SLEZSKÁ UNIVERZITA V OPAVĚ
FILOZOFICKO-PŘÍRODOVĚDECKÁ FAKULTA
ÚSTAV INFORMATIKY

OPERAČNÍ SYSTÉMY

Texty k přednáškám

Poslední změny: 2. července 2007

Mgr. Šárka Vavrečková
fpf.slu.cz/~vav10ui

Opava 2007

OBSAH

1 Úvod do operačních systémů **1**
- 1.1 Co je to operační systém . . . 1
- 1.2 Funkce operačního systému . . . 3
- 1.3 Typy operačních systémů . . . 4
- 1.4 Realtimové operační systémy . . . 5
- 1.5 Distribuované operační systémy . . . 7

2 Struktura operačních systémů **10**
- 2.1 Základní typy struktur operačních systémů . . . 10
- 2.2 MS-DOS a Windows s DOS jádrem . . . 11
- 2.3 Windows řady NT . . . 16
- 2.4 Systémy Unixového typu . . . 19

3 Správa paměti **23**
- 3.1 Modul správce paměti . . . 23
- 3.2 Reálné metody přidělování paměti . . . 25
  - 3.2.1 Přidělení jedné souvislé oblasti paměti . . . 25
  - 3.2.2 Přidělování bloků pevné velikosti . . . 26
  - 3.2.3 Dynamické přidělování bloků paměti . . . 27
  - 3.2.4 Segmentace . . . 28
  - 3.2.5 Jednoduché stránkování . . . 30
- 3.3 Řešení fragmentace paměti . . . 31
- 3.4 Virtuální paměť . . . 33
  - 3.4.1 Stránkování na žádost . . . 34
  - 3.4.2 Segmentace se stránkováním na žádost . . . 37
  - 3.4.3 Swapování procesů . . . 37

3.5 Správa paměti v některých operačních systémech 38
3.5.1 MS-DOS a Windows 38
3.5.2 Unixové systémy včetně Linuxu 40
3.5.3 MacOS 41

4 Procesy **43**
4.1 Evidence procesů 43
4.2 Běh procesů a multitasking 45
4.3 Multithreading 49
4.4 Správa front procesů 51
4.5 Přidělování procesoru 53
4.5.1 Fronta (FCFS) 54
4.5.2 Cyklické plánování (RR) 54
4.5.3 Nejkratší úloha (SPN) 55
4.5.4 Priority 56
4.5.5 Kombinace metod s více frontami 57
4.6 Komunikace procesů 58

5 Synchronizace procesů **60**
5.1 Úvod do problematiky 60
5.2 Petriho sítě 61
5.3 Základní synchronizační úlohy 63
5.3.1 Kritická sekce 63
5.3.2 Producent–konzument 64
5.3.3 Model–obraz 66
5.3.4 Čtenáři–písaři 67
5.3.5 Pět hladových filozofů 68
5.3.6 Souběh procesů 69
5.4 Implementace čekání před kritickou sekcí 70
5.5 Synchronizační nástroje operačního systému 74
5.5.1 Semafory 75
5.5.2 Mechanismus zpráv 78
5.5.3 Monitory 79
5.5.4 RPC 79

6 Uváznutí procesů (Deadlock) **81**
6.1 Základní pojmy 81
6.2 Popis stavu přidělení prostředků 82
6.3 Podmínky vzniku uváznutí 83
6.4 Prevence uváznutí 84
6.5 Předpovídání uváznutí 86
6.5.1 Graf nároků a přidělení prostředků 86

6.5.2 Bankéřův algoritmus . . . 87
6.6 Detekce uváznutí . . . 89
6.6.1 Úprava grafu přidělení prostředků . . . 90
6.6.2 Úprava Bankéřova algoritmu . . . 90
6.7 Reakce při zjištění zablokování . . . 92

7 Správa periferií **93**
7.1 I/O systém . . . 93
7.2 Druhy periferií . . . 94
7.3 Ovladače . . . 95
7.4 Přerušení . . . 96

8 Paměťová média **99**
8.1 Základní pojmy . . . 99
8.2 Adresářová struktura . . . 101
8.3 Soubory a systém souborů . . . 104
8.4 Souborové systémy ve Windows . . . 106
8.4.1 Starší verze souborových systémů typu FAT . . . 106
8.4.2 VFAT a FAT32 . . . 109
8.4.3 Souborový systém NTFS . . . 111
8.4.4 Srovnání souborových systémů pro Windows . . . 114
8.5 Souborové systémy pro Linux . . . 115
8.5.1 VFS . . . 115
8.5.2 Souborové systémy typu ext$x$fs . . . 116
8.5.3 Další žurnálovací souborové systémy . . . 120
8.5.4 Virtuální souborové systémy . . . 121
8.5.5 Srovnání Linuxových souborových systémů . . . 122

9 Správa disků **123**
9.1 Problémy s BIOSem . . . 123
9.2 Základní pojmy . . . 124
9.3 Struktura disku . . . 124
9.4 Nástroje pro správu disků . . . 126
9.5 Zaváděcí programy . . . 128
9.6 Možnosti instalace operačních systémů . . . 132
9.7 Emulace jiného operačního systému . . . 133
9.7.1 Virtuální počítače . . . 134
9.7.2 Emulátory operačního systému a podsystémy . . . 135

10 Grafický subsystém **137**
10.1 Základní pojmy 137
10.2 X Window System 139
10.3 Technologie rozšiřující možnosti grafiky 141

A Výstupy některých diskových nástrojů pro Windows **144**

Literatura **146**

Rejstřík **149**

KAPITOLA 1

Úvod do operačních systémů

Pojem operační systém budeme v následujícím textu chápat trochu šířeji než je obvyklé. Zahrneme zde také software, který slouží k řízení jakéhokoliv výpočetního systému, včetně programovaných laserových tiskáren.

Tato kapitola je úvodem do problematiky, seznámíme se zde se základními pojmy, definicí operačního systému, funkcemi a typy operačních systémů.

1.1 Co je to operační systém

Pro definování operačního systému použijeme následující pojmy:

Výpočetní systém (například počítač) je stroj na zpracování dat provádějící samočinně předem zadané operace.

Instrukce – nejkratší, již dále nedělitelný povel, těmto povelům rozumí procesor (viz dále).

Zakázka – pokyn, který má výpočetní systém provést.

Fyzické prostředky výpočetního systému jsou:

- *procesor* – vykonává zadané instrukce, určuje *hardwarovou platformu* systému (např. Intel x86, x86-64, AMD, AMD64, PowerPC, Alpha, MIPS, atd.), ve výpočetním systému předpokládáme existenci alespoň jednoho procesoru,

- *vícejádrový procesor* – procesor s více jádry, tedy jediný integrovaný obvod s více jádry procesorů (narozdíl od víceprocesorového systému, kde má každé „jádro“ vlastní integrovaný obvod) – dnes se objevují dvoujádrové procesory, neplést si s víceprocesorovým systémem, kde každý procesor má vlastní integrovaný obvod,
- *vnitřní paměť* (operační paměť) – rychlá, obvykle chipy, podle různých vlastností rozlišujeme RAM (Random Access Memory), ROM (Read-Only Memory), DRAM, SDRAM, atd.), používá se obvykle během výpočtu a počítá se s tím, že po dokončení výpočtu budou zabrané adresy uvolněny,
- *vnější paměť* – slouží k uložení dat a programů, které zrovna nejsou zpracovávány, je stálá (relativně), jsou to pevné disky (HD – Hard Disk), CD, DVD, diskety, USB flash disky, paměťové karty, atd.,
- *vstupně-výstupní systém* (V/V, I/O systém, periferní zařízení) – souhrn všech zařízení určených pro komunikaci s okolím, například monitor, tiskárna, klávesnice.

Logické prostředky výpočetního systému jsou:

- *uživatel* – každý, do zadává zakázku výpočetnímu systému,
- *úloha* (job) – posloupnost (obecně souhrn) činností potřebných ke splnění zakázky, jde tedy o specifikování postupu řešení zakázky,
- *krok úlohy* – část úlohy, prvek posloupnosti provedení úlohy obvykle představující spuštění konkrétního programu (úloha může být posloupností více programů, jejichž práce probíhá simultánně nebo navazuje),
- *proces* – instance úlohy nebo kroku úlohy, je prováděn ve vnitřní paměti za použití konkrétních dat.

Paměťový prostor systému je souhrn všech pamětí systému, vnitřní + vnější paměti.

Paměťový prostor procesu je souhrn všech paměťových možností procesu, tedy jemu přidělená operační paměť pro programový kód a data procesu.

