soubor a jejich uložení k podobě posloupnosti jejich čísel (indexů) na začátku souboru (v jeho prvním klastru). Indexová tabulka je tedy jakousi „malou FAT“ u každého souboru. Toto řešení tedy má podobné vlastnosti jako při použití FAT tabulky, ale přístup k disku je rychlejší, protože čísla použitých klastrů jsou na jednom místě. Nevýhodou je, že každý soubor potřebuje více prostoru na disku pro uložení tabulky indexů. Abychom umožnili vytváření velkých souborů, tabulka musí být hodně velká, což je pak ale zvlášť neefektivní u malých souborů. Tento problém lze řešit buď víceúrovňovými odkazy (jako u stránkovacích tabulek, viz kap. 9.3 na straně 89). Výhodnou alternativou může být kompromisní schéma, kde indexová tabulka má jen několik málo záznamů, kde prvních několik z nich jsou přímé odkazy na klastry a další ukazují nepřímo na klastry obsahující pokračování indexové tabulky. Tento způsob je použitý např. v systému UFS.

13.6.4 Srovnání

Nelze říci, který alokační model je obecně nejlepší, protože různé způsoby (strategie) využití disku mohou vést k volbě jiného alokačního modelu. Například aplikace vyžadující především sekvenční čtení jistě budou nejlépe fungovat s obyčejnou souvislou alokací, protože čtení je zde nejrychlejší. Tento způsob je také de facto přenesení způsobu, jakým fungují magnetické pásky, na disky. Proto software určené původně pro pásková zařízení bude se souvislou alokací místa na disku fungovat stejně dobře.

Spojové seznamy řeší řadu problémů souvislé alokace, zvláště fragmentaci, ale mají problémy s přímým přístupem. Některé systémy proto pro soubory přímého přístupu používají jiný alokační model než pro klasické sekvenční soubory. Soubory určené pro sekvenční čtení pak používají spojový seznam, zatímco soubory určené pro sekvenční i přímý přístup používají souvislou alokaci a musejí mít předem pevně danou velikost.

Indexová alokace je také pomalá, zvláště u víceúrovňových tabulek, kde pro načtení indexu prvního klastru může být potřeba načíst několik jiných klastrů umístěných třeba i v různých částech disku.

Všechny druhy spojových seznamů i indexových tabulek je možno uchovávat v cache, ovšem jen pokud je k dispozici dostatečná paměť. V případě, že cache je použita, práce s diskem může být srovnatelně rychlá (oproti souvislé alokaci), bez cache je zpomalení (zvláště u pomalých médií jako je např. disketa) dosti znatelné.

Obecně může rychlosti prospět kromě správné volby alokačního algoritmu také jeho dodatečná úprava. Např. upřednostnění alokace více sousedících klastrů najednou obvykle pomáhá, neboť umožní větší (delší) nepřerušené DMA přenosy z/na disk. U víceúlohových systémů může takový postup také znamenat snížení fragmentace souborů (což u diskety opět vede ke značnému zrychlení čtení i zápisu). Jelikož CPU je dnes o několik řádů rychlejší než disk, i velmi složitý kód může být přínosný, pokud povede k ušetření aspoň několika přístupů na disk.

13.7 Evidence volného místa

Problém evidence volného místa na disku jsme zmínili již v předchozí sekci v souvislosti s jeho alokací. Evidence volného místa je potřeba v okamžiku, kdy zakládáme soubor či zapisujeme nová data. Bez ohledu na použitý alokační algoritmus, systém musí vědět, odkud brát další klastry pro zapisovaný soubor.

Základní metodou evidence volného místa je bitová mapa zmíněná v předchozí sekci. Nevýhodou může být velikost této tabulky. Poměrná velikost k celému disku je sice malá (1 bit na klastr, čili minimálně 1:4096 a v praxi i více), zdlouhavé může být procházení této tabulky při hledání onoho jednoho volného klastru. Např. u 400GB disku je třeba projít až 100MB a hledat jedničku či nulu (viz pověstné hledání jehly v kupce sena). Procházení je samozřejmě v průměru tím pomalejší, čím plnější je disk. Kromě samotného procházení je však především třeba oněch 100MB načíst do paměti a zde je časová náročnost již značná.

Alternativou může být použití spojového seznamu, obdobně jako u alokace místa pro soubory. Tentokrát tedy spojový seznam eviduje všechno volné místo, jakoby patřilo do speciálního souboru. Klasická implementace je zde velmi nevýhodná, protože zjištění volného místa můžeme provádět jedině tak, že volné klastry načítáme (zbytečně) do paměti. Vhodnější může být evidence většího počtu volných klastrů v prvním z nich pomocí indexové tabulky, kde poslední odkaz vede na klastr s pokračováním této indexové tabulky. Práce s tímto volným místem nyní bude mnohem rychlejší.

Vhodným doplňujícím mechanizmem je namísto evidence každého jednotlivého klastru evidovat jejich souvislé bloky. V tom případě pro každý souvislý blok volných klastrů evidujeme jeho počátek a délku. Při vysoké fragmentaci disku je tento přístup dvakrát náročnější (potřebujeme dvě hodnoty pro každý volný klastr), ale v praxi díky němu naopak místo obvykle ušetříme.

13.8 Efektivita a výkon diskových operací

Disky jsou dnes nejpomalejší součástí počítače, zefektivnění jejich činnosti je tedy velmi žádoucí. Obecně lze říci, že moderní operační systémy dosahují nejlepších výsledků tehdy, když minimalizují počet přístupů na disk a když data na něm organizují tak, aby se v každém krátkém časovém úseku přistupovalo pokud možno jen do nějaké dostatečně malé oblasti disku (princip lokality).

Obvykle bývá vhodné do jisté míry obětovat efektivitu využití prostoru na úkor rychlosti. Proto se běžně používají větší klastry, kdy dochází z vnitřní fragmentaci diskového prostoru, ale zrychlují se všechny diskové operace díky snížení vnější fragmentace.

Systémy nabízející vyšší funkcionalitu ji často nabízejí právě na úkor rychlosti. Např. NTFS eviduje nejen běžný čas poslední změny souboru, ale i čas posledního čtení. Při každém přístupu k souboru tedy systém musí načíst blok popisující soubor, změnit v něm datum a opět jej uložit. Počet diskových operací se tím samozřejmě zvyšuje.

U velkých disků (což je relativní pojem, takže se to týká vlastně všech disků) je otázkou také velikost datových položek pro číslování klastrů. Jsou-li malé, pak je maximální velikost disku limitována. Jsou-li však příliš velké, je mnoho diskové kapacity vyplýtváno na tyto hodnoty. Například u běžné 1.44MB diskety není vhodné používat stejný systém jako u 1TB pevného disku, protože bychom tím skutečně využitelnou kapacitu diskety zbytečně zmenšili.

Prvkem ovlivňujícím rychlost práce s diskem je především cache. MS-DOS implementuje diskovou cache jako samostatný program (`smartdrv`), který si pro sebe alokuje jisté množství paměti, ve které udržuje posledně používané klastry disku. Vzhledem k tomu, že samotný MS-DOS využije v zásadě jen 1MB, zbývá obvykle dost paměti pro potřeby cache. Moderní systémy (NT, Linux, Solaris aj.) však pracují s operační pamětí efektivněji: Používají jednotnou cache pro paměťové stránky i soubory a soubory cachují na úrovni souborových dat (vyšší abstrakce), namísto klastrů. Správa paměti a diskové operace tedy jsou implementovány dohromady v jedné části systému. Toto komplikované řešení vyžaduje řadu detailních nastavení, aby nedocházelo k tomu, že při náročnějších diskových operacích se odswapují procesy a paměť zbytečně zaplní diskové soubory, nebo obráceně.

Průvodce studiem

Zajímavou anomálii můžeme objevit při porovnání MS-DOSu a Windows NT. Windows NT má svou jednotnou cache přímo v jádru systému, zatímco ovladač disku je samostatný modul. MS-DOS naopak implementuje ovladač disku přímo v jádru systému, ale diskovou cache musíme doplnit pomocí dodatečného programu. Řešení MS-DOSu je de facto přesným opakem optimálního stavu, proto lze s určitou nadsázkou hovořit o anomálii.

Cachovací algoritmy jsou u disků stejné jako u operační paměti s tím, že při práci s diskem máme úplnou kontrolu nad tím, kdy se ke kterému bloku přistupuje, takže je možno implementovat i např. algoritmus LRU (least recently used), který u operační paměti efektivně implementovat nelze. U disků je přitom implementace jednoduchá: Stačí evidovat seznam bloků seřazený tak, že na konci je vždy posledně přistupovaný. Tento seznam udržujeme v pevné velikosti, takže při přidání nového bloku vždy nejstarší vyhodíme. Podobně lze bez problému implementovat a použít i LFU či MFU. Jelikož provozní režie diskové cache je velmi malá ve srovnání s možnou úsporou času přístupu k disku, osvědčují se i sofistikovanější algoritmy, například kombinace LFU a LRU, kdy čerstvě načtené stránky (ty mají nejméně přístupů) jsou organizovány pomocí LRU a dříve načtené stránky jsou organizovány pomocí LFU. Při zařazení nové stránky se tato zařadí vždy do první sekce, nejstarší stránka z ní se zařadí do druhé sekce a z ní se vyřadí nejméně používaná stránka. (Samozřejmě existuje více možností, jak LRU a LFU kombinovat.)

Použití jednotné správy virtuální paměti společně se systémem mapování souborů do paměti vede paradoxně k velmi jednoduchému způsobu práce se soubory: Při otevření souboru se provede jeho mapování do paměti a všechny stránky se označí jako alokované a odswapované (a „nečekaně“: namísto swapovacího souboru jsou odswapovány právě do onoho mapovaného souboru). Při práci s jednotlivými částmi souboru pak dochází k jeho načtení přes demand paging. Takto se vyřizuje jak čtení, tak zápis a při zavření souboru stačí jen odswapovat stránky, které mají příznak změny.

Průvodce studiem

Prvním masově rozšířeným systémem s jednotnou správou virtuální paměti byl NT. Jak uvádí Silberschatz [SGG05], Solaris například veškerou práci s diskem interně takto převádí na mapování souboru a tím jej zahrne do svého jednotného cachovacího systému.

U knihoven vyšších jazyků, které implementují vlastní diskové operace (např. u jazyka C), obvykle najdeme vlastní cachování. Soubor se pak cachuje vlastně (minimálně) dvakrát: Poprvé správcem paměti v operačním systému při fyzickém čtení z disku a podruhé

v knihovně vyššího jazyka. Díky ostrému nepoměru rychlosti CPU a disků se toto v praxi obvykle příliš neprojeví. Ve speciálních případech však může být dvojí cachování zdrojem zpomalení, ale také zrychlení, právě podle konkrétní situace.

Dalším prvkem zrychlujícím práci s diskem je asynchronní zápis. Toto souvisí s diskovou cache: Zapisovaná data jdou jen do cache a na disk se zapisují až později. Tento způsob používání cache se nazývá *write–back cache*, zatímco přímý zápis s uchováním kopie v paměti je *write through cache*. (Druhý jmenovaný způsob byl používán zejména v MS-DOSu.) NT například nabízí funkce pro explicitní asynchronní zápis, kdy je proces informován v okamžiku skutečného fyzického zapsání dat. To však v praxi nepřináší žádný zvláštní benefit, běžné systémy pro tento účel navíc mají i zvláštní funkci (`flush` – vyprázdni buffery). NT však stejnou funkcionalitu nabízí i pro čtení, tj. umožňuje zahájit asynchronní čtení s tím, že proces je informován o jeho dokončení. Asynchronní čtení disku bylo po dlouhá léta výsadou NT, v poslední verzi Linuxu (kernel 2.6) ji však již najdeme také a stejně tak se postupně dostává do různých klonů Unixu. (Linux a Unix v tomto směru však nejsou kompatibilní, aspoň zatím.)

Průvodce studiem

Ačkoliv pojmy blokující/neblokující a synchronní/asynchronní jsou ve skutečnosti nezávislé, v praxi používané metody jsou obvykle buď synchronní a blokující, nebo asynchronní a neblokující. Proto tyto pojmy do určité míry splývají.

Sekvenčním čtení souboru je možno dále optimalizovat. (Základem je, aby uživatelský proces systému oznámil, že bude číst pouze sekvenčně.) Obecně používaný algoritmus výběru oběti LRU je při sekvenčním čtení vhodné nahradit jiným, protože posledně používaná stránka se po přesunu na další již používat nebude, ale v LRU bude mít nejvyšší prioritu a nejspíše ještě dlouho zůstane v paměti. Princip *free–behind* likviduje z paměti přečtené stránky souboru okamžitě, jakmile se načítá stránka další. Princip *read–ahead* naopak při čtení načítá více stránek souboru najednou, protože ví, že při sekvenčním čtení by se stejně později načetly a načtením většího bloku najednou, jak víme, se práce s diskem zrychlí.

Průvodce studiem

Disk CD-ROM (a podobné) ukládá data do spirály a disk se v mechanice neustále otáčí. Při čtení se čtecí hlava postupně posouvá po spirále. Při přerušení čtení ztratíme pozici ve spirále a při dalším pokračování ve čtení se tato musí poměrně obtížně hledat. Proto se u CD a DVD zvlášť vyplatí načítat i velké bloky dat dopředu. Na druhou stranu je třeba dodat, že tato vlastnost je samozřejmě všeobecně známa, takže i samotná CD/DVD mechanika se snaží načítat data dopředu (read–ahead) a uchovává si je ve svém bufferu. Pokud přijde požadavek na přečtení dalšího bloku dostatečně brzy, mechanika předá již načtená data ze svého bufferu a čtení disku tedy pokračuje nepřerušeně dál. Při delším zdržení CPU je pak však další penalizace kvůli zmíněnému nutnému hledání pozice posledního čtení. Čtení CD je tedy díky read–ahead asynchronní, i když jsou čtecí funkce blokující. (Stejný princip se používá i na pevném disku, jen s menším rychlostním ziskem.)

Použití write–back cache přináší i možnost optimalizace postupu zápisu. Ovladač disku totiž namísto postupného zapisování bloků dat z cache může změnit pořadí požadavků tak, aby posuny hlavy byly co nejkratší. Existuje přitom několik strategií, představíme si je podrobněji v kapitole 13.11 na straně 151.

13.9 Chyby a zotavení z nich

Chyby související se souborovým systémem jsou nejčastěji důsledkem selhání systému v okamžiku, kdy data na disku nejsou v konzistentním stavu. V této sekci se tedy nebudeme příliš zabývat hardwarovými selháními disků a jiných datových médií, ale zaměříme se právě na problematiku konzistence.

13.9.1 Konzistence

Jak víme, řada často používaných informací je dlouhodobě udržována v cache za účelem zrychlení diskových operací. Víme, také, že změny v těchto datech nejsou na disk zapisovány ihned, ale obvykle až později a postupně dle vytížení systému (write–back princip). Dojde-li k selhání systému, čili k havárii počítače (softwarové či hardwarové), pak data v cache, která nebyla zapsána na disk, jsou ztracena. S ohledem na to, že disky a cache jsou dnes již poměrně velké, běžně se stává, že při takovém selhání jsou některá spolu související data již zapsána na disk, zatímco jiná ne.

Průvodce studiem

Místo pojmu „selhání systému“ by možná bylo přesnější hovořit o „havárii“, neboť nás zajímají jen ty selhání, která vedou k předčasnému ukončení činnosti či úplného restartu operačního systému a počítače. Dohodněme se však, že v této kapitole budeme tyto stavy nazývat právě selháním systému.

Uveďme si jednoduchý příklad nekonzistence disku: Při zápisu souboru se systémem FAT se nejprve uloží data souboru, pak se v adresáři vyznačí nová délka souboru a na závěr se ve FAT aktualizuje řetěz klastrů náležejících k souboru. Při selhání systému před aktualizací FAT dojde k tomu, že v adresáři už je uvedena nová délka souboru, ale řetěz ve FAT neexistuje. Použije-li se opačné pořadí operací, tj. nejprve se aktualizuje FAT, potom adresář a na konec soubor, pak opět při selhání systému po prvním kroku je stav disku nekonzistentní. Speciálně u systému používajících FAT tabulku nebo jiné typy tabulek zasahujících do více klastrů je pak nejvážnějším problémem možné selhání během aktualizace této tabulky. Tabulka je pak z části nová a z části stará, což často vede až k rozsáhlým ztrátám dat na disku. V praxi je navíc výhodné FAT tabulku zapisovat až na konci, protože např. při zápisu 1000 souborů je zbytečné po každém jednotlivém zápisu zdlouhavě aktualizovat FAT tabulku; stačí na závěr zapsat nový stav po všech změnách. Unix naopak ukládá informace o souboru vždy na jedno místo a to před zahájením zápisu. Při selhání tedy je záznam v adresáři většinou již nový a korektní, ale data souboru mohou chybět.

Jelikož selhání systému může být časté a nekonzistentní stav disku může vést k rozsáhlým chybám v budoucnu, včetně celkové ztráty všech dat, operační systémy mají pro tyto případy kontrolní programy (`fsck` (file system check) v Unixu/Linuxu, `chkdsk` (checkdisk) ve Windows). Poškození konzistence, jak bylo vidět na příkladu výše, se nejčastěji týká evidence obsazených a volných klastrů a adresářů. Pokud jsou všechny tyto údaje v pořádku, ale není správně zapsán obsah souboru, pak je to sice také chyba, ale neovlivňuje konzistenci svazku jako celku.

Systémy obvykle dokáží poznat, kdy je třeba spouštět kontrolní programy konzistence disku. Při správném vypnutí systému se na každý svazek uloží značka, že je v konzistentním stavu. Při předčasném vypnutí tedy tato značka chybí, takže při příštím startu systém bezpečně pozná, které svazky je nutno zkontrolovat. Nalezne-li kontrolní program chyby, pokusí se je opravit. Možnosti oprav jsou samozřejmě závislé na konkrétní implementaci alokace a struktuře svazku. Výsledky se mohou lišit – od kompletní opravy všech problémů, až po ztrátu dat. (Ztráta vzniká

například tehdy, když za účelem znovuzískání konzistence svazku jsou zrušeny vadné soubory, u kterých se neví, zda jsou jejich data v pořádku, či nikoliv.)

Průvodce studiem

Možnosti oprav konzistence jsou závislé na konkrétním formátu poškozeného svazku. Otázky na toto téma se mohou objevit na zkoušce, protože správnou odpovědí student potvrdí, že správně chápe strukturu daného souborového systému.

13.9.2 Zálohy

Jedním z důležitých podpůrných mechanizmů ochrany před selháním je zálohování dat. Zálohování lze provádět i ručně, např. překopírováním důležitých dat na diskety či CD a jejich uložení na bezpečném místě. Zálohy není vhodné ukládat na místě serveru, protože např. při požáru o ně přijdeme spolu se serverem. U větších systémů, třeba zmíněných serverů, je však místo ručního zálohování vhodnější nasadit automatický zálohovací systém. Těch existuje celá řada a jejich podrobnější studium je nad rámec tohoto kurzu.

Zálohování velkého množství dat je obvykle prováděno na magnetické pásky, v poslední době ale čím dále častěji i na záložní pevné disky. Pásky jsou obvykle levnější, mají vyšší kapacitu, ale pracuje se s nimi hůře. Klesání cen disků a růst jejich kapacity však postupně pásková zařízení vytlačuje z trhu.

Pro organizaci zálohování lze zvolit různé postupy. Obvykle jednou za čas provedeme kompletní zálohu, zatímco v mezilehlém období vždy zálohujeme jen změny od poslední zálohy. Děláme-li zálohy pravidelně, pak změněné soubory poznáme podle jejich data (datum poslední změny souboru je uložen na disku) nebo také podle příznaku zálohování, který většina operačních systémů podporuje. Každému souboru je tento příznak nastaven při zápisu dat a vynulován při vytvoření zálohy. (Je to souborová obdoba dirty bitu známého ze správy paměti.)

13.10 Žurnálové systémy

Žurnálování zajišťuje konzistenci disku.

Moderní souborové systémy obvykle věnují problematice konzistence větší pozornost a implementují tzv. log či žurnál (dva názvy téhož). Tento princip je odvozen od relačních databází, kde se začal používat za stejným účelem – aby při selhání systému neskončila databáze v nekonzistentním stavu.

Log je de facto zápis o všech operacích, které se dějí na svazku. Můžeme si jej představit jako běžný textový soubor, kde si ručně zapisujeme, jaké části disku se postupně mění a jak. Ve skutečnosti však log zapisuje sám systém a jeho podoba je pro něj více čitelná (než textové soubory blízké člověku). Do logu se zapisují obvykle jen metadata, tj. „data o datech“, nikoliv však samotná data souborů. Metadata svazku jsou tedy všechny informace zapsané na disku kromě vlastních těl souborů.

Princip práce s logem je poměrně jednoduchý a popíšeme si jej na příkladu systému NTFS, kde se logování jako první masově rozšířilo: Log je soubor na disku a zapisují se do něj všechny plánované operace tak, aby z toho bylo možno tyto operace provést nebo zrušit. Vlastní data (těla) souborů se do logu nezapisují. Do procesu zápisu na disk se také výrazně projevuje použití cache, takže každá změna disku je provedena ve čtyřech krocích:

1. Změna je zapsána do logu. Log ve skutečnosti zůstává jen v cache.
2. Změna je provedena. Opět ve skutečnosti vše zůstává v cache.
3. Log je zapsán z cache na disk. (write–back)

4. Změněná cache je zapsána na disk. (write–back)

Jakmile jsou změny skutečně na disku, log je vynulován (log nikdy neobsahuje již provedené změny). Jde-li o operace při kterých se zapisují i (nelogovaná) data, pak data souboru mohou být z cache zapsána fyzicky na disk až po zapsání logu. V případě selhání se pak podle logu mohou jednotlivé neprovedené operace provést či zrušit.

Nevýhodou použití logu je zpomalení kvůli nutnosti zapisovat vše dvakrát. Výhodou je však to, že log je na jednom místě a zápis do něj je rychlejší než přímý zápis změn na disk. Přitom samotné změny lze zapsat až později, jakmile bude počítač méně vytížen. V praxi tedy žurnálové souborové systémy nejsou dvakrát pomalejší, jak by se teoreticky mohlo zdát.

13.11 Plánování přístupu k disku

V závěru kapitoly ještě navážeme na téma efektivity diskových operací probrané v sekci 13.8 na straně 146. Řada systémů dnes používá write back cache a s tím spojený opožděný zápis na disk. Opožděný zápis přitom lze použít nejen v souvislosti s cache, ale i jako samostatně použitelný mechanizmus, jak optimalizovat práci s diskem. V obou případech můžeme ušetřit přesuny raménka (seek) tím, že přeuspořádáme požadavky diskových operací do takového pořadí, aby mezi sousedními operacemi byly přesuny co nejkratší.

Průvodce studiem

Problematiku strategie zápisu na disk je třeba vnímat v souvislosti s víceuživatelským prostředím. Je běžné, že na počítači v danou chvíli pracuje více než jeden uživatel a na pozadí ještě často probíhají nějaké systémové operace (defragmentace disku, antivirová kontrola, atp.). V tom případě disk dostává téměř souběžně rozdílné požadavky a obvykle existuje řada variant, v jakém pořadí je plnit, které se mohou dosti lišit co do časové efektivity.

Tuto optimalizaci může však dělat nejen operační systém, ale často již sám disk má jistou inteligenci a vestavěnou cache a dokáže požadavky zpracovávat efektivněji než jen klasicky v pořadí, v jakém je dostává.

Disk pracuje asynchronně.