Adresový prostor procesu je paměťový prostor ve vnitřní paměti, který je vyhrazen tomuto procesu. Je to paměťový prostor procesu, na kterém jsou zavedeny adresy.

Holý počítač je výpočetní systém s pouze nejzákladnějším paměťovým vybavením, to se obvykle nazývá BIOS.

Definice 1.1 Operační systém *výpočetního systému je správce fyzických prostředků daného systému, který zpracovává pomocí logických prostředků úlohy zadané uživatelem. Pod pojmem* softwarová platforma *systému obvykle chápeme právě operační systém.*

1.2 Funkce operačního systému

Operační systém má mnoho funkcí, z nichž některé jsou nutné a vyplývají už z definice operačního systému, jiné až tak nutné nejsou a ne každý operační systém je zajišťuje. Následující výčet není úplný, specializované operační systémy mohou zajišťovat i mnoho dalších jiných funkcí. Nejdůležitějším funkcím jsou vyhrazeny samostatné kapitoly.

Správa paměti představuje vedení evidence vnitřní paměti, přidělování paměti procesům, řešení situací vznikajících při nedostatku paměti, správu virtuální paměti.

Správa procesů znamená evidenci spuštěných procesů, plánování přidělování procesoru, sledování stavu procesů, zajišťování komunikace mezi procesy.

Správa periferií zahrnuje vytváření rozhraní mezi I/O zařízeními a procesy, sledování stavu zařízení, přidělování zařízení procesům a řešení možných kolizí s tím souvisejících, atd.

Správa systému – v moderních systémech je obvyklé rozlišování různých režimů práce systému, alespoň uživatelský a privilegovaný. V uživatelském režimu probíhají běžné činnosti, zatímco privilegovaný režim je určen pro údržbu, instalaci, konfiguraci. Můžeme zde zahrnout také bezpečnostní funkce systému – ochranu proti škodlivým kódům (např. viry), poruchám a neoprávněnému přístupu.

Správa souborů (týká se dat na vnějších paměťových médiích) znamená nejen vytváření rozhraní umožňujícího procesům přistupovat k souborům (a také jiným datům) jednotným způsobem, ale také udržování informací o struktuře souborů na disku, kontrolu přístupových práv procesů k souborům.

Správa uživatelů – systém vede informace o uživatelích systému a jejich činnosti, zajišťuje přihlašování a odhlašování uživatelů.

Správa úloh – totéž, co se týká uživatelů, týká se také úloh a jejich průběhu.

Uživatelské rozhraní (user interface – UI) je rozhraní mezi uživatelem a systémem. Jedná se o sadu programů, které slouží ke komunikaci mezi uživatelem a operačním systémem.

Programové rozhraní je rozhraní mezi programy (procesy) a výpočetním a operačním systémem, obvykle se označuje API (Application Programming Interface). Většinou je představováno sadou *knihoven* (ve Windows např. DLL knihovny), které může program využívat pro svou práci (grafické prvky rozhraní, dialogová okna, funkční prvky, rozhraní časovače, atd.).

1.3 Typy operačních systémů

Operační systémy dělíme podle počtu ovládaných procesorů na

- *jednoprocesorové* (monoprocesorové) – Windows s DOS jádrem (verze 9x, ME),
- *víceprocesorové* (multiprocesorové) – unixové systémy včetně Linuxu, Windows s NT jádrem (NT, 2000, XP, Vista), dokážou rozplánovat alespoň některé úlohy tak, aby mohly být zpracovávány na více procesorech zároveň. Při *asymetrickém multiprocessingu* (ASMP) je jeden procesor vyhrazen pro procesy systému a uživatelské procesy běží na ostatních procesorech, při *symetrickém multiprocessingu* (SMP) může kterýkoliv proces běžet na kterémkoliv procesoru.

Ve skutečnosti i v běžných desktopových počítačích, které neoznačujeme jako víceprocesorové, najdeme více procesorů. Jeden z nich je hlavní, ostatní jsou určeny pro konkrétní činnosti a jejich úkolem je odlehčit hlavnímu procesoru od „rutinních“ nebo speciálních činností a urychlit práci celého systému. Takovou funkci má například grafický procesor na grafické kartě, který přebírá zejména zpracovávání požadavků 3D grafiky. Nejde o víceprocesorový systém, protože pomocné procesory nezpracovávají běžnou sadu instrukcí, ale pouze svou specifickou sadu. Právě s grafickým procesorem pracují OpenGL, Direct3D apod. (viz kap. 10.3).

Podle složitosti správy uživatelů dělíme operační systémy na

- *jednouživatelské* (monouživatelské) – Windows s DOS jádrem,
- *víceuživatelské* (multiuživatelské, multiuser) – unixové systémy, Windows s NT jádrem, mají propracovanou správu uživatelů, která umožňuje v systému pracovat více uživatelům najednou (tj. ve stejný okamžik) bez vzájemného ovlivňování, uživatelé se mohou přihlašovat na terminálech připojených k počítači nebo v případě serveru po síti. Tyto systémy především musí zajistit přísné oddělení prostředků (např. paměti) využívaných různými uživateli.

Podle počtu provozovaných programů (podrobněji viz kap. 4.2) na

- *jednoprogramové* (monoprogramové) – v jednom okamžiku může být spuštěn jen jeden program,
- *víceprogramové* (multiprogramové) – v jednom okamžiku může být spuštěno i více programů, dále zde odlišujeme podskupinu *víceúlohové* (multitaskové) systémy, které umožňují kromě toho i sdílení prostředků mezi procesy těchto programů (správa vnitřní paměti, přidělování tiskárny apod.).

  Víceprogramové systémy, které nejsou víceúlohové (tj. jsou *jednoúlohové*), řeší tento problém například odložením veškerého paměťového prostoru „odstaveného“ programu na vnější paměť nebo do chráněné části vnitřní paměti a následným obnovením stavu ve chvíli, kdy tento program má pokračovat ve své činnosti.

Podle schopnosti práce v síti na

- *lokální* – Windows s DOS jádrem, v síti typu klient-server mohou být jen klienty,
- *síťové* – unixové systémy a Windows s NT jádrem, kromě klientské verze mají také serverovou verzi.

Podle míry specializace na

- *speciální* – jsou specializované na jeden typ (nebo několik málo typů) úloh,
- *univerzální* – běžné operační systémy na PC, řeší různé typy úloh.

Rozlišujeme také podskupiny operačních systémů – realtimové a distribuované.

1.4 Realtimové operační systémy

Realtimové operační systémy jsou operační systémy pracující v reálném čase. Používají se především tam, kde jsou vysoké požadavky na interaktivitu systému, zadávané úlohy musí být vyřízeny téměř okamžitě nebo ve vhodně krátkém čase. Jde například o systémy na řízení letadel, některých výrobních provozů, laboratoří, elektráren včetně atomových, v automobilovém průmyslu, atd.

Realtimový systém nemusí reagovat okamžitě, je pouze požadována „horní časová hranice“, tedy musí být zaručena maximální doba reakce v nejhorším možném případě. Běžné operační systémy s multitaskingem toto zaručit nemohou, zvláště pokud je spuštěno hodně procesů, třebaže obvykle nabízejí možnost přidělit procesu tzv. *realtimovou prioritu* výrazně vyšší než je priorita běžných procesů. Přesto jsou možnosti, jak tyto operační systémy upravit, aby pracovaly jako realtimové.

Realtimová priorita existuje i u klasických operačních systémů, ale narozdíl od realtimových zde nelze zaručit maximální dobu zpracování procesu, třebaže takový proces má přednost před procesy s jinými typy priorit.

Většina realtimových systémů má malé jádro (*mikrojádro*), které plní pouze nejdůležitější funkce (především správu procesů, případně správu paměti apod.), zbytek systému je implementován jako běžné procesy. Tento model odpovídá struktuře typu klient-server (viz kapitola 2). Pokud systém vznikl přepsáním z klasického systému, často jádro původního systému je mikrojádrem „odstaveno“ a běží pouze jako jeden z procesů (to je případ mnohých upravovaných unixových systémů).

Dále uvádíme jeden realtimový systém vzniklý podstatným upravením unixového systému, jeden vytvořený úpravou Linuxu a jednu možnost, jak přidat podporu reálného času do Windows s NT jádrem.

QNX je realtimový systém postavený na hodně upraveném unixovém klonu. Má malé mikrojádro a několik nejdůležitějších serverů (správa procesů, správa paměti apod.), zbytek systému běží jako běžné procesy. Vyznačuje se mimořádnou stabilitou a rychlostí, a to i při práci v grafickém rozhraní. Běží výborně i na slabších počítačích. Vyznačuje se výbornou podporou sítě, dá se také použít pro přístup na internet v případě, že pevný disk je z nějakého důvodu nedostupný. Je kompatibilní s normou POSIX.

Je to původně komerční systém, ale ke stažení je také několik různě rozsáhlých nekomerčních verzí (OpenQNX). Nevýhodou je nedostatek aplikací pro tento systém, ale vzhledem k tomu, že jde vlastně o unixový systém, není takový problém portovat na QNX unixové aplikace (na internetu včetně stránek výrobce tohoto systému je k nalezení mnoho aplikací takto upravených pro QNX).

RTLinux je upravený Linux. Má realtimové mikrojádro, samotné linuxové jádro běží jako samostatný proces s nižší prioritou. Systém byl vytvářen tak, aby zásahů do původního Linuxu bylo co nejméně. Zpracování přerušení[1] (tedy požadavků, které by případně mohly být i realtimové) probíhá tak, že nejdřív je přerušení zachyceno mikrojádrem, a teprve tehdy, když čas procesoru nevyžaduje žádný realtimový proces, je přerušení předáno původnímu linuxovému jádru, které je již zpracuje klasickým způsobem. Tento systém je stejně jako většina jiných Linuxů volně ke stažení na internetu.

[1]Přerušení znamená přerušení normálního běhu programu. Může to být například přerušení generované klávesnicí (po stisku klávesy se program o tom musí dovědět a vhodně reagovat).

RTX (RealTime eXtension) je modul, který rozšiřuje možnosti Windows s NT jádrem (NT/2000/XP) směrem k realtimovým systémům. Nejde tedy o realtimový systém, pouze o nástavbu pro operační systém klasického typu.

K systému je přidáno zvláštní rozšíření vrstvy HAL (RTX Real-time HAL Extender), nad kterým běží nový subsystém reálného času (RTX RTSS), v tom pracují procesy čistě real-timové. S tímto subsystémem komunikuje RTX ovladač, který umožňuje běžet také Win32 procesům s podporou pro RTX (real-timovým procesům využívajícím také prostředky Windows). Další informace viz stránky firmy Microsoft, ve vyhledávání zadejte řetězec `rtx real-time`.

1.5 Distribuované operační systémy

Distribuovaný systém[2] (nejen operační) je systém splňující tyto podmínky:

- pracuje na více než jednom procesoru (může to být i vhodně navržená a spravovaná počítačová síť),
- má svůj program rozdělen na (samostatné) části, které vzájemně komunikují,
- každá taková část je (může být) zpracovávána na jiném procesoru se zajištěním co největší transparentnosti.

Distribuovanost tedy znamená možnost co nejvíce rozložit výpočet v systému na více míst, která pracují paralelně. Rozlišujeme dva druhy distribuovanosti:

distribuovanost s hrubou granularitou – části systému jsou spíše větší, samostatnější, méně mezi sebou komunikují, použitelné v případě, že je problém zajistit dobrou a rychlou komunikaci (horší propojení počítačů - procesorů v systému),

distribuovanost s jemnou granularitou – části systému jsou co nejmenší, hodně mezi sebou komunikují.

Rozlišujeme dva druhy distribuovaných systémů – distribuované aplikace a distribuované operační systémy.

Distribuovaná aplikace je distribuovaný systém běžící na více propojených počítačích, každý z počítačů má svůj vlastní operační systém. Tato síť počítačů může být i Internet.

[2] Můžeme také najít název *grid*.

Jednou z nejznámějších aplikací tohoto druhu je BOINC (zkratka pro Berkeley Open Infrastructure for Network Computing[3]) umožňující kterémukoliv uživateli počítače připojenému k Internetu propůjčovat výpočetní kapacitu svého počítače některému z projektů využívajících tuto aplikaci. Jsou to například projekty Climateprediction.net (celosvětová předpověď počasí), SETI@home (analýza rádiových signálů potenciálně přicházejících od mimozemských civilizací), Einstein@home (hledání gravitačních vln generovaných pulsary), několik projektů z biomedicíny (buňky, proteiny apod.), atd.

Grid je možné vytvořit i doma, existují nástroje pro vytváření gridů v malé domácí síti (používají se pro časově složité operace jako je například dlouhodobé překládání softwaru ze zdrojových kódů – Gentoo Linux, zpracovávání multimédií apod.).

V souvislosti s Linuxem je třeba zmínit také *distribuované systémy pro správu verzí*. Systém pro správu verzí umožňuje skupině programátorů dostatečně efektivně pracovat na tomtéž projektu. Jde především o synchronizaci přístupů a změn v zdrojových kódech, systém u každého registrovaného souboru uchovává historii změn, několik posledních verzí, informace (metadata) o souborech a jejich autorech, a také stanoveným způsobem reaguje v případě, že více uživatelů systému chce měnit tentýž soubor – buď první přistupující soubor uzamkne nebo se provádí tzv. „slučování změn“. Stav projektu je veden ve větvích, slučování změn může odpovídat právě slučování těchto větví.

Na linuxových programech a také na Linuxu samotném pracují poměrně rozsáhlé skupiny programátorů fyzicky se nacházejících v různých částech světa. Proto je často potřeba pro dostatečně rychlou synchronizaci jejich práce používat systém pro správu verzí, který je distribuovaný (pro menší projekty je však tato vlastnost zbytečná). Donedávna vývojáři Linuxu používali systém BitKeeper, ale především z licenčních důvodů se přechází na nový systém Git vytvořený samotným tvůrcem Linuxu Linusem Torvaldsem. Git není plnohodnotný systém pro správu verzí, i když pro tyto účely dostačuje (je to také distribuovaný systém). Prosazuje se jeho varianta rozšířená o další skripty, Cogito, která je již plnohodnotným systémem pro správu verzí (autorem je Petr Baudiš).

Distribuovaný operační systém je samostatný operační systém běžící na síti procesorů, které nesdílejí společnou paměť, a zároveň poskytuje uživateli dojem jednoho počítače.

[3] `http://boinc.berkeley.edu`

Třebaže je fyzicky rozmístěn na různých počítačích, nemá (nemělo by) to mít vliv na jeho činnost a uživatel neurčuje, kde se konkrétně jeho data zpracovávají nebo kde ve skutečnosti jsou uložena. Dále se již budeme věnovat pouze distribuovaným operačním systémům.

Základní vlastnosti distribuovaného operačního systému jsou:

1. transparentnost („průhlednost" – strukturu či postup není vidět),
2. flexibilita (přizpůsobivost),
3. rozšiřitelnost.

Transparentnost je nejdůležitější vlastností distribuovaného operačního systému, znamená pro uživatele a případně i pro procesy určitý dojem jednolitosti systému. Tato vlastnost se týká především vztahu procesů a prostředků celého systému.

Vyžaduje se, aby proces jednotným způsobem přistupoval k lokálním i vzdáleným prostředkům (přístupová transparentnost), a zároveň aby nemusel znát fyzické umístění tohoto prostředku (tj. při konkretizaci prostředku, ke kterému chce přistupovat, neudává jeho umístění - adresu, ale identifikuje ho jiným způsobem, to se nazývá lokační transparentnost), prostředky mohou být libovolně přesouvány a připojovány k různým částem celého systému bez ovlivnění činnosti procesů (migrační transparentnost), procesy mohou běžet na kterémkoliv procesoru a dokonce mohou být při svém běhu přemístěny na jiný procesor, aby se vhodně vyrovnala zátěž různých částí systému (exekuční transparentnost), atd.

Flexibilita znamená schopnost systému přizpůsobovat se veškerým změnám prostředí, ve kterém pracuje, včetně různých poruch a výpadků částí systému. Souvisí také s vlastností migrační transparentnosti.

Aby systém dosáhl dostatečné flexibility, je vhodné, aby každá část systému byla pokud možno co nejvíce samostatná ve své práci, centrální rozhodování může tuto vlastnost narušit. V dostatečně flexibilním systému je možné přemísťovat provádění procesů na ty procesory, které zrovna nejsou vytížené, odlehčovat příliš vytíženým procesorům, a totéž platí i o přemísťování prostředků mezi částmi systému.

Rozšiřitelnost souvisí s flexibilitou. Distribuovaný operační systém by měl být schopen rozšíření o (teoreticky) jakékoliv množství procesorů. Prakticky je samozřejmě toto množství limitováno především problémy při komunikaci. Nejde jen o propustnost linek, která může komunikaci zdržovat nebo komplikovat, ale také o náročnost synchronizace systému, kde se z důvodu distribuovanosti odbourává centralizované řízení čehokoliv. Proto se velmi rozsáhlé distribuované systémy budují především v oblastech, kde tyto problémy nejsou podstatné nebo je lze řešit.