Běžný disk v současných počítačích pracuje asynchronně, tj. během jeho práce se procesor počítače může zabývat jinou činností. V době, kdy disk právě nevykonává žádné operace, může nově příchozí požadavek okamžitě začít vykonávat. V opačném případě jsou všechny příchozí požadavky na diskové operace zařazovány do fronty. Po dokončení aktuální operace vybere systém další operaci z fronty čekajících a začne ji vykonávat. Operační systém může použitím vhodné strategie výběru pořadí diskových operací zrychlit celkovou práci s diskem. Tato strategie je dána použitým algoritmem výběru požadavků z fronty. Jedná se tedy o jistou obdobu algoritmů pro výběr procesu k běhu či stránky k odswapování, protože se opět vybírá jedna položka z mnoha prvků seznamu, ale v případě disku se používají jiné algoritmy než u plánování procesů či stránkování paměti.

13.11.1 Algoritmus FCFS (first come, first served)

Algoritmus FCFS je nejjednodušší algoritmus, který jednoduše zpracovává požadavky v pořadí, v jakém přicházejí. Tento algoritmus je vždy férový, ale pochopitelně je také neefektivní.

13.11.2 Algoritmus SSTF (shortest seek time first)

Algoritmus SSTF vybírá požadavek, jehož seek time je nejkratší. Seek time je čas potřebný pro přesun raménka z aktuální pozice na pozici, kde bude vykonán požadavek. Tento algoritmus

tedy upřednostňuje požadavky, které lze nejrychleji vyřídit. Není férový, protože ve frontě mohou zůstávat a stárnout požadavky na vzdálených místech disku, na které nikdy nepřijde řada. Při běžné víceúlohové práci s diskem je takové stárnutí požadavků u tohoto algoritmu dokonce běžné.

13.11.3 Algoritmus SCAN

Algoritmus SCAN (tzv. „výtah“) systematicky přesunuje raménko mezi oběma konci disku tam a zpět a cestou vykonává požadavky na aktuální pozici.

Nevýhodou tohoto algoritmu je, že požadavky ve střední části disku se zpracovávají zhruba dvakrát rychleji než požadavky v okrajových částech disku. Při rovnoměrném rozložení požadavků na disku totiž raménka v okamžiku dojetí na jeden okraj disku bude opět zpracovávat nejprve požadavky ve stejné části disku, zatímco nejstarší požadavky, které jsou na opačné straně disku, zpracuje až na konci zpětného chodu.

13.11.4 Algoritmus C–SCAN (Circular SCAN)

Algoritmus C–SCAN je variantou předešlého, která se snaží řešit výše zmíněný nedostatek. C–SCAN také posouvá raménko po disku od kraje ke kraji, ale požadavky zpracovává vždy jen při cestě jedním směrem. Tím je odstraněno zvýhodňování střední části disku oproti okrajům.

Průvodce studiem

Název Circular SCAN je odvozen od představy, že stopy na disku jsou v kruhu a raménko se pohybuje jedním směrem po tomto kruhu. Za poslední stopou je tedy jakoby opět stopa první, což je pak totéž jako přesun raménka zpět na začátek disku bez realizace čekajících operací.

13.11.5 Algoritmy LOOK a C–LOOK

LOOK a C–LOOK jsou úpravou předešlých dvou, kdy raménko necestuje až ke kraji disku, pokud tam není čekající operace na vyřízení. Tím se tedy jednotlivé cesty zkrátí a vyřizování požadavků se zrychlí.

Průvodce studiem

Název „look“ je odvozen od principu, že systém se nejprve „podívá“ (anglicky look), na kterém místě disku je nejkrajnější požadavek, než tam přesune hlavu.

13.11.6 Výběr vhodného algoritmu

Jak už víme, možných algoritmů plánování přístupu na disk je mnoho, je tedy otázkou, který z nich je v praxi nejvhodnější. Silberschatz [SGG05] uvádí, že nejběžnější je SSFT, který je rychlejší než FCFS, zatímco SCAN varianty se hodí pro systémy s velkým vytížením disku. V praxi naštěstí toto plánování přebírá také samotný disk, který obsahuje vlastní inteligenci a poměrně velkou cache. Výkon disku je ovlivněn také způsobem využití počítače. Budou-li v jednom okamžiku dva procesy sekvenčně číst velké soubory, každý z nich bude generovat souvislou řadu požadavků na čtení, obvykle po sobě jdoucích sektorů. Z hlediska celkového a dlouhodobého výkonu systému je pak nejvhodnější nejprve načíst celý jeden soubor, a pak

teprve přejít na druhý (SSTF), ovšem při spolupráci více uživatelů to povede k velmi nevyváženému chování systému. Naopak střídavé vyřizování požadavků jednotlivých uživatelů (varianty SCAN a LOOK) zajistí rovnoměrný běh všech procesů, i když celkový výkon systému bude zjevně velmi špatný.

Jak už bylo řešeno v předchozím textu, výkon diskových operací negativně ovlivňuje také nutnost přistupovat k adresářům (minimálně při otevírání souboru je to nezbytné). Hodně tedy záleží nejen na algoritmu přístupu k disku, ale také na samotné organizaci svazku (tj. kde jsou adresáře) a cache (tj. zda se adresáře cachují přednostně).

Z hlediska efektivity práce je také zajímavý jistý „souboj“ mezi operačním systémem a samotným diskem. Moderní pevné disky totiž jsou schopny organizovat požadavky tak, aby byl celkový výkon co nejvyšší; operační systém na druhé straně ví, že některé diskové požadavky jsou důležitější než jiné. Například výpadky stránek při stránkování je třeba vyřizovat dříve než zápis dat z diskové cache. Podobně čtení disku má přednost před zápisem, pokud je však cache téměř plná, pak zápis musí mít přednost před čtením. Podobně při žurnálování diskových operací je přímo nezbytné pevně určovat pořadí zápisů, aby byla konzistence svazku zaručena. (Pokud aktualizace řídicích dat na disku předběhne zápis žurnálu, může při výpadku systému dojít k narušení konzistence. Viz kap. 14.4 na straně 161.) Situace, kdy operační systém přebírá kontrolu nad časovou organizací diskových požadavků, bývá neformálně označována jako „spoon–feeding“ (krmení lžičkou – sousto po soustu).

Shrnutí

V této kapitole jsme se podívali z praktického a více technického hlediska na práci s diskem jakožto základním médiem pro souborový systém. V úvodu kapitoly jsme se seznámili se strukturou disku a rozdělením na svazky. Dále jsme se věnovali implementačním otázkám: Probrali jsme implementaci adresářů, evidenci volného místa, otázky efektivity diskových operací a algoritmy plánování přístupu na disk, algoritmy alokace místa na disku. Věnovali jsme se také problematice chyb, jejich detekci a předcházení jim pomocí žurnálování.
V následující kapitole budeme pokračovat ještě více prakticky a představíme si některé konkrétní souborové systémy.

Pojmy k zapamatování

- plotna
- stopa
- cylindr
- MBR (master boot record)
- FCB (file control block)
- write through cache
- write back cache
- free behind
- read ahead
- žurnálování (neboli logování)
- plánování přístupu k disku
- FCFS (first come first served)
- SSTF (shortest seek time first)
- SCAN
- C–SCAN (circular scan)
- LOOK
- C–LOOK (circular look)
- spoon–feeding („krmení lžičkou“)

Kontrolní otázky

1. *Vysvětlete význam master boot recordu na disku.*
2. *Jakým způsobem operační systémy řeší to, že v systému je obvykle zapojeno více než jedno zařízení se souborovým systémem? Je běžně používané řešení optimální, nebo byste preferovali spíše jiné? Proveďte diskuzi.*
3. *Popište výhody a nevýhody cache typu write through a write back.*
4. *Při implementaci adresářů se někdy používají hashovací tabulky. Jaký je jejich význam? Je hashovací tabulka struktura základní, nebo jen doplňková? Vysvětlete proč.*
5. *Diskový prostor lze alokovat různými způsoby. Uveďte různé možnosti, jak v systému evidovat prostor přidělený souborům, a proveďte diskuzi.*
6. *Jak se eviduje volné místo na disku? Je taková evidence důležitá i u systémů, které evidují přidělené místo, takže volné je tam pochopitelně všechno ostatní (nepřidělené souborům)? Proveďte diskuzi.*
7. *Jaké cachovací algoritmy jsou vhodné pro cachování diskových operací? Je cache systém součástí modulu souborového systému, nebo jiné části operačního systému? Vysvětlete, proč tomu tak je a jaké výhody či nevýhody mají jednotlivá technicky možná řešení.*
8. *Popište princip žurnálování diskových operací. Některé systémy žurnálují jen metadata, jiné pak všechny operace. Diskutujte výhody a nevýhody obou těchto přístupů.*
9. *Popište základní algoritmy plánování přístupu k disku. Který algoritmus je obecně nejvhodnější? Je lepší, aby toto řešil operační systém, nebo spíše přímo elektronika v pevném disku?*

14 Příklady souborových systémů, RAID

Studijní cíle: V této kapitole si představíme několik souborových systémů z praxe. Jmenovitě to budou systémy MS-DOSu (FAT), Unixu (UFS) a Windows NT (NTFS). Cílem každé sekce je popsat strukturu daného souborového systému a také diskutovat jeho výhody či nevýhody. Na konci kapitoly se podíváme na systémy diskových polí RAID.

Klíčová slova: FAT, UFS, NTFS, RAID

Potřebný čas: 160 minut.

14.1 Úvod

Jednotlivé souborové systémy byly obvykle navrženy také speciálně pro určitá datová média, jako hard disky (česky pevné disky) nebo CD-ROM; některé z nich přitom měly to štěstí, že z historických důvodů se používaly či dodnes používají na různých systémech či médiích. Např. systém FAT, který původně používal MS-DOS na disketách, se později používal také na pevných discích a dodnes je to nejrozšířenější souborový systém výměnných médií, jako jsou např. USB disky.

Kromě systému FAT, který si představíme podrobněji, se dále v této kapitole podíváme také na Unix File System (UFS) a NT File System (NTFS). Zmíníme také Extended File System (ext) používaný na Linuxu. Ve všech případech (včetně FAT) jde o souborové systémy existující v mnoha verzích, které se v různých případech různě liší, od malých detailů až po zásadní systémové rozdíly.

14.2 FAT

V následujících sekcích si představíme několik konkrétních souborových systémů. První na řadě je systém FAT, který původně sloužil v systému MS-DOS, kde se používal pro diskety i pevné disky. Jeho varianty se používaly jako primární souborový systém až do Windows Me (ovšem ne ve Windows NT) a rozšířil se jako de facto standardní formát disket. Varianty FAT12 a FAT16 bez dlouhých jmen (bude vysvětleno níže) jsou dnes standardem i de jure a podporují je prakticky všechny operační systémy pracující s těmito disketami. Stejný formát se dnes používá také pro USB disky. To se však v budoucnu může změnit, jak se jejich kapacity budou zvětšovat, neboť FAT12 a FAT16 nejsou pro velké svazky vhodné.

Průvodce studiem

Základem toho, aby systém mohl podporovat formát FAT, je mít stejný či kompatibilní hardware. Přitom nejde jen o samotné diskety, ale také o použitý řadič. Např. PC, Macintosh nebo taky Atari ST používají kompatibilní řadič a jejich operační systémy tedy mohou diskety navzájem číst. Naopak třeba Commodore Amiga používá stejné 3.5” diskety, ale její řadič není schopen magnetickou informaci z PC přečíst (platí i obráceně).

Samotná kompatibilita řadiče však nestačí ke čtení disket. Např. Sam Coupé má stejný řadič jako Atari ST, ale diskety formátuje na 800KB a používá vlastní zvláštní souborový systém. Naopak Atari ST používá přímo systém FAT, takže může vyměňovat diskety s PC počítači přímo, zatímco diskety ze Sam Coupé jdou číst je fyzicky po jednotlivých sektorech. Některé mechaniky na PC pak často mohou mít problémy s hustotou zápisu, když na disketě je místo obvyklých 720KB celých 800KB. (Je řeč o disketách s dvojitou hustotou MF-2DD 3.5”.)