KAPITOLA 2

Struktura operačních systémů

Abychom mohli porozumět tomu, jak pracují operační systémy, potřebujeme alespoň základní informace o jejich struktuře. U moderních operačních systémů je struktura vytvářena především s ohledem na bezpečnost a stabilitu celého systému, vždy najdeme rozdělení na privilegovanou část (privilegovaný režim, režim jádra) a uživatelskou část (uživatelský, neprivilegovaný režim) s tím, že procesy běžící v uživatelské části nemají možnost jakkoliv zasahovat do privilegované. Ovšem svou strukturu mají také jednodušší systémy, jim často stačí jednodušší stavba.

V této kapitole nejdřív probereme základní druhy struktur, pak si ukážeme strukturu některých konkrétních operačních systémů rodiny Windows a Unix.

2.1 Základní typy struktur operačních systémů

Monolitická struktura je nejjednodušší struktura používaná v jádrech některých operačních systémů nebo v zařízeních (tiskárny). Systém se skládá z jádra a rozhraní, které zprostředkovává komunikaci mezi jádrem a okolím.

Vrstvená (hierarchická) struktura – části systému jsou uspořádány do vrstev, každá vrstva využívá služeb nižších vrstev, ne naopak. Systém je budován od vnitřních vrstev k vnějším, proto vnitřní vrstvy, které jsou obvykle nejdůležitější z hlediska stability a bezpečnosti, bývají nejlépe otestovány. Tento typ struktury je u moderních operačních systémů nejběžnější.

Virtuální počítače (virtuální stroje) – systém je rozdělen do samostatných modulů (virtuálních počítačů, virtuálních zařízení), každý z nich je stejně vybaven pro-

středky (čas procesoru, paměť, apod.), obvykle se nemohou příliš vzájemně ovlivňovat kromě základní komunikace mezi procesy (např. předávání dat a jiných informací). Používá se v operačních systémech pro podsystémy, které je nutné z nějakého důvodu oddělit vzájemně nebo od prostředků systému.

Abstraktní počítače – systém je rozdělen do modulů, ale narozdíl od virtuálních počítačů abstraktní počítače mají každý svou specifickou funkci (např. modul, který zprostředkovává přístup k tiskárně, udržuje tiskovou frontu, snímá z ostatních procesů nutnost během tisku neustále komunikovat s tiskárnou a posílat jí data). Typické použití je v primárním rozhraní zařízení – ovladače.

Model klient-server – systém má co nejmenší jádro (minikernel, mikrokernel), které obsahuje pouze základní funkce (obvykle pouze funkce řídící činnost ostatních částí systému, jako je přepínání mezi procesy a řízení mechanismu zasílání zpráv mezi procesy), ostatní funkce systému provádějí speciální systémové procesy, které nazýváme *servery*. Procesy, které spouští uživatel (nejsou systémové), se nazývají *klienty*, využívají služeb procesů typu server. Výhodou je vyšší stabilita systému – pokud chyba nastane u některého serveru, může být resetován, ale nemusí být znovu zaváděn celý systém (pravděpodobnost poškození jádra je malá vzhledem k jeho jednoduchosti). Tuto strukturu využívá mnoho realtimových systémů.

Stavebnicová struktura – opět máme co nejmenší jádro, zbytek je přilinkován jen tehdy, když je vyžadován některou klientskou aplikací. Tento typ struktury je výhodný, pokud kromě jiného vyžadujeme také minimálnost a efektivitu běhu (příliš mnoho přilinkovaných modulů, které jsou navíc zbytečné, by zdržovalo a zabíralo místo v paměti či jiné zdroje). Stavebnicovou strukturu mají některé realtimové systémy.

2.2 MS-DOS a Windows s DOS jádrem

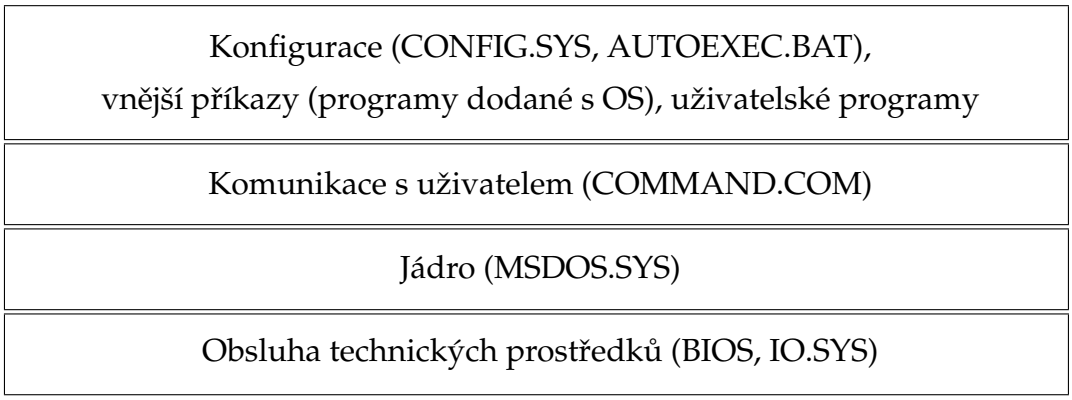
Řada Windows s DOS jádrem zahrnuje Windows 95, 95 OSR2, 98, 98 SE a ME. Tyto verze vznikly úpravou a včleněním původně samostatného operačního systému MS-DOS jako jádra do Windows, které se tímto staly samostatným operačním systémem[1].

[1]Windows do verze 3.11 byly pouze grafickou nástavbou MS-DOSu, nikoliv samostatným operačním systémem. Tím se staly právě až od Windows 95, vnitřně Windows 4.0.

Struktura operačních systémů Windows s DOS jádrem vychází z původního systému MS-DOS, proto se nejdřív podíváme na strukturu tohoto jednoduchého systému a pak ji rozšíříme na tuto řadu Windows.

MS-DOS je jednoprocesorový jednouživatelský jednoprogramový lokální univerzální systém. Samotný MS-DOS bez spuštěné nástavby Windows má velmi jednoduchou vrstvenou strukturu. Nejblíže hardwaru je BIOS (Basic Input-Output System) a dále soubor IO.sys, který se stará o obsluhu periferií.

BIOS poskytuje programátorům základní ovládání hardwaru (např. klávesnice, monitoru) přes hardwarová a softwarová přerušení[2]. Pokud programátor potřebuje komunikovat s určitým zařízením (třeba vypsat či vykreslit něco na obrazovku), vyvolá příslušné přerušení (k tomu jsou v programovacích jazycích speciální příkazy), případně se může napojit na některé přerušení a nechat provést určitou funkci ve chvíli, kdy je přerušení vyvoláno jinde než v programu (například takto hlídá stisknutí kláves nebo pohyb myši).



Obrázek 2.1: Struktura OS MS-DOS 6.22

Nad vrstvou pro ovládání hardwaru je vrstva samotného jádra systému představovaná souborem MSDOS.SYS. Tento systém má tedy monolitické jádro složené z jediného souboru. Jádro poskytuje další softwarová přerušení, například pro přístup k souborům nebo pokročilejší práci s grafikou.

Následující vrstva tvořená souborem COMMAND.COM je textové rozhraní mezi uživatelem a systémem. Tento program je spuštěn po celou dobu práce systému a komunikuje s uživatelem (spuštěné programy komunikují s nižšími vrst-

[2]Přerušení znamená přerušení normálního běhu programu. Může to být například přerušení generované klávesnicí (po stisku klávesy se program o tom musí dovědět a vhodně reagovat).

vami, což uživatel nedovede, potřebuje „překladatele“). Uživatel zadává příkazy a rozhraní na ně reaguje a vypisuje výsledky či chybová hlášení. Samotný COMMAND.COM obsahuje sadu *vnitřních příkazů*. Ostatní příkazy se nazývají *vnější příkazy* a jsou implementovány jako krátké jednoduché programy s příponou EXE nebo COM.

Poslední vrstva je určena k „zjednodušení práce“ uživatele. Kromě uživatelem spuštěných programů zde běží také programy představující vnější příkazy a řadíme zde také konfigurační soubory, ve kterých si uživatel může určit, jak má systém reagovat. Základní konfigurační soubory jsou dva – CONFIG.SYS pro nastavení hardwaru (například spuštění určitých ovladačů pro monitor s určením znakové sady pro češtinu) a AUTOEXEC.BAT pro nastavení softwaru (zde například určujeme, které programy nebo příkazy se mají spustit po startu systému).

Když v MS-DOSu 6.22 spustíme Windows 3.x v rozšířeném módu[3], struktura celého systému se v horní části změní. Na obrázku 2.2 je spodní část trochu shrnuta (BIOS, MSDOS.SYS). K nim je přidán soubor WIN.COM, který slouží ke spuštění celých Windows (je také ve všech Windows s DOS jádrem), a dále řadiče (ovladače). Windows přidávají multitasking, 16-bitové knihovny (MS-DOS je 8-bitový systém) a ve verzi 3.11 for Workgroups základní podporu sítě (pouze síť peer-to-peer).

Řadiče (ovladače, drivery) ovládají periferní zařízení pro Windows; řadiče přímo pro Windows jsou spouštěny v souboru SYSTEM.INI pomocí příkazu DEVICE (některé, které můžeme využívat i v DOSu, v souboru CONFIG.SYS).

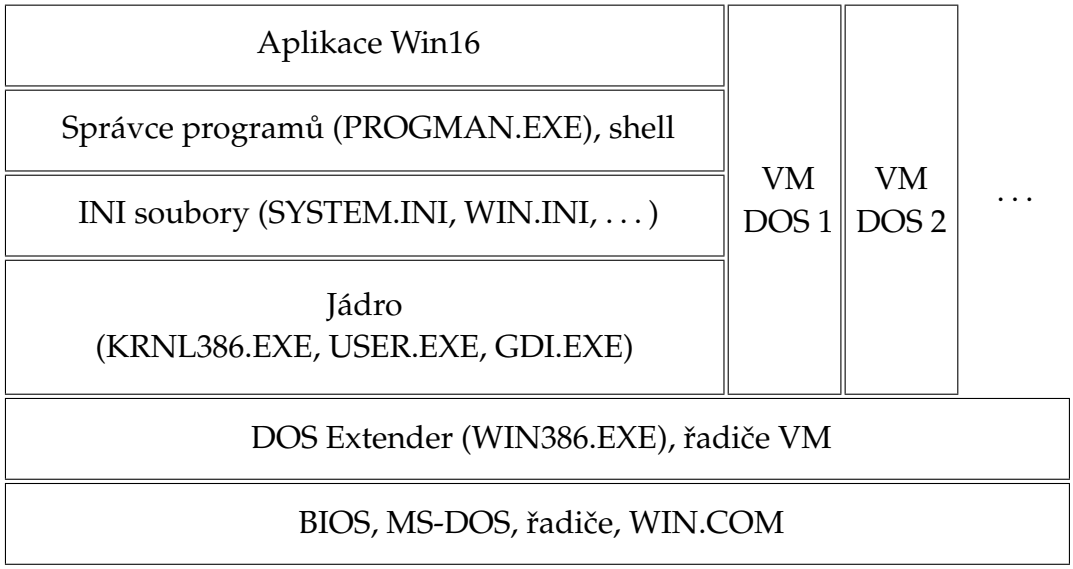
DOS Extender je modul pro podporu využití rozšířené paměti (Extended Memory). Je představován souborem Win386.EXE.

Součástí tohoto souboru je také *Správce virtuálních zařízení* (VMM = Virtual Machine Manager), který ovládá možnosti Windows pro souběh s programy DOSu. *Řadiče virtuálních zařízení* (VxD) jsou řadiče, které správce virtuálních zařízení potřebuje pro manipulaci s I/O zařízeními pro programy DOSu v rozšířeném módu.

V další vrstvě je jádro Windows (pozor, jádrem operačního systému stále zůstává MSDOS.SYS), které zde pracuje jako správce prostředků vzhledem k programům běžícím pod Windows (i DOS programům zde spuštěným). Skládá se ze tří částí – souborů:

- KRNL386.EXE – plní především úlohu správce paměti a správce procesů (řízení přidělování paměti procesům, přidělování prostředků systému procesům, . . .),

[3]MS-DOS pracuje v reálném módu, kde lze paměť využívat pouze do 1 MB. Windows 3.x, aby byly práceschopné, se zapínají v rozšířeném módu, dostupném až od procesorů i386, kde kromě rozšířené paměti také například mohou využívat chráněný mód procesoru pro ochranu paměti.



Obrázek 2.2: Struktura OS MS-DOS + Windows 3.x v rozšířeném módu

- GDI.EXE – rozhraní grafického zařízení (obsahuje funkce pro vytváření kurzoru, ikony, písmo, . . .),
- USER.EXE – uživatelské ovládání rozhraní, zdroje, které nepatří do GDI (dialogová okna, menu, okna, . . .).

V další vrstvě najdeme konfigurační soubory s příponou INI. Z nich jsou nejdůležitější WIN.INI (konfigurace software a nastavení pro urč. uživatele) a SYSTEM.INI (konfigurace hardware). INI soubory mohou mít také různé programy.

Následuje vrstva, která rozhraním mezi uživatelem, programy a samotným systémem. Soubor PROGMAN.EXE je *Správce programů*, shell je grafické a textové rozhraní mezi uživatelem a systémem.

Zde bychom také mohli zařadit část *API rozhraní* (Application Programming Interface) reprezentovaného *dynamicky linkovanými knihovnami* (obsahují funkce, objekty apod.) a využívaného procesy pro přístup k systému. Knihovny celého API rozhraní jsou však rozmístěny v různých vrstvách, patří zde také soubory jádra KRNL386.EXE, USER.EXE a GDI.EXE. Většina knihoven má příponu DLL, ale některé mají příponu EXE, přípona knihoven fontů zase závisí na typu fontu (například TTF pro True-type fonty), atd.

Všechny dosud uvedené vrstvy platí zejména pro aplikace psané pro Windows (16-bitové aplikace pro Windows do verze 3.x), DOS programy ani netuší o existenci jádra Windows a INI souborů, proto horní vrstvy vůbec nevyužívají. Protože však je samotný MS-DOS jednoprogramový systém (kromě ovladačů a rezidentních programů je spuštěn vždy jen jeden program), tyto programy jsou napsány bez jakýchkoliv ohledů na možnost sdílení paměti s jinými procesy. Proto je nutné „separovat" je do virtuálních počítačů, které programům vytvoří iluzi výlučné existence v systému a znemožní jim zásahy do prostředků jiných procesů.

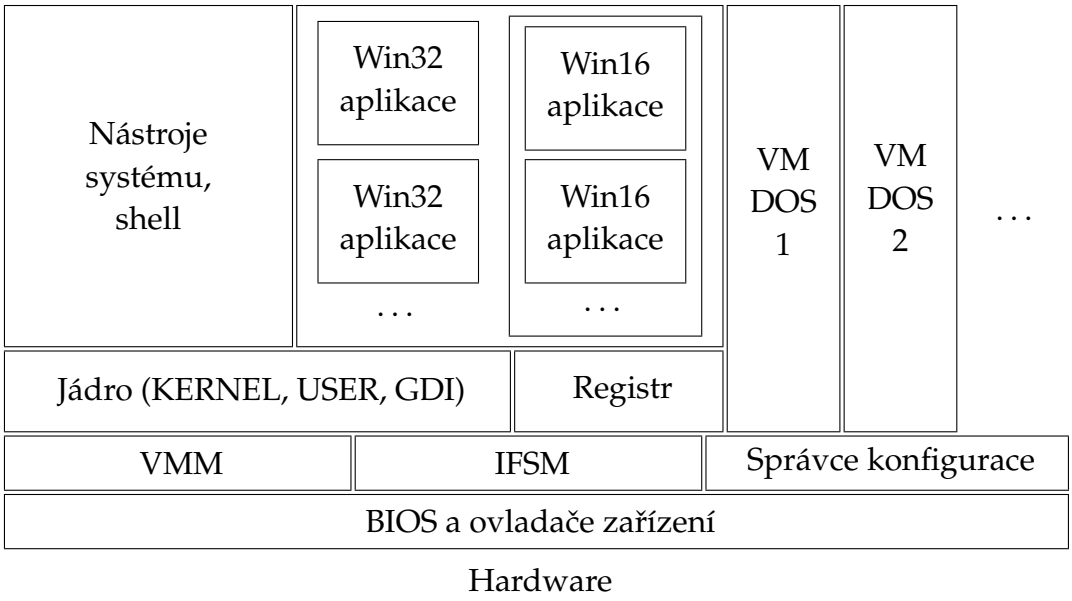
Jak již bylo uvedeno, od verze 4.x (95) jsou Windows již operační systém se samostatným jádrem. Oproti sestavě MS-DOS+Windows 3.x je to 32-bitový systém (ale některé knihovny zůstávají 16-bitové), ostatní vlastnosti zůstávají. Na obrázku 2.3 na str. 16 je naznačena zjednodušená struktura tohoto systému.