Systém FAT je poměrně jednoduchý. Na začátku disku je boot sektor, za ním jsou dvě identické kopie FAT tabulky, potom kořenový adresář a za ním soubory. Adresáře (kromě kořenového) se

ukládají také jako soubory. FAT nepodporuje přístupová oprávnění (každý soubor je přístupný všem) ani jiný typ ochrany dat, nepoužívá žurnál a hlavní nevýhodou je omezení jmen souborů a adresářů na 8.3 (8 znaků jméno + 3 znaky přípona).

FAT12

Nejstarší verzí FAT je FAT12, která se objevila již v roce 1977 (tedy již 4 roky před uvedením MS-DOSu). Buňky FAT tabulky zde mají 12 bitů, čili každé tři bajty popisují vždy dva sousední klastry. Svazek tedy může obsahovat max. $2^{12} = 4096$ klastrů. Číslo uložené ve FAT tabulce vždy označuje číslo následujícího klastru v řetězu souboru. Volné klastry mají 0 a konec souboru je označen jako -1. Dalších několik hodnot je vyhrazeno pro označení vadných klastrů, takže skutečná maximální kapacita je o něco menší než oněch 4096 klastrů a závisí také na konkrétní implementaci FAT. Při maximální velikosti klastru 8KB je celková maximální velikost svazku (necelých) 32MB. I v současnosti se FAT12 stále používá jako primární formát pro svazky do 32MB.

Průvodce studiem

Limit 32MB platí pro souborový systém. V praxi ale může být limitujícím faktorem také implementace v konkrétním operačním systému. Některé indexy jsou navíc rezervované, takže skutečná maximální velikost je vždy o něco menší.

FAT tabulka nesoucí nejdůležitější informace o svazku je na jeho začátku. Konvencí dokonce je, že je celá na první stopě. Diskety s porušenou první stopou jsou tedy ve FAT systému nepoužitelné. Ačkoliv FAT je na disketě uložena dvakrát za sebou, MS-DOS druhou kopii nepoužívá, ani při fyzické chybě na první stopě disku.

Adresáře

Původní FAT systém nepodporoval podadresáře, všechny soubory byly umístěny ve společném adresáři umístěném na disku za FAT tabulkou. Záznam jednoho souboru zde má 32 bajtů. Stromový systém adresářů přibyl v systému MS-DOS 2.0 (1983). Zatímco kořenový adresář zůstal nezměněn (má pevnou velikost), další podadresáře je možno vytvářet a libovolně vnořovat do stromu, limitujícím faktorem je pouze délka cesty (počet znaků). Podadresáře jsou uloženy jako soubory, samy mohou obsahovat libovolný počet souborů a záznam o každém má opět 32 bajtů.

FAT16

MS-DOS 3.0 (1984) přinesl novou verzi FAT, kde položky FAT tabulky měly každá 16 bitů. Od té doby se původní FAT označuje jako FAT12 a tato druhá verze jako FAT16. Některé nové vlastnosti FAT16 bylo však možno používat až v pozdějších verzích systému, protože původně bylo hlavním limitujícím faktorem 16bitové číslování sektorů. (Bez ohledu na další formát svazku byla maximální velikost 32MB.)

Kromě nové velikosti FAT tabulky nepřináší FAT16 žádné novinky. Maximálně tedy lze použít svazky o necelých 2^{16} klastrech. Systémy většinou podporují klastry do 32KB a sektory číslují 32bitově, takže maximální velikost svazku je 2GB. (Některé verze Windows podporují i 64KB klastry a 4GB svazky.) Nutno dodat, že při 32KB klastrech dochází k velké fragmentaci (běžně je ztraceno i 10%–15% diskové kapacity).

Průvodce studiem

Pro zajímavost: Velikost klastru je na 32KB omezena tím, že počet sektorů v klastru je uložen jako znaménkové 8bitové číslo. Maximální mocnina dvojky tedy je 64, což při 512 bajtech na sektor dává celkově 32KB. Ačkoliv velikost sektoru může být také zvětšena (na 1024, 2048 atd.), v praxi se téměř výhradně používaly právě sektory 512bajtové.

Velikost kořenového adresáře se nastavuje při formátování, protože jde o pevně vyhrazený prostor za FAT tabulkou. Jeho velikost je uložena jako 16bitová znaménková hodnota, čili je možno nastavit max. přibližně 32 tisíc. Na disketách se však běžně používá jen 256 či 512 položek v adresáři (což i tak pro kořenový adresář zabere 8KB či 16KB prostoru diskety).

FAT32

Dosud poslední (a zřejmě skutečně poslední) verzí FAT je 32bitové rozšíření známé jako FAT32, které přišlo s druhou verzí Windows 95 (rok 1996). Položky FAT tabulky nyní mají 32 bitů, což umožňuje používat svazky větší než 2GB a především opět se vrátit k menším velikostem klastrů. 32bitová FAT tabulka teoreticky dokáže popsat velmi velké disky, v praxi však FAT32 naráží na řadu implementačních nedokonalostí či omezení. Běžně se používá jen 28 bitů, celkový počet sektorů na disku je taky ve 32bitovém čísle, program `scandisk` pro kontrolu disku ve Windows 95 pojme jen 22 bitů, program `fdisk` pro vytváření oddílů a svazků na disku je limitován na 64GB apod.

Microsoft přidal do NT podporu FAT32 od verze Windows 2000, ovšem později netradičním způsobem FAT32 „pohřbil“ tím, že nastavil limit velikosti svazku na 32GB. Důvodem není technické omezení, ale prostě svobodná volba Microsoftu s odkazem na to, že práce a údržba velkých FAT32 disků je velmi časově náročná. A skutečně, při selhání systému trvá kontrola velkého FAT32 disku velmi dlouho, nicméně Microsoft tím FAT32 efektivně pohřbil. Dalším problémem FAT32 je limit velikosti souboru na $2^{32} - 1$ bajtů (4GB), což je nedostatečné zejména při práci s videem.

FAT32 byl původně určen především pro (tehdy) velké pevné disky, s ústupem řady Windows 9x však i FAT32 ztratil svou pozici a dnes může být snad jen alternativou pro velké USB disky. (Je ke zvážení, zda USB disky formátovat do NTFS, či zůstat u FAT. Některé zdroje uvádějí, že žurnálování v NTFS snižuje životnost paměťových čipů typu flash používaných v USB discích.)

Dlouhé názvy (LFN)

Původní omezení názvů na 8.3 znaků padlo ve Windows 95, které přišlo s tzv. „dlouhými názvy“ neboli LFN (long file names). Dlouhé názvy na FAT svazcích podporovalo již dříve i Windows NT, ovšem jiným nekompatibilním způsobem, který se dnes již nepoužívá. LFN rozšíření se týká jen dat uložených v adresáři, je proto použitelné ve všech verzích FAT, pokud to systém podporuje. Pokud systém LFN nepodporuje, pak LFN záznamy přeskakuje díky nastavení atributu volume label (soubor totiž nemůže být jménem svazku), takže lze bez problému přenášet diskety mezi Windows a systémy bez podpory LFN. Dlouhé jméno může mít až 255 znaků v kódování UTF-16 (což je taky zásadní změna oproti pouze velkým anglickým písmenům v původní FAT). Jména jsou ukládána do buněk adresářové struktury, vždy 13 znaků LFN zabere jeden slot. (Tj. např. soubor se jménem o 30 znacích potřebuje 3 sloty v adresáři – jeden pro vlastní soubor a dva pro LFN.) Dodejme, že 8.3 názvy jsou vždy velkými písmeny a každý soubor má ve Windows kromě dlouhého názvu i 8.3 alternativu. Tyto alternativy vytváří systém automaticky, pokud je celý název souboru kompatibilní s 8.3 systémem, pak se LFN záznam nevytváří. (Windows 9x přitom za kompatibilní akceptuje i české znaky, zatímco Windows XP pro změnu kóduje malá/velká na zvláštním místě, aby nepotřeboval LFN a přitom zůstal plně kompatibilní se standardem.)

Zhodnocení FAT systému

Souborový systém FAT je více než 30 let starý a byl navržen pro svazky malých kapacit (řádově stovky KB na původních disketách) s ohledem na tehdejší převažující způsob práce s diskem a také se snahou vyhnout se složité implementaci, protože tehdejší počítače nebyl ani rychlé, ani neměly hodně paměti pro pomocné datové struktury. Z dnešního pohledu tedy může hodnocení FAT být značně deformované, nicméně právě z dnešního pohledu jde o systém s mnoha nedostatky a nevýhodami.

Názvy 8.3 mohou být doplněny o LFN, v praxi však prakticky jen systémy Windows toto podporují a ostatní systémy zůstávají jen u 8.3 názvů. Velikost svazku je značně omezena, na 2GB u FAT16 a 32GB u FAT32. Takto velké svazky však již mají velkou vnitřní fragmentaci (v případě FAT16) či velmi pomalé operace jako je kontrola disku při restartu nebo zjišťování volného místa na disku. (Zjistit volné místo lze jen načtením a prozkoumáním celé FAT tabulky. Ta je u FAT32 velmi velká.)

Systém nepodporuje uživatelská oprávnění. Rychlost je také poměrně malá, neboť klíčová data svazku jsou všechna soustředěna na jeho začátku a vynucují si tak neustálé přesuny raménka na začátek svazku. Ze stejného důvodu je FAT také nebezpečný v okamžiku fyzického porušení disku v místě začátku svazku – takový svazek je nepoužitelný. Zvláště u disket je podobná porucha celkem běžná (při častém používání, týká se i ZIP a jiných podobných magnetických médií). (Umístění FAT na jiné místo disku či její rozdělení na více částí by přineslo minimálně podstatné zrychlení a zřejmě i větší spolehlivost.)

FAT systém jsme si představili poměrně podrobně, ne však z důvodu jeho důležitosti, ale pouze pro ukázku, že i v takto jednoduchém souborovém systému lze najít velkou řadu detailů komplikujících situaci.

14.3 Unix File System (UFS)

UFS, jak už sám název napovídá, je souborový systém systému Unix, který se používá v různých variantách v jednotlivých verzích Unixu. V Linuxu se tento systém jako primární nikdy nepoužíval, i když aktuální verze Linuxu již dokáže s UFS svazky pracovat. Reálně používané varianty UFS obvykle obsahují různá navzájem nekompatibilní rozšíření, což je zřejmě především důsledek stáří systému či historického vývoje. V praxi tedy nelze příliš spoléhat na možnost sdílení svazků mezi jednotlivými Unixy, i když třeba možnost základního přečtení obsahu disku a svazku je díky stejným datovým strukturám poměrně pravděpodobná.

Stejně jako FAT, pracuje UFS se soubory a adresáři a nepodporuje žurnálování. Kromě těchto několika základních shodných bodů se však UFS a FAT velmi liší. UFS také používá jinou terminologii (což může být trochu matoucí pro čtenáře.) Svazky UFS pracují přímo s 512bjtovými sektory, či s většími klastry, pouze jsou zde nazývány *bloky*.

Struktura

Na začátku disku je bootovací blok, používaný jen při startu systému z daného svazku. Následuje *superblok*, což je hlavička obsahující informace o svazku jako celku. Každý soubor (včetně adresářů) je popsán strukturou zvanou *inode*, kterých je na disku pevný počet. V adresáři jsou jen jména souborů a odkazy na inode (zvané inumber). Více adresářových položek může odkazovat na tentýž inode (hard link), čili soubor může mít více jmen. Počet odkazů je uložen v inode a soubor se maže při odstranění posledního hard linku.

Různé varianty Unixu vnesly do UFS řadu úprav, kterými se snažily záplatovat nedostatky původního systému bez toho, aby vytvářely zcela nové struktury. Během let se tak objevila řada zajímavých úprav z nichž některé si nyní představíme.

Žurnálování

Žurnálování bylo do UFS přidáno jako nadstavba ještě v původním BSD Unixu, kdy dolní vrstva funguje klasickým způsobem a nad ní je další vrstva, která implementuje žurnálování pomocí standardních diskových operací. Toto řešení sice nedosahuje kvalit přímého žurnálování (viz NTFS níže), ale oproti původnímu UFS je to jistě pokrok. Rozdíl oproti NTFS, kde žurnálování je základem systému, v mnoha implementacích UFS se žurnálování nepoužívá vůbec, nebo bylo jako standardní funkce přijato až poměrně pozdě (např. Solaris v roce 2004 [McMa06], 11 let po Windows NT a NTFS).