Spodní vrstva opět slouží k přístupu systému k zařízení. Následující vrstva se také vztahuje k hardwaru, ale již na abstraktnější úrovni. Skládá se ze tří základních modulů:

- *VMM* je správce virtuálních zařízení (Virtual Machine Manager), vytváří a udržuje prostředí virtuálních zařízení,
- *IFSM* je správce instalovatelných souborových systémů (Installable File Systems Manager), spravuje různé typy souborových systémů, které lze instalovat, např. FAT16, VFAT (FAT32 s rozšířeními), CDFS (systém souborů pro CD-ROM), . . . ,
- *Správce konfigurace* spravuje ovladače hardware na vyšší úrovni, především zařízení typu Plug&Play.

Jádro se skládá ze tří modulů, každý z nich má dvě dynamicky linkované knihovny (jedna pro 16-bitové aplikace s příponou EXE, druhá pro 32-bitové aplikace s příponou DLL):

- *KERNEL* – multithreading, multitasking, správa paměti, synchronizace objektů, vstupu a výstupu u souborů, . . . ,
- *GDI* (Graphics Device Interface) – správce tisku, spooler (viz přednášky), zpracování grafiky, základní grafické objekty, . . . (funkce podobná jako u Win 3.x),
- *USER* – vstupy z klávesnice, myši apod. (řízené přerušeními), výstupy do uživatelského grafického rozhraní (okna, menu, ikony, . . .), práce s časovačem, . . .



Obrázek 2.3: Struktura OS Windows 9x/ME

Registr je centrální informační databáze systému, najdeme zde většinu toho, co ve Windows 3.x bylo v INI souborech (ty jsou však zachovány kvůli zpětné kompatibilitě). Fyzicky je uložen v souborech SYSTEM.DAT a USER.DAT (ve Windows 9x/ME).

Aplikace Win32 (tj. psané pro Windows od verze 95 výše) a Win16 (pro nižší verze) běží v systémovém virtuálním počítači, každá Win32 aplikace má vyhrazen svůj vlastní adresový prostor, aplikace Win16 mají jeden společný adresový prostor[4]. DOS aplikace mají stejně jako ve Windows 3.x každá svůj virtuální počítač (a proto také každá svůj vlastní adresový prostor, v rámci virtuálního počítače).

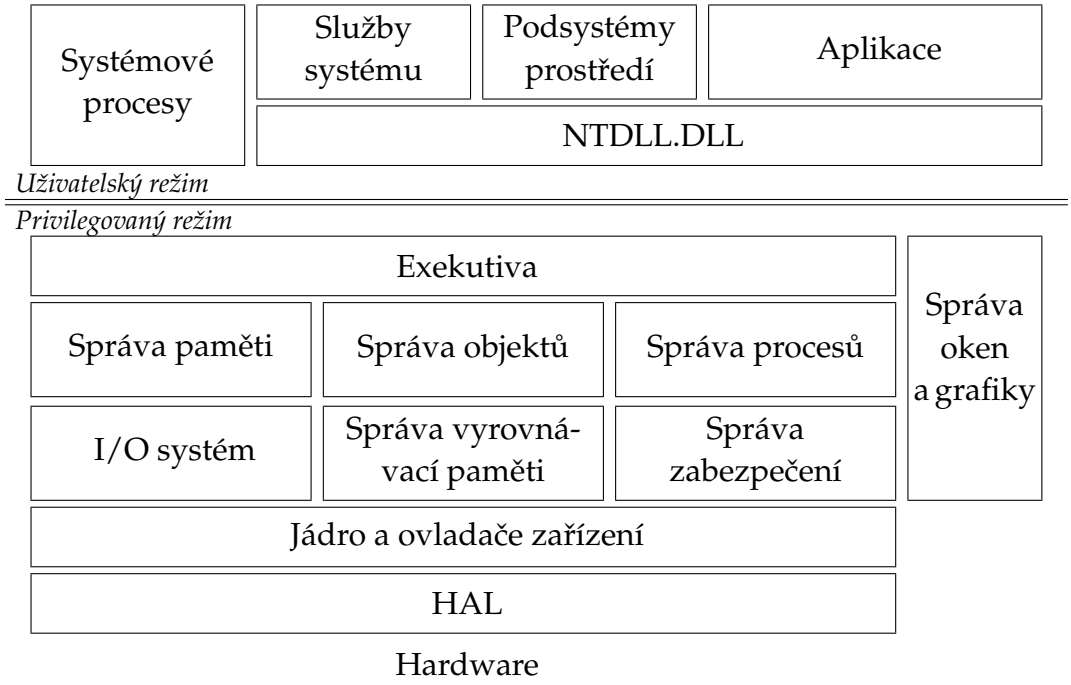
2.3 Windows řady NT

Jádro systému Windows řady NT vznikalo nezávisle na systému MS-DOS, už při jeho návrhu bylo bráno v úvahu typické použití tohoto systému jako serveru nebo klienta v síti, a tudíž hlavním hlediskem je stabilita a možnosti zabezpečení. Systém byl navržen jako víceprocesorový (SMP – symetrický multiprocessing) víceu-

[4]To je kromě jiného proto, že Win16 aplikace byly psány pro systém, kde procesy mohou spolu komunikovat přes společnou paměť v DLL knihovnách, u Win32 aplikací a 32-bitových knihoven to již není možné, mohl být proto zvolen model více vyhovující požadavkům stability a bezpečnosti systému.

živatelský multitaskový univerzální síťový systém. Následující struktura platí pro Windows NT 4.x, Windows 2000 a XP, ale v hlavních rysech platí i pro předchozí verze.

Zjednodušená struktura systému je naznačena na obr. 2.4. Některé prvky jsou podobné částem struktury Windows s DOS jádrem, ale vnitřně pracují jinak. Důležité je především rozdělení do dvou základních částí – části běžící v privilegovaném režimu (režimu jádra) a části běžící v uživatelském režimu.



Obrázek 2.4: Struktura OS Windows s jádrem NT

HAL je vrstva abstrakce hardware (Hardware Abstraction Layer), rozhraní mezi hardwarem a zbytkem jádra systému. Je řízena souborem HAL.DLL. Je oddělena od ostatních částí systému z důvodu snadnější přenositelnosti systému. Ovladače komunikují se zařízeními pouze zprostředkovaně přes tuto vrstvu.

Jádro je součástí souboru NTOSKRNL.EXE (zároveň s exekutivou). Zde jsou obsluhována přerušení, provádí se správa procesorů (synchronizace přidělování procesorů), apod.

Exekutiva je řídicí program operačního systému, má na startosti řízení celého jádra běžícího v privilegovaném režimu. Fyzicky je zároveň s jádrem součástí souboru NTOSKRNL.EXE.

Podsystémy (subsystémy) prostředí jsou rozhraní zajišťující správný a bezpečný běh různých typů aplikací. V těchto podsystémech běží aplikace, které pak nemusí být kompatibilní s Windows NT. Podsystémy poskytují aplikacím rozhraní, které překládá komunikaci (požadavky na informace, zdroje, provedení určité akce apod.) mezi aplikací a operačním systémem tak, aby si obě strany „rozumněly". Jsou zde například podsystémy pro aplikace psané pro 32-bitová Windows, MS-DOS a aplikace pro 16-bitová Windows (VDM – Virtual DOS Machine), OS/2, POSIX[5], atd.

Podsystém pro 32-bitová Windows včetně NT (podsystém Win32) je představován souborem CSRSS.EXE, pro POSIX je to především soubor PXSS.EXE (to je server podsystému). Podsystém Win32 je potřebný pro běh operačního systému, proto se jako jediný spouští hned po startu počítače, ostatní podsystémy jsou spouštěny až na žádost při spuštění aplikace patřící tomuto podsystému. Každý podsystém má kromě svého řídicího programu (například CSRSS.EXE u Win32) také knihovny, ve kterých jsou uloženy funkce a objekty, obsahují API (Application Programming Interface) daného podsystému. Například ke knihovnám Win32 patří také knihovny KERNEL32.DLL, USER32.DLL a GDI32.DLL. Jejich funkce je podobná jako v jiných variantách Windows, vnitřně však mají odlišnou strukturu.

Soubor NTDLL.DLL představuje rozhraní mezi běžnými procesy a systémem. Pokud některý proces v uživatelském režimu volá službu běžící v privilegovaném režimu, volání jde vždy přes tento soubor, aby se vyloučila možnost modifikace systémových knihoven a dalších systémových zdrojů. NTDLL představuje tzv. *dokumentované rozhraní* systému, které každému procesu zprostředkovává komunikaci s daným podsystémem.

Systémové procesy jsou procesy, které spouští systém, například procesy zajišťující uživatelské prostředí. V zobrazení Správce úloh, karta Procesy, je při zapnutém zobrazování uživatelů procesu poznáme podle hodnoty SYSTEM, LOCAL SERVICE, NETWORK SERVICE apod. Pracují v uživatelském režimu z důvodu bezpečnosti, k částem systému pracujícím v privilegovaném režimu však mají trochu jednodušší přístup než ostatní procesy.

Služby systému (serverové procesy) jsou služby poskytované systémem, seznam lze najít např. v nástroji Nástroje pro správu – Služby. Služby jsou systémové procesy běžící často i bez přihlášení uživatele, obdoba rezidentních programů v MS-DOSu. Běh služeb zajišťuje proces řadiče služeb představovaný souborem SERVICES.EXE.

[5]POSIX (Portable Operating System Interface for Unix) je standard navržený původně pro Unix, a to tak, aby programy napsané podle tohoto standardu bylo možné snadno přenášet mezi různými operačními systémy (různé varianty Unixu od různých firem mohou být vzájemně nekompatibilní).

Modul pro *správu oken a grafiky* (GUI, Win32 User a GDI) je kód uživatelského rozhraní a rozhraní grafických zařízení pro podsystém Win32. Ve Windows řady NT od verze 4 byla tato část kódu přesunuta do režimu jádra z důvodu urychlení práce aplikací hodně využívajících grafická zařízení. Tento modul je určen pro podsystém Win32, ale aby nebylo nutné tyto funkce implementovat v každém podsystému zvlášť, je překládáno volání grafických funkcí jiných podsystémů na volání v podsystému Win32.

Umístění kódu grafického rozhraní do režimu jádra je neobvyklé. Nevýhodou tohoto postupu je ovšem větší bezpečnostní riziko a riziko porušení stability systému při chybné práci tohoto modulu (pracuje v režimu jádra, proto má přístup do praměti systémových procesů). Další nevýhodou je náročnější postup výměny uživatelského rozhraní za alternativní.

Jak je vidět na obr. 2.4, Windows řady NT nejsou přísně vrstvený systém, ale kombinují více různých architektur pro své různé části. Jsou to tyto architektury:

1. Vrstvená architektura se uplatňuje v jádře, vrstvě HAL a I/O systému.

2. Modulární architektura – uzavřené moduly, vnitřně kompaktní, které poskytují služby přes nadefinované rozhraní, komunikace probíhá volně mezi různými moduly, tuto architekturu zde používá exekutiva při řízení správce procesů, správce paměti, I/O systému atd. (modulů běžících v privilegovaném režimu).

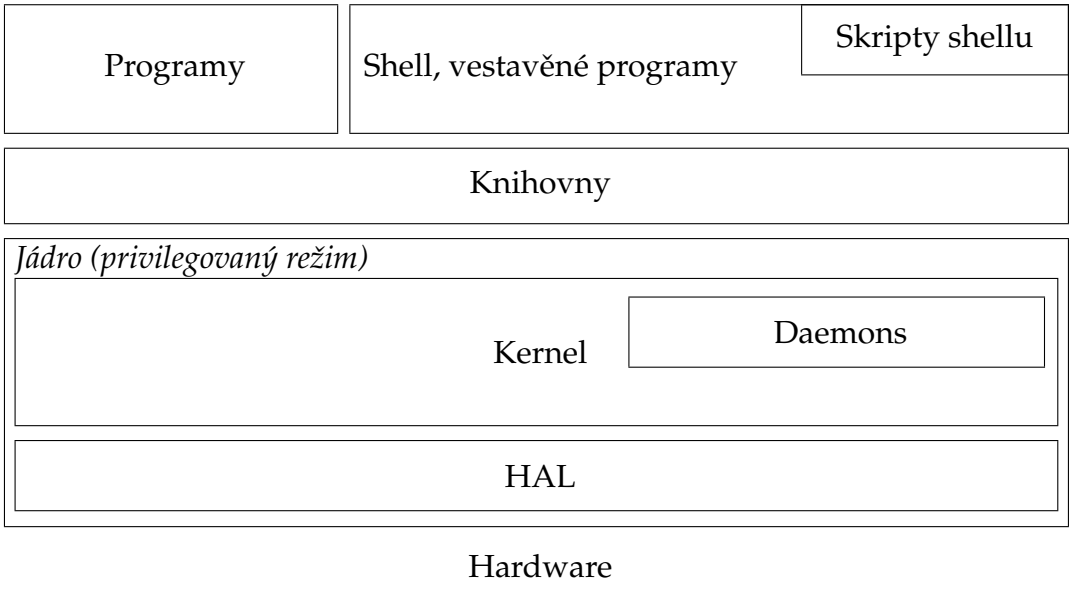
3. Architektura klient-server se uplatňuje v API (Application Programming Interface), což je sada dynamicky linkovaných knihoven, zde považovaných za servery, procesy z vyšších vrstev (klienti) využívají jejich služeb (přes knihovnu NTDLL.DLL).

2.4 Systémy Unixového typu

Většina Unixových systémů má hodně podobnou strukturu (kromě těch, které byly upraveny pro real-time provoz). Jádro běží v privilegovaném režimu (režimu jádra), je často tvořeno jediným souborem (proto je nazýváno monolitické, i když jeho vnitřní struktura je přesně rozvržena, například u linuxu je to soubor /boot/vmlinuz).

Unixové systémy jsou víceprocesorové víceuživatelské multitaskové univerzální síťové systémy již od svého počátku. Na to byl brán zřetel už při navrhování

jejich struktury, proto za dlouhá desetiletí existence Unixů ani nebylo třeba ji výrazněji měnit. Tato struktura také zřejmě posloužila jako jeden ze vzorů při návrhu struktury Windows řady NT.



Obrázek 2.5: Struktura operačních systémů Unixového typu

Jádro systému běží v privilegovaném režimu (kernel mode, vše ostatní běží v uživatelském režimu – user mode) a skládá se ze dvou oddělených částí:

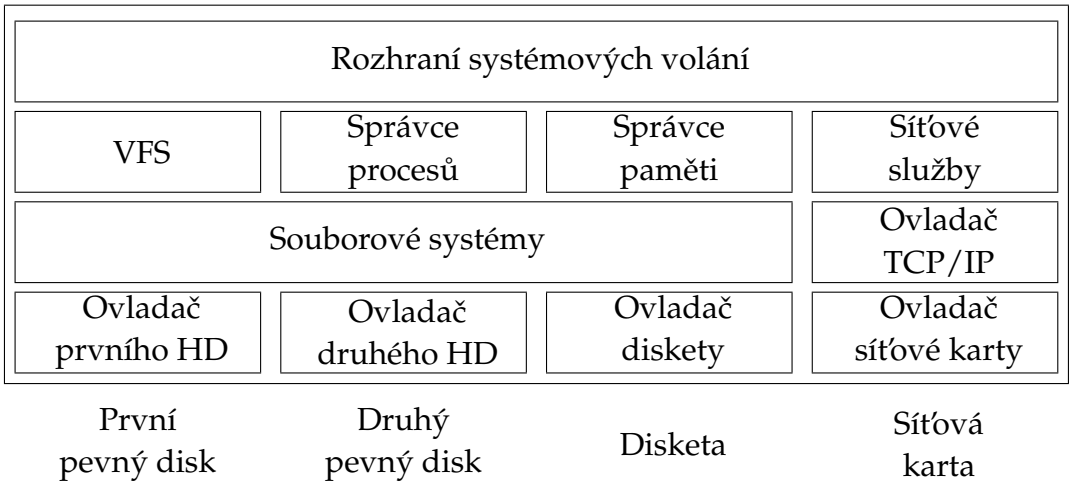
- *HAL* (Hardware Abstraction Layer) je část jádra závislá na hardware, jsou zde především ovladače zařízení,
- *Kernel* je jádro nezávislé na hardware, běží zde také *démoni* (daemons) – systémové procesy obdobné službám ve Windows řady NT. Běží na pozadí často bez ohledu na běh uživatelských procesů a přihlašování či odhlašování uživatelů.

Knihovny v Unixu mají podobnou roli jako DLL ve Windows, tedy obsahují objekty a různé rutiny. Mnohé z nich jsou součástí grafického subsystému (např. X Window).

Shell je rozhraní pro komunikaci s uživatelem. Unixové systémy obvykle nabízejí více různých shellů. Komunikace probíhá v textové formě (uživatel zadává příkazy, systém reaguje textovými výpisy), ale současné Unixové systémy většinou mají také velmi propracované grafické rozhraní a běžný uživatel s textovým shellem

ani nemusí přijít do styku. Stejně jako ve Windows příkazy můžeme shrnovat do textových souborů, které se nazývají *skripty*.