Skupiny stop

BSD Unix upravil také strukturu svazku tak, že jej rozdělil do menších částí (skupin stop – cylinder group) a data z každého jednoho adresáře se snaží (pokud se vejdou) umisťovat vždy do stejné části. Při práci v nějakém adresáři pak nedochází k divokým přesunům raménka mezi vzdálenými stopami. (Tato změna přinesla velké zrychlení, veškerý zisk však samozřejmě končí v okamžiku, kdy na systému pracuje řada různých uživatelů současně, což je u serverových Unixů i docela běžné.) Každá skupina má vlastní inode objekty a také nese kopii superbloku. Rovněž evidence volného místa (ve formě seznamu volných bloků) je vedena v každé skupině zvlášť.

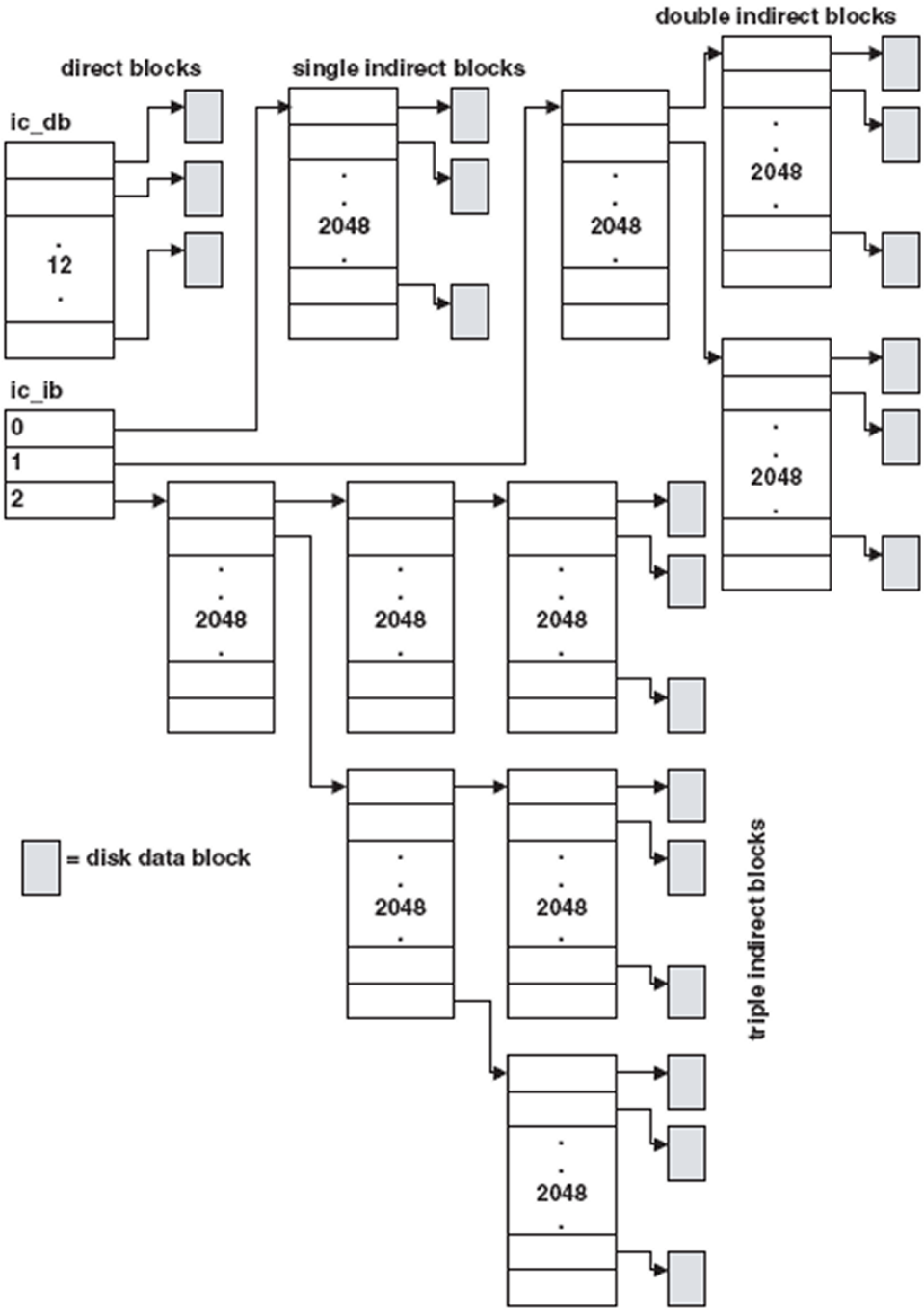
Adresáře

Adresáře jsou ukládány jako soubory. Adresář přitom nese jen jméno souboru a číslo inode, vše ostatní je v inode. (Zde je rozdíl oproti systémům Microsoftu, kde např. datum změny souboru a atributy jsou uloženy v adresáři.) Jména byla původně omezena na délku 14 znaků kvůli pevné délce záznamů v adresáři, později se však přešlo na záznamy proměnlivých délek a jméno může mít až 255 znaků. Kořenový adresář / má vždy inode č.2 a každý adresář obsahuje (pro snadnou navigaci ve stromu i bez cache) jako první dvě položky . s odkazem na sebe sama a .. s odkazem na nadřazený adresář. (Tyto odkazy . a .. má v adresářích i systém FAT.)

Těla souborů

Inode obsahuje také popis, kde na disku je tělo souboru. Systém nemá žádnou centrální FAT tabulku, každý soubor nese seznam bloků se souborem. Inode obsahuje odkazy na 12 bloků, dále jeden nepřímý odkaz na pokračování této tabulky, pak ještě jeden dvojitě nepřímý a jeden trojitě nepřímý odkaz. Nepřímé odkazy jsou použity jen u velkých souborů, které nelze přímými odkazy popsat. Každá nepřímá tabulka zabírá celý blok, takže maximální velikosti souborů závisí především na velikosti bloku. UFS obvykle používá 8KB bloky (16 sektorů, tedy 2–4 krát větší než u systémů Microsoftu).

Princip přímých a nepřímých odkazů na bloky je zobrazen na obrázku 22, zde pro velikost bloku 8KB a 32bitový odkaz na blok. Původní UFS používalo 16bitové odkazy, takže velikost svazku byla limitována na $2^{16} \times 8KB = 512MB$. Přechod na 32bitové odkazy zobrazené na obrázku 22 přinesl možnost používat větší svazky, ale vyžádal si změnu struktur inode, jmenovitě zvětšení ze 64 na 128 bajtů. Proto tyto dvě varianty UFS jsou zcela nekompatibilní. Maximální velikost souboru je tedy daná verzí UFS a (nelineárně) roste s velikostí bloku, je však také limitována údajem o délce souboru v inode, což je na řadě implementací 32bitové znaménkové číslo, takže soubor nemůže mít víc než 2GB. (Novější kvalitní implementace samozřejmě toto nepříjemné omezení již nemají.) Podobně datum je dle Unixových zvyklostí kódován tak, že nelze vyjádřit roky po 2031, a i toto omezení je v novějších variantách UFS odstraněno použitím větší proměnné.



Obrázek 22: Přímé a nepřímé odkazy na bloky v inode. [McMa06]

Bloky a fragmenty

Jak víme, používání velkých bloků s sebou nese vyšší rychlost, ale také vnitřní fragmentaci. Každý soubor zabírá na disku v průměru o polovinu velikosti bloku více místa, než je jeho délka. Toto ztracené místo je přitom vždy v posledním bloku souboru. UFS tyto neúplně využité bloky rozděluje na menší kousky (na polovinu, čtvrtinu či osminu), které pak může přidělit různým souborům. Důsledky fragmentace se tak výrazně zmenší.

Přesnější chování UFS se může lišit dle implementace. Solaris kouskuje velké soubory, tak že do první skupiny stop umístí jen 12 přímo odkázaných bloků a další pak dává do dalších skupin tak, že do každé přidá jednu celou tabulku odkazů (tj. 2048 odkazů krát 8KB dat = 16MB dat souboru v každé skupině stop). Skupiny jsou vybírány cyklicky, přeskakují se přitom ty, které již nemají potřebnou volnou kapacitu. Celý svazek je tedy zaplňován rovnoměrně, soubor

přitom začíná vždy ve skupině, kde je jeho adresář. Zajímavé také je, že velikost skupiny stop je limitována na 54MB, takže se tam vejdou 16MB části tří souborů. (Popsaný algoritmus zjevně má problémy v situaci, kdy je svazek téměř plný a žádná skupina stop již nemá 16MB volného místa.)

14.4 NTFS

NTFS je souborový systém Windows NT (tj. od Microsoftu). Používá se od roku 1993 a od té doby doznal v několika verzích několika úprav, které jsou však jen kosmetické ve srovnání s bouřlivým vývojem UFS. Systém je od začátku flexibilně a moderně navržen a i přes strmý růst kapacit disků za posledních 10 let (s čímž se některé systémy obtížně vypořádávají) je bez problémů použitelný na discích libovolných kapacit. Přitom „masově“ se začal používat až od Windows 2000 (rok 1999), kdy nahradil dosluhující FAT32.

Microsoft uvádí, že NTFS byl navržen (jako náhrada FAT) pro splnění těchto cílů:

1. Zajištění integrity souborového systému při selhání systému.
2. Možnost ochrany citlivých dat před neoprávněným přístupem.
3. Možnost softwarově řízené datové redundance (RAID-1 a RAID-5, viz kap.14.5 na straně 163).

Klastry jsou zde od začátku číslovány 64bitově, což posouvá teoretické kapacity svazků až do řádů miliard GB. Současné 32bitové Windows používají jen 32 bitů, čímž je limit svazku přibližně 130TB (což zřejmě také ještě nějakou dobu bude stačit).

Na rozdíl od FAT je v NTFS integrována podpora přístupových oprávnění, přičemž na rozdíl od UFS používá NTFS daleko flexibilnější systém ACL. Systém je implicitně žurnálovaný, takže při selhání systému se na rozdíl od FAT nemusí spouštět kontrola disku. Názvy souborů jsou 255 znaků unicode a v režimu POSIX se rozlišují velká a malá písmena (což standardně ve Windows není zapnuto). Zajímavostí také je, že čas je ukládán výhradně v GMT, takže každoročně při přechodu na letní čas a zpět se jakoby posouvá čas všech souborů na disku o hodinu. (Na disku se nic nemění, jen uživatel se přesune do jiného časového pásma. Jinak řečeno, jde tedy jen o změnu zobrazení času na monitoru.)

Průvodce studiem

Oprava dat podle žurnálu proběhne prakticky okamžitě, ve srovnání s minutami či desítkami minut nutnými na kontrolu FAT32 disku (pouze pro kontrolu, čas pro opravu není započten). Paradoxně, právě Windows 9x, které je proslulé neustálým padáním, NTFS nepodporuje a tak po každém restartu je nutno zdlouhavě kontrolovat FAT32 disky. Naopak Windows NT, které tak nepadá, používá implicitně NTFS systém, u kterého se kontroly provádět nemusí.

Struktura svazku

Na začátku svazku je tzv. bootovací sektor (partition boot sector, délka až 16 sektorů). Následuje master file table (MFT), což je seznam všech souborů na disku. Následují systémové soubory a pak ostatní soubory.

Skupina 16 systémových souborů (tzv. metasoubory) je klíčová pro chod disku, proto jsou odkazy na ně uloženy vždy ve stejném pořadí na začátku MFT. V systému NTFS je všechno soubor, proto nepřekvapí, že prvním souborem v MFT je samotná MFT. Druhým souborem je MFT2 obsahující kopii prvních 16 souborů, který je uložen vždy uprostřed svazku a slouží jako ochrana před fyzickým poškozením začátku disku (svazku). Další v řadě je log (žurnál),

potom hlavička disku, definice atributů, kořenový adresář, bitmapa volných klastrů, boot sektor, uživatelské kvóty, tabulka akordance písmen. Z toho seznamu je třeba vyjasnit jen dvě položky: Soubor definice atributů obsahuje seznam rozšiřujících atributů podporovaných na tomto svazku a udává, podle kterých z nich lze indexovat (řadit soubory) a které z nich se žurnálují. Tabulka akordance (anglicky accordance) slouží jako pomůcka pro práci s malými a velkými písmeny. Jména souborů jsou v UTF-16 a pomocí této tabulky lze rychle porovnávat názvy bez ohledu na malá a velká písmena (což je výpočetně náročné, ale je to výchozí chování všech systémů Microsoftu).

Pro tabulku MFT je vyhrazeno začátečních 12% prostoru svazku – je to prostor, kam dle potřeby může MFT růst (a bez fragmentace). Soubory jsou ukládány do zbylých 88%. V okamžiku, kdy není další místo pro soubory, může být prostor pro MFT zmenšen, v okamžiku opětovného uvolnění místa opět zvětšen. MFT se tím může fragmentovat, což může zpomalit chod disku, ale neovlivňuje to funkčnost.

Záznamy o souborech jsou v MFT, včetně jména. (Zde rozdíl od UFS, kde jména jsou zvlášť v adresářích a zbytek je v inode.) Položky MFT mají pevnou velikost a každý soubor může v případě potřeby použít víc položek (třeba při velmi dlouhém názvu a řadě atributů), nemusí přitom jít o položky přímo navazující. Samotné tělo souboru je pak kdekoliv na disku. Pokud je však soubor malý, může být jeho tělo připsáno přímo do položky souboru v MFT. Díky tomu malé soubory nezabírají prakticky žádné místo na disku (což je velký rozdíl oproti jiným souborovým systémům). Tento netradiční přístup se v praxi velmi osvědčil.

Soubor může obsahovat více proudů dat. Ve skutečnosti všechny informace o souboru jsou nazývány *atributy* a jsou to proudy. A v NTFS je tělo souboru je jen jedním z mnoha takových atributů, takže potom nepřekvapí, že těl může být více. Možnost víceproudových souborů je v poslední době poměrně populárním tématem odborných studií, avšak reálné použití je stále poměrně malé. Konkrétně ve Windows to používají některé databázové servery a také údaje zadané na kartě „Souhrn“ ve vlastnostech souboru jsou uloženy v alternativním proudu. Průzkumník však u souborů obecně alternativní proudy nijak viditelně neoznačuje, dokonce jejich velikost ani nezahrnuje do délky souboru. Programy ve Windows obecně s alternativními proudy moc nepočítají, je proto možné, že při práci se soubory pomocí některých programů dojde ke ztrátě dat (alternativní proud se prostě ztratí při amatérském překopírování souboru atp.).

Adresáře

NTFS ukládá adresáře jako B+ stromy s prvky řazenými dle jména (max. 255 znaků UTF-16). Výhodou pak je rychlé vyhledání souboru (řádově $\log n$ oproti n u běžného lineárního seznamu). Přidání nové položky do stromu i lineárního seznamu je přibližně stejně náročné. Položky adresáře obsahují jméno souboru, jeho atributy a odkaz na soubor do MFT (tj. jeho číslo). Oproti UFS tedy NTFS ukládá atributy do adresáře, nikoliv k souboru. Jak už víme, kořenový adresář je jedním z metasouborů svazku, další adresáře pak mohou být obsaženy v libovolném adresáři.

Žurnálování

Jednou z nejznámějších vlastností NTFS je žurnálování diskových operací. Systém NTFS byl prvním souborovým systémem, který rozšířil žurnálování mezi „masy“ uživatelů. Ve srovnání s FAT mohou být některé operací kvůli žurnálování pomalejší, ale v praxi je naopak díky chytrému systému cachování řada operací na NTFS svazku rychlejší. NTFS žurnáluje pouze metadata, což je de facto vše kromě datových proudů. Při řádném vypnutí systému (či odpojení svazku) systém na svazek uloží záznam, že svazek byl bezpečně odpojen (unmounted). Při selhání systému tam tato informace chybí, systém pak podle žurnálu zruší nedokončené operace, či dokončí operace, které jsou v žurnálu, ale nebyly provedeny na disku.

Komprese

NTFS implicitně podporuje kompresi souborů. Používá přitom jednoduché kompresní schéma s cílem, aby komprese neovlivnila chod systému negativně. Standardně není komprese zapnuta pro všechny soubory; je to příznak (atribut) použitelný na úrovni jednotlivých souborů či adresářů.

Popisem, jak funguje komprese, si zároveň ukážeme, jak vlastně NTFS ukládá informace o tom, kde je který soubor na disku. Záznam o souboru v MFT obsahuje tabulku mapování virtuálních klastrů (VCN – virtual cluster number, tj. číslo klastru v souboru) na logické klastry (LCN – logical cluster number, tj. číslo klastru ve svazku). Tato tabulka má tři sloupce: VCN, LCN, délka. První řádek začíná na $\mathrm{VCN} = 0$, další řádky pak mají VCN vyšší o délku předchozího řádku. Číslo LCN na příslušném řádku označuje počátek souvislého bloku klastrů. V ideálním případě je pak celý soubor popsán jediným řádkem, v nejhorším případě je každý klastr (VCN) na samostatném řádku.

NTFS umožňuje přeskakovat prázdné klastry. V mapovací tabulce může VCN řádku být větší, než VCN předchozího řádku + délka. Pak vynechané klastry (VCN) jsou prázdné (plné nul). Soubory využívající tuto vlastnost jsou tzv. řídké soubory (sparse). Toto chování není standardně zapnuto, každý proces zapisující soubor si může nastavit, že má být řídký.

NTFS kromě řídkých souborů však podporuje také skutečnou kompresi. Tu můžeme zapnout i uživatelsky (z karty vlastnosti souboru) nebo opět programově. Komprese funguje tak, že při zápisu se systém snaží zkompresovat bloky 16 klastrů souboru slovníkovým algoritmem (specifická varianta LZ). Pokud výsledek zabírá méně než 16 klastrů, jsou data uložena zkompresovaná. V opačném případě je tento blok 16 klastrů uložen bez komprese. Které části souboru jsou takto zkompresované, se jednoduše pozná opět podle tabulky mapování VCN na LCN, tentokrát podle sloupce délka. Práce se souborem je zcela transparentní, bez ohledu na to, zda je nebo není řídký či kompresovaný.

14.5 RAID

V předchozích sekcích jsme si tedy představili několik konkrétních souborových systémů, závěr této kapitoly bude věnován technologii RAID (Redundant Array of Independent Disks) sloužící ke zvýšení spolehlivosti a/nebo rychlosti fyzických disků (čili je to na jiné úrovni, než souborový systém). Zatímco rozdělení disku na svazky umožňuje rozdělit jeden fyzický disk na samostatné souborové systémy, RAID naopak umožňuje vytvořit jeden souborový systém na více discích dohromady, a přitom také zvýšit rychlost a spolehlivost. Systémy RAID mohou být implementovány jak hardwarově (řadičem disků), což je preferováno, tak softwarově (emulace operačním systémem). RAID přitom rozeznává několik samostatných *úrovní*, z nichž ty nejdůležitější si představíme v následujících sekcích.

14.5.1 RAID 0 (stripping – pruhování)

Úroveň 0 neřeší bezpečnost, ale pouze rychlost. Data svazku jsou po částech (pruhy – stripes) ukládána na více disků. Tím, že se data ukládají střídavě, při čtení většího množství dat se pracuje se všemi disky najednou, čímž se rychlost pro n disků teoreticky zvyšuje až na n-násobek rychlosti jednoho disku. Skutečná rychlost je pak závislá zejména na rychlosti řadiče a použité sběrnice.

Spolehlivost pole o N discích vyjádřená jako MTTF (Mean Time To Failure, česky průměrná doba do poruchy) či MTBF (Mean Time Between Failures, česky průměrná doba mezi poruchami) RAID 0 je rovna spolehlivosti jednoho disku dělené počtem disků.

$$MTTF_{pole} = \frac{MTTF_{disk}}{N}$$

14.5.2 RAID 1 (mirroring – zrcadlení)

Úroveň 1 řeší bezpečnost. Data svazku jsou zrcadlena na více disků. Všechny disky tedy obsahují stejná data a systém může být v provozu, dokud je v provozu alespoň jeden disk. Jedná se tedy o poměrně nehospodárný způsob zajištění bezpečnosti (porovnejte s dalšími úrovněmi popsanými níže).

RAID 1 při čtení může při čtení poskytovat určitě zrychlení, podobně jako RAID 0. Je toho dosaženo tím, že data se čtou střídavě z různých disků – transfer time se tedy násobí počtem disků, zatímco seek time zůstává stejný jako u disku samostatného. Typické využití je na víceúlohových systémech, kde z každého disku lze číst jiný soubor (čili máme něco jako dvoujádrový procesor, ale u disku). Při zápisu je u RAID 1 nepatrné zpomalení oproti jednomu disku. (Pro srovnání: RAID 0 zrychluje transfer i seek time.)

Spolehlivost pole o N discích vyjadřujeme v závislosti na procentuální pravděpodobnosti selhání jednoho disku v určitém časovém úseku. Označíme-li tuto pravděpodobnost jako p^F, pak pravděpodobnost selhání celého systému je rovna pravděpodobnosti selhání všech disků v poli během tohoto časového úseku.

$$p^F_{pole} = (p^F_{disk})^N$$

14.5.3 RAID 2

Úroveň 2 zajišťuje vyšší bezpečnost použitím Hammingova kódu. Data jsou dělena na jednotlivé disky na úrovni bitů a Hammingův kód zajišťuje detekci a korekci dat.

Průvodce studiem

Hammingův kód je ECC algoritmem (error checking & correction. Teorie kódování je samostatným předmětem v magisterském studiu informatiky je téma nad rámec našeho studia operačních systémů. Pro příklad uveďme, že běžný je například (7,4)–kód, kde v každých 7 bitech jsou uloženy 4 bity dat a 3 bity informace. Každá taková 7bitová jednotka je schopna detekovat 2 bitové chyby (tj. libovolné až dva bity z této skupiny s chybnou či žádnou hodnotou lze detekovat) a opravit 1 bitovou chybu (tj. jeden bit chybný či bez hodnoty ze skupiny lze opravit).

Dodejme, že RAID 2 je jedinou úrovní standardu RAID, která se v praxi (alespoň prozatím) vůbec nepoužívá.

14.5.4 RAID 3

Úroveň 3 používá namísto Hammingova opravného kódu jednoduchou paritní informaci. Pro N disků v poli vždy $N - 1$ disků obsahuje data a poslední disk obsahuje paritní informaci. Data jsou na disky dělena na úrovni bajtů, což znemožňuje provádět paralelní čtení, neboť při čtení souboru musí být raménka všech disků na stejné pozici.

Paritní disk používá jednoduchý algoritmus xorující data na odpovídajících pozicích ostatních disků. Systém tedy umí detekovat 1 bitovou chybu ve skupině. Opravit lze pouze data, u kterých víme, který z disků je vadný. Čili například při úplném selhání některého disku či fyzicky vadném sektoru (což jsou mimochodem nejčastější v praxi se vyskytující vady) lze ze zbylých disků data dopočítat.

14.5.5 RAID 4

Úroveň 4 je obdobou předchozí, ovšem dělí data na větší bloky a díky tomu (při odpovídající podpoře řadiče disků) umí číst více dat paralelně (v rámci jednoho bloku).

Průvodce studiem

Paralelní čtení je operace, při kterém se čte více souborů současně. Jde tedy o funkci velmi žádanou na serverech, přitom bez použití RAID polí je to nemožné, neboť disk má jen jedno raménko a to může v daný okamžik být jen na jediné pozici. Proto je na obyčejných jednotlivých discích vždy daleko rychlejší nejprve načíst celý jeden soubor, a pak přejít na čtení jiného souboru. Při paralelním čtení RAID pole se využije toho, že každý jeden blok je vždy pouze na jednom disku, takže mezitím, co se čte, mohou ostatní disky číst jiná data. (Samozřejmě to funguje pouze tehdy, když máme štěstí a právě požadovaná data opravdu jsou rozložena na různé disky.)

14.5.6 RAID 5

Úroveň 5 je obdobou předchozí s tím rozdílem, že paritní informace je střídavě ukládána na jednotlivé disky. Výsledkem je ještě vyšší možná rychlost při paralelním čtení, protože zatímco u RAID 4 není při čtení nikdy poslední disk použit (protože neobsahuje žádná data, ale jen paritní informace), u RAID 5 je rozložení dat rovnoměrné na všechny disky a všechny se tedy mohou zúčastnit paralelního čtení.

RAID 5 je v praxi nejpopulárnější úroveň, implementována běžně bývá hardwarově i softwarově. Lze ji použít pro minimálně 3 disky a výhodou je poměrně nízká cenová náročnost. (Pro libovolných N disků nám stačí vždy jen jeden disk navíc pro paritní informace, takže čím více disků máme, tím méně musíme zaplatit navíc.)

RAID 5 pole je pomalé při zápisu krátkých bloků dat, protože zapisujeme-li méně než celý pruh, tak musíme použít metodu čti–změň–zapiš jak pro samotný blok dat, tak pro jemu odpovídající blok paritní. Naopak rychlost čtení je velmi dobrá, téměř porovnatelná s RAID 0 (zejména pro větší počty disků). Při náhlém vypnutí či selhání systému v okamžiku, kdy na disk již byla zapsána nová data, ale ještě ne nová paritní informace, může diskové pole dále fungovat, protože při čtení jsou použity jen datové bloky. Pokud však tato vadná či chybějící paritní informace není opravena dříve, než dojde k poruše disku, pak dojde k nenávratné ztrátě dat. Tyto případy u hardwarových řešení RAID 5 eliminuje použití baterií zajištěného řadiče, který i při výpadku proudu nevypne disky dříve, než jsou všechna data konzistentně zapsána. U softwarového řešení RAID 5 toto samozřejmě možné není.

14.5.7 RAID 6

Šestá úroveň již nepatří do původního standardu, ovšem v poslední době se stále častěji používá. Jde o rozšíření páté úrovně o druhý paritní blok v každém pruhu. Pomocí Reed–Solomon kódu je pak možno opravit libovolnou jednu chybu. Ještě zajímavější je však schopnost opravit až dvě chyby, pokud známe jejich pozice. V praxi tedy je možno spolehlivě data opravit i při současném fyzickém poškození dvou disků. (Opět: Při fyzickém poškození celého disku nebo jeho části je toto poškození obvykle přesně identifikovatelné – přesně víme, který disk nefunguje.)

RAID 6 je v praxi používán tam, kde je třeba vyšší bezpečnost. Při poruše disku se tento okamžitě vymění (kvalitní řadiče RAID umožňují disky měnit za běhu počítače) a řadič dopočítá chybějící informace na nově zařazený disk. Pokud během toho dojde k další poruše (jiného

disku), opět je možno jej za běhu vyměnit a systém stále pokračuje v bezchybném provozu. Jakmile řadič dokončí aktualizaci dat na nově vložených discích, systém je připraven na další dvě poruchy. (V každém okamžiku systém funguje, pokud alespoň $N - 2$ disků je v provozu. Jsou-li porouchány jen části disků, tak celý systém může být v bezchybném provozu teoreticky i při selhání všech disků, dokud jsou u každého pruhu chyby na nejvýše dvou discích.)

Poznámka: Podle některých definic lze za RAID 6 považovat každou implementaci diskového pole, která „přežije“ selhání dvou disků. Nemusí tedy nutně jít jen o použití Reed–Solomon kódu.

14.5.8 Softwarová emulace

Jak již bylo řečeno v úvodu, některé operační systémy podporují RAID technologii softwarově, tedy bez potřeby jakéhokoliv speciálního hardwaru. Např. Windows NT již od dávných verzí obsahuje softwarovou implementaci úrovní 0, 1 a 5 a tyto jsou nejčastěji podporovány i jinými systémy. Softwarová emulace RAID je v systému implementována jako další vrstva abstrakce napojená na ovladač disku, nebo přímo jeho součást. Kromě jistého zatížení procesoru, který se musí o organizaci (kódování) dat na discích pochopitelně starat, to s sebou nese i některá další specifika: Především systém musí být nejprve nabootován, takže obvykle potřebuje mít systémový svazek na ne–RAID disku. Na druhu stranu tyto systémy obvykle umožňují vytvářet RAID svazky z částí disků, tedy z logických částí (partition) a ne nutně z celých disků.

U softwarových řešení RAID polí nemůže být zcela zaručena bezpečnost, neboť při selhání systému během zápisu nelze čistě softwarově zaručit, že všechny související sektory byly zapsány. U hardwarových řadičů RAID toto může být řešeno například záložní baterií, která zajistí, že disky se vypnou až po dosažení konzistentního stavu.

Shrnutí

V této kapitole jsme si představili některé konkrétní souborové systémy, které se nejčastěji používají v praxi. Byl to jmenovitě FAT systém používaný pro výměnná média, UFS používaný v různých verzí Unixu a NTFS používaný ve Windows NT. V závěru kapitoly jsme si ještě představili RAID pole, což je důležitý doplněk souborových systémů používaný zejména na serverech a obecně všude tam, kde je vyžadována vyšší bezpečnost.

Pojmy k zapamatování

- FAT (file allocation table)
- dlouhý název (LFN – long file name)
- UFS (Unix file system)
- inode
- NTFS (NT file system)
- MFT (master file table)
- akordance písmen
- RAID (redundant array of independent drives/disks)
- MTTF (mean time to failure)
- MTBF (mean time between failures)

Kontrolní otázky

1. *FAT tabulku lze do cache načítat po jednotlivých klastrech v okamžiku jejich první potřeby, nebo jako celek na začátku práce s diskem. Vysvětlete výhody a nevýhody těchto dvou řešení.*

2. *Co se děje s časovými záznamy u souborů v okamžiku přechodu na letní čas nebo zpět? Uveďte pro jednotlivé souborové systémy.*
3. *Pokuste se definovat alternativní způsob implementace RAID 6 (tj. jiný než použití Reed–Solomon kódu).*
4. *Vysvětlete, co je to paralelní čtení a díky čemu jej lze v některých systémech RAID provádět.*
5. *Popište algoritmus a přínos komprese v systému NTFS.*
6. *Popište, jakým způsobem jsou do systému FAT zakomponovány dlouhé názvy. Proč právě u FAT je problematika dlouhých názvu tak důležitá?*
7. *Systém FAT je obvykle popisován jako špatný. Uveďte jeho nevýhody či problémy a vysvětlete, proč na vyměnitelných médiích je stále nejvíce používaný.*
8. *Jmenujte alespoň tři nevýhody softwarových řešení RAID.*

Úkoly k textu

1. Zajímá-li vás, jak přesně fungují opravné kódy zmíněné v této kapitole, přečtěte si o nich další informace na internetu (např. ve Wikipedii [Wiki]).

A Jak vznikají programy

Studijní cíle: V této kapitole se podíváme na postup, při kterém se ze zdrojových textů programu stane skutečný program a jak se pak tento v počítači spustí. Jedná se o doplňkové téma, proto je zařazeno mezi přílohami.

Klíčová slova: překlad, spuštění, knihovna, linker, komponenta

Potřebný čas: 40 minut.

A.1 Překlad

Vyšší jazyky se obvykle překládají přes Assembler.

Vyšší jazyky, jako C/C++ či Pascal, jsou do spustitelného tvaru překládány pomocí dvoufázového překladu. V první fázi se program přeloží do Assembleru, ve druhé fázi se pak Assembler převede do strojového kódu. Tento proces má na různých počítačích a v různých operačních systémech všelijaké nuance, podíváme se tedy pouze na jeden vzorový příklad.

Za překlad vyššího jazyka do Assembleru je vždy odpovědný překladač onoho vyššího jazyka. Některé překladače překládají svůj kód nejprve do jiného vyššího jazyka (např. do C/C++), ze kterého se teprve do Assembleru. Překlad Assembleru probíhá tak, že se jde řádek po řádku a analyzuje se jméno instrukce a typy operandů. Podle toho je určen binární kód této instrukce. Je-li v operandech použito nějaké symbolické jméno (název adresy, proměnné apod.), toto se nahradí jeho skutečnou hodnotou. Není-li v okamžiku použití symbolického jména ještě známa jeho hodnota, proběhne pak ještě další průchod. Většinu programů lze přeložit v nejvýše dvou až třech průchodech.

Modul je výsledek překladu jednoho souboru.

Přeložený výsledek, čili kód z jednoho zdrojového souboru, se uloží do jednoho výstupního souboru. Jedná se o tzv. „modul“, obvykle s příponou *.obj*, ve kterém je již výsledný strojový kód, ovšem jsou v něm také tabulky všech použitých symbolických jmen a jejich výskytů v programu. Modul se také může odkazovat na externí jména, tj. neznámé prvky, které se nacházejí v jiných zdrojových souborech. (Např. funkci pro psaní na obrazovku printf v našem souboru nemáme, přesto ji můžeme zavolat. Překladač do modulu našeho programu uloží informaci, že tento modul potřebuje funkci printf. Odkud tu funkci vzít, již v modulu uvedeno není.) Podobně je v modulu uveden seznam veřejných jmen, tj. jmen nabízených pro použití v jiných modulech. Některé jazyky (např. C/C++) nabízejí standardně všechna svá jména jako veřejná.

Jmenné tabulky obsahují pro každé jméno jeho adresu uvnitř modulu, je-li tam jméno definováno, a seznam adres, kde se toto jméno v modulu používá, je-li tam jméno používáno.

Průvodce studiem

Jména funkcí vyšších jazyků jsou obvykle dekorována, aby nedošlo ke kolizi s nějakým systémovým jménem. Např. eax je jméno registru a funkce se nemůže jmenovat stejně. Jazyk C proto všechna jména dekoruje přidáním znaku _ na začátek názvu. Jazyk C++ pak navíc umožňuje definovat stejně pojmenované funkce lišící se jen v počtu či typech parametrů. Součástí dekorovaného jména je v C++ tedy i zakódovaná informace o počtu parametrů a jejich typech. Takto dekorovaná jména jsou bohužel nesrozumitelná a liší se v jednotlivých překladačích (podle výrobce). Proto v Assembleru spolupracujeme jen s jazykem C.

A.2 Linkování

Při linkování se moduly spojí dohromady.

Spustitelný program vytvoříme procesem zvaným *linkování*; program provádějící linkování

nazýváme *linker*. Jeho úkolem já dát dohromady všechny uvedené moduly, jak naše vlastní, tak moduly vestavěných knihoven vyššího jazyka. Při linkování se projdou všechny jmenné tabulky a odkazy na neznámá jména se vyřeší a napojí napříč moduly. Na rozdíl od překladu, linkování je již poměrně jednoduchý proces práce s mnoha poli.

Kromě vyřešení jmen linkování také provede spojení kódu všech modulů do jednoho bloku. K tomu použije také tabulky použitých adres každého modulu, protože aby moduly mohly být spojeny, musí se jednotlivé moduly posunout v paměti (aby nebyly všechny na stejném místě). Tomuto procesu se říká *relokace*. Toto vyžaduje změnit některé instrukční kódy, konkrétně právě adresní operandy (což je zcela pochopitelné).

Výsledkem linkování je soubor s příponou *.com* nebo *.exe*. První jmenovaný je starý formát, který je přežitkem ze systému CP/M (předchůdce MS-DOSu). Tento typ souborů obsahuje jen čistý kód bez jakékoliv hlavičky. Nahraje se vždy na stejné místo v paměti a spustí se od začátku. Druhá jmenovaná přípona je běžná např. ve Windows. Jiné operační systémy ale používají jiné formáty a i samotný „ekzáč“ může ve skutečnosti mít několik různých vnitřních formátů.

A.3 Knihovny a spouštění programů

Knihovny jsou programy či části programů, které nabízejí svou funkcionalitu jiným knihovnám či programům. Z hlediska překladu a spouštění kódu rozlišujeme knihovny především na statické a dynamické.

Staticky linkovaná knihovna je ve formátu modulu. Před jejím použitím se musí slinkovat dohromady s naším programem a vznikne tak jeden celek. Výhodou takového postupu je rychlejší běh, nevýhodou pak použití více paměti, když spouštíme programy se stejnými knihovnami (každý má svou kopii).

Dynamicky linkované knihovny jsou sdíleny mezi více programy.

Dynamický linkovaná knihovna je ve formátu spustitelného programu. Ve Windows má příponu *.dll*. Formát takového souboru je totožný se souborem *.exe*. V souboru však chybí běžný startovací bod (dll nelze spustit) a navíc je zde seznam zveřejněných jmen. Ten je obdobou jmen zveřejněných v modulu a vytváří ho linker. Výhodou dynamických knihoven je, že umožňují sdílení kódu. Nevýhodou je složitější postup při spuštění: Především musíme opět provést relokaci (posunutí adres, jako v linkeru), protože nejde umístit více knihoven na stejné místo paměti

Dynamicky linkované knihovny je možno k programu připojovat (bindovat) staticky nebo dynamicky. Nejčastěji se používá statické připojení dynamicky linkované knihovny. Tento poněkud tajemný název značí, že připojení knihovny k programu a vyřešení a připojení všech jmen se provede ihned při spuštění programu. Nelze-li některé jméno vyřešit, program ohlásí chybu a nespustí se.

Alternativou je dynamické připojení dynamicky linkované knihovny, kdy se jednotlivá jména řeší až v okamžiku jejich použití. Tento způsob je možno provozovat jen s pomocí podpůrného kódu, protože namísto obyčejného zavolání nějaké funkce v dll knihovně je třeba ručně ošetřit její načtení a připojení daného jména. Tento způsob je tedy pomalejší při běhu, ale je flexibilnější – umožňuje používat i knihovny neznámé při tvorbě programu, měnit knihovny za běhu programu, atp.

Při spuštění programů tedy systém začne od exe souboru. Umístí jej do paměti a pomocí dll knihoven se snaží vyřešit všechna externí jména. Každou knihovnu načte a aktivuje stejně jako samotný exe soubor – knihovna má svůj startovací bod, kde může mít krátký program, kterým se inicializuje. Knihovna se samozřejmě může odkazovat na další knihovny. Každý soubor přitom při načítání do paměti může být umístěn na jiné místo, než kam byl původně určen. V tom případě se provádí relokace adres (stejně jako při linkování). Používá-li nějaký program větší množství svých vlastních knihoven, je vhodné je přeložit tak, aby každá používala jinou výchozí adresu. Tím se umožní spouštět program bez relokace adres a spuštění je tak rychlejší.

A.4 Komponenty

Používání klasických dll knihoven má řadu nevýhod. Proklamovanou výhodou dll totiž mělo být sdílení kódu mezi různými programy. V praxi se to však přeměnilo v noční můru (v literatuře obvykle najdeme výstižný pojem „DLL Hell“ – DLL peklo), neboť řada programů drze umísťuje DLL soubory do adresáře Windows, často i přepíší novou verzi starší verzí téže knihovny apod.

Komponenty jsou moderní náhradou knihoven.

Tento problém řeší komponenty. Existuje přitom několik komponentních technologií různých výrobců (autorů). Microsoft používá ve Windows systém COM (Component Object Model), alternativou je např. systém CORBA (Common Object Request Broker Architecture).

Základním přínosem komponentních systémů je podpora objektově orientovaného přístupu – knihovny jsou třídy či sady tříd. Pro vyřešení DLL Hell je přidána řada informací, především číslo verze, a je odstraněno odkazování jménem souboru – identifikace nyní probíhá jménem knihovny a jménem třídy. V systému může tedy současně koexistovat více verzí stejné knihovny. Příkladem systému, který COM bohatě používá, je DirectX – v počítači máte obvykle mnoho verzí této rozsáhlé knihovny.

Některé komponentní technologie (např. DCOM – Distributed COM) umožňují téměř transparentně také rozmístit komponenty na více počítačů – klient v krajním případě ani nemusí vědět, že používá funkce komponenty, která je fyzicky umístěna na vzdáleném počítači. Tato funkcionalita ale samozřejmě vyžaduje poměrně rozsáhlý kód v pozadí (v operačním systému).

Zvláštní kapitolou jsou platformy Java a .NET. Jsou to moderní objektově orientované systémy, které téměř odstiňují program od operačního systému. Samy tak vlastně vystupují v roli operačního systému. Jedním ze základních kamenů těchto systémů jsou právě komponenty. Microsoft svůj .NET samozřejmě realizoval jakožto jakousi další inkarnaci COM, kde jsou lépe vyřešeny některé problémy, které v COM byly. Systém však je zpětně kompatibilní a spolupráce mezi COM a .NET programy a komponentami je velmi snadná. Oproti tomu Java společnosti Sun je platformově nezávislá a svůj systém komponent umožňuje používat na různých operačních systémech.

Celá problematika komponent je bohužel již nad rámec tohoto kurzu.

Shrnutí

Tato kapitola čtenáře stručně seznámila s problematikou překladu programů, tj. vysvětlila „jak ze zdrojových textů vznikají programy“.

B Ochrana a zabezpečení

Studijní cíle: V této sekci se krátce zmíníme o ochraně a zabezpečení, což je téma spadající do studia operačních systémů, ale je již nad rámec tohoto našeho úvodního kurzu.

Klíčová slova: ochrana, zabezpečení

Potřebný čas: 10 minut.

Téma ochrany a zabezpečení je u operačních systémů poměrně nové, v současnosti je zde však již zcela „zdomácnělé“. O *ochraně* hovoříme u systémů jako takových (ochrana paměti mezi procesy, ACL – ochrana přístupu k souborům). O *zabezpečení* pak hovoříme při práci s lidmi (RSA šifrování, digitální podpis, firewall, login). Toto názvosloví se někdy plete, protože není zrovna intuitivní, nejde však o zásadní problém.

O ochraně byla řeč již v kapitolách týkajících správy paměti a disku. Další velkou doménou je síťové či obecně jakékoliv veřejné prostředí. Právě rozvoj internetu také tuto problematiku dostal do hledáčku zájmu studia operačních systémů, za největší problém minulosti můžeme označit nekódované posílání hesel po síti. Např. běžně používaný síťový protokol FTP je možno velmi snadno odposlouchávat na lokální síti a získat tak hesla uživatelů. Modernější řešení problematiky vzdáleného přihlašování (či obecněji a přesněji autentifikace/ověření) uživatelů řeší např. protokol *Kerberos*, který umožňuje vzájemné zabezpečené ověření obou stran (serveru i klienta) i na síti, která sama zabezpečená není. Kerberos používají Windows 2000 a novější, Mac OS X i další moderní systémy.

Kerberos ověřuje uživatele v síti.

Toto téma je také známo z poloviny 90.let, kdy Microsoft upravil tehdejší verze Windows NT do podoby, která vyhovuje zvýšeným bezpečnostním standardům americké armády. Standard ministerstva obrany USA TCSEC (Trusted Computer System Evaluation Criteria) definuje několik úrovní bezpečnosti operačních systémů od A (nejvyšší bezpečnost) až po D (žádná bezpečnost). Windows NT získalo certifikaci C2, která vyžaduje mimo jiné správu přístupových oprávnění na úrovni objektů systému (implementováno jako systém ACL), nulování přidělené paměti, možnost odejmout přístupová oprávnění (implementováno jako DACL) nebo povinný audit událostí týkajících se zabezpečení. Ačkoliv řadu z těchto funkcí běžní uživatelé nepotřebovali (a zejména ne v tehdejší době, před masovým rozšířením internetu), Windows NT díky tomu získalo na poli bezpečnosti poměrně velký náskok před konkurenčními systémy, které tuto problematiku obvykle neřešily vůbec. (Unixové systémy například běžně posílaly nezašifrovaná hesla po síti tak, že bylo velmi snadné je odchytit třetím počítačem v síti.

Průvodce studiem

Jak známo, každá mince má dvě strany. I přes dobrý návrh systému Windows NT jeho skutečnou bezpečnost limitují chyby v systému. Ty jsou samozřejmě ve všech operačních systémech, ovšem v těch více používaných jsou více „na ráně“.

Pojmy k zapamatování

- ochrana
- zabezpečení
- Kerberos
- TCSEC (Trusted Computer System Evaluation Criteria)

C Seznam obrázků

Reference

[Bac86] Maurice J. Bach. *Design of the Unix Operating System.* Prentice Hall, **1986**. 486 pp., ISBN 0-13-201799-7. český překlad: *Principy operačního systému Unix.* Softwarové Aplikace a Systémy, Praha, **1993**. 514 pp. ISBN 80-901507-0-5.

[BoCe05] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel, 3.vydání.* O'Reilly, **2005**. 942 pp., ISBN 0-596-00565-2.

[Bra94] Michal Brandejs. *Mikroprocesory Intel Pentium a spol.* Grada, **1994**. ISBN 80-7169-041-4.

[Her91] Maurice Herlihy. Wait–Free Synchronization. In: *ACM Trans.* Program. Lang. Syst. 13(1):124–149.

[Hof90] Micha Hofri. Proof of a Mutual Exclusion Algorithm. In: *Operating Systems Review*, January 1990.

[IA32-1] *IA-32 Intel Architecture Software Developer's Manual.* Volume 1: Basic Architecture. Intel, 2006. 476 pp. (Existuje ve verzích pro jednotlivé procesory Intel, ke stažení na www.intel.com.)

[IA32-3A] *IA-32 Intel Architecture Software Developer's Manual.* Volume 3A: System Programming Guide, Part 1. Intel, 2006. 630 pp. (Existuje ve verzích pro jednotlivé procesory Intel, ke stažení na www.intel.com.)

[Kep08a] Aleš Keprt. *Assembler.* Univerzita Palackého, 2008. Studijní text pro distanční vzdělávání, v přípravě, dostupný studentům na adrese http://www.keprt.cz/vyuka/.

[Kep08b] Aleš Keprt. *Systémové programování v jazyce C#.* Univerzita Palackého, 2008. Studijní text pro distanční vzdělávání, v přípravě, dostupný studentům na adrese http://www.keprt.cz/vyuka/.

[Knu98] Donald E. Knuth. *Art of Computer Programming, volume 3: Sorting and Searching.* Addison-Wesley Professional, 2.vydání, **1998**. ISBN 0-201-89685-0.

[Lov03] Robert Love. Introducing the 2.6 Kernel. In *Linux Journal.* http://www.linuxjournal.com/article/6530

[McMa06] Richard McDougall, Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2.vydání. Prentice Hall, **2006**. 1072 pp., ISBN 0-13-148209-2. http://www.informit.com/content/images/0131482092/samplechapter/mcdougall_ch15.pdf

[MSDN] *Microsoft Developer Network.* http://www.microsoft.com/msdn/

[Pat71] Suhas Patil. *Limitations and capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes.* technical report, MIT, **1971**.

[Pet81] Gary L. Peterson. Myths About the Mutual Exclusion Problem. In: *Information Processing Letters*. 12(3):115–116, 1981.

[SGG05] Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating Systems Concepts, 7. vydání.* John Wiley & Sons, **2005**. 944 pp., ISBN 0-471-69466-5.

[SoRu05] David A. Solomon, Mark E. Russinovich. *Microsoft Windows Internals, 4.vydání.* Microsoft Press, **2004**. 976 pp., ISBN 0-7356-1917-4.

[Sta05] William Stallings. *Operating Systems, 5.vydání.* Prentice Hall, **2005**. ISBN 0-13-147954-7.

[Tan01] Andrew Tanenbaum. *Modern Operating Systems, 2.vydání.* Prentice Hall, **2001**. ISBN 0-13-031358-0.

[Več04] Arnošt Večerka. *Základní algoritmy.* Text distančního vzdělávání, Katedra informatky, UP Olomouc, **2004**.

[Wiki] *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/