Nyní se podrobněji podíváme na strukturu jádra. Vrstva HAL je tvořena především ovladači periferních zařízení. Následuje vrstva, která přistupuje k zařízením na abstraktnější úrovni – jsou zde ovladače souborových systémů na paměťových médiích a ovladač TCP/IP komunikující s ovladačem síťové karty.



Obrázek 2.6: Struktura jádra Unixových systémů

Souborový systém je rozhraní mezi ovladačem vnějšího paměťového média a vyššími vrstvami jádra. V Unixových systémech jsou souborové systémy implementovány přímo v jádru včetně podpory přidávání nových souborových systémů[6]. Protože v Unixech platí, že „všechno je soubor“, jsou zde nejen souborové systémy pro vnější paměťová média, ale i další, abstraktní, zprostředkující přístup k informacím o momentálním stavu systému, konfiguraci apod. (např. v Linuxech s jádrem 2.4 souborový systém /proc) nebo sdružující jiné souborové systémy či představující část jiného souborového systému (v Linuxech to může být např. /etc).

VFS (Virtual File System, virtuální souborový systém) představuje rozhraní pro podobný způsob přístupu k různým souborovým systémům. Všechny souborové systémy sdružuje v jediné „stromové“ struktuře. Pokud uživatel chce s konkrétním souborovým systémem pracovat, připojí ho na stanovené místo do této struktury a tím ho zpřístupní (připojování je obvykle automatizováno, uživatel se o ně nemusí starat). Důležitou funkcí VFS je zajištění stejného způsobu zacházení s daty, ať už

[6]Proto narozdíl od Windows si můžeme nainstalovat podporu téměř jakéhokoliv souborového systému včetně těch, které v době sestavování OS ještě neexistovaly, jen musíme mít instalační soubory a pak jádro znovu přeložit ze zdrojových kódů.

se nacházejí v jakémkoliv souborovém systému, tedy uživatel se nemusí starat o fyzické umístění souboru, atributy (např. nastavení času posledního přístupu k souboru), konvence pro práci se soubory (jak sdělit souborovému systému, že chci otevřít určitý soubor, apod.).

Rozhraní systémových volání je rozhraní mezi jádrem a čímkoliv, co může přímo ovlivnit uživatel (programy, příkazy shellu, skripty). S touto vrstvou lze komunikovat přes knihovny obsahující definice *API funkcí* (Application Programming Interface, ve Windows je obdoba tohoto rozhraní – WinAPI, Win32 API). Hlavní úlohou je zajištění bezpečnosti, znemožnění zásahu uživatele do jádra (zde se Microsoft inspiroval při programování knihovny NTDLL.DLL, která má podobnou funkci), znamená také zjednodušení práce programátorů, kteří se nemusejí zabývat jednotlivými systémovými voláními a mohou používat API funkce.

KAPITOLA 3

Správa paměti

Pod pojmem paměť budeme rozumět vnitřní (operační) paměť. V této kapitole probereme různé metody, které se používají při správě paměti, a ukážeme si, jak jsou implementovány v některých operačních systémech.

3.1 Modul správce paměti

Modul správce paměti má tyto funkce:

1. Udržuje informace o paměti (která část je volná, která část je přidělena, kterému procesu je přidělena, atd.).
2. Přiděluje paměť procesům na jejich žádost.
3. Paměť, kterou procesy uvolní, zařazuje k volné paměti.
4. Pokud je to nutné, odebírá paměť procesům.
5. Jestliže je možné detekovat případy, kdy proces ukončí svou činnost bez uvolnění paměti (například při chybě v programu nebo při násilném ukončení), pak modul tuto paměť uvolní sám a zařadí k volné paměti.
6. Pokud to dovoluje úroveň hardwarového vybavení (především procesor), může zajišťovat ochranu paměti, tedy nedovolí procesu přístup do paměťového prostoru jiného procesu nebo dokonce do paměťového prostoru operačního systému.

Fyzicky je operační paměť umístěna na základní desce, ale také na rozšiřujících kartách (deskách, adaptérech), například část operační paměti, kterou nazýváme *videopaměť*[1], se nachází na grafické kartě, ale přesto je součástí operační paměti a v některých operačních systémech mají procesy do této paměti přímý přístup (řídí tak přímo, co se má zobrazit).

Souhrn těchto umístění operační paměti v různých částech hardwaru výpočetního systému je třeba utřídit do posloupnosti, kde má každá část své jednoznačné umístění. Proces k paměti přistupuje přes adresy. Adresa místa v paměti je počet Bytů k tomuto místu od začátku této posloupnosti. První Byte má adresu 0 (před ním žádný Byte není), druhý Byte má adresu 1, atd. Takovou adresu nazýváme *absolutní adresa*.

Relativní adresa se nevztahuje k počátku paměti, ale k určité absolutní adrese (to bývá obvykle začátek paměťového bloku nebo adresového prostoru procesu) a je to tedy počet Bytů od této absolutní adresy.

Každý proces má přidělen paměťový prostor v rozsahu určitých adres, proto hovoříme o *adresovém prostoru*. Rozlišujeme

- *fyzický adresový prostor* – adresový prostor, který je fyzicky k dispozici ve výpočetním systému,
- *logický adresový prostor* – adresový prostor, který mají k dispozici procesy.

Logický adresový prostor může být menší nebo roven fyzickému, ale s rostoucími potřebami procesů nemusí pro jejich práci rozsah fyzického adresového prostoru dostačovat. Proto může být operační paměť „nastavována“ prostorem na vnějším paměťovém médiu (obvykle pevném disku), pak je logický adresový prostor větší než fyzický. Pokud je možné, aby logický adresový prostor byl větší než fyzický, pak hovoříme o *virtuálních metodách přidělování paměti*.

Část adresového prostoru obvykle zabírá samotný operační systém, jsou to obvykle adresy na začátku adresového prostoru. V případě, že operační systém používá metody ochrany paměti nebo alespoň rozlišuje procesy na běžící v privilegovaném režimu a běžící v uživatelském režimu, běžným procesům není dovolen přístup do této oblasti nebo je před nimi přímo skryta. Existuje více metod ochrany paměti, většinou ale vyžadují hardwarovou podporu (je dostupná prakticky na všech novějších procesorech, včetně Intel od 386).

[1] Velikost videopaměti záleží na potřebách grafické karty a především na zvoleném grafickém módu zahrnujícím rozlišení obrazovky, barevnou hloubku a případně počet videostránek („logických obrazovek“, které lze střídat a tak rychle měnit obraz na monitoru).

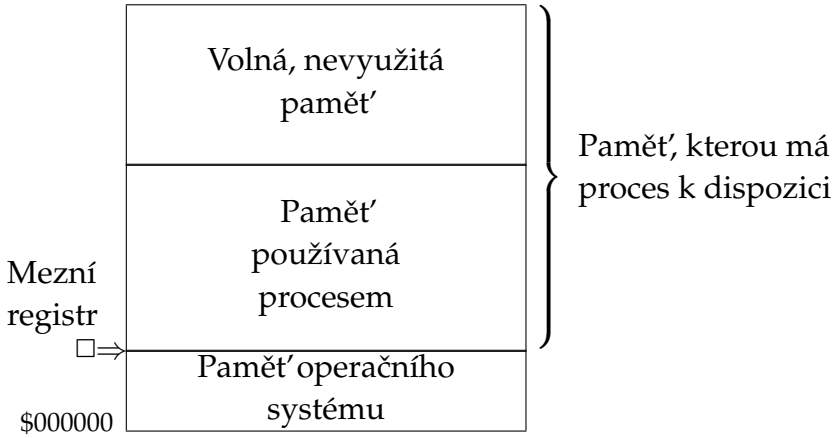
3.2 Reálné metody přidělování paměti

Zde probereme metody používané v případě, že logický adresový prostor nepřekračuje fyzický, tedy fyzická vnitřní paměť dostačuje potřebám procesů, a možnosti řešení problémů vznikajících při používání těchto metod.

3.2.1 Přidělení jedné souvislé oblasti paměti

Tato jednoduchá metoda spočívá v přidělení veškerého adresového prostoru procesu kromě oblasti operačního systému. Paměť je rozdělena na tři části: paměť vyhrazenou pro operační systém, paměť využívanou procesem a nevyužitou paměť. Tuto metodu používaly operační systémy, které nebyly multitaskové (například CP/M).

Pro ochranu paměti je vhodné alespoň používat mezní registr, ve kterém je uložena adresa začátku paměti procesu (tedy odděluje paměť operačního systému od paměti procesu), tento registr pak proces nesmí překročit.



Obrázek 3.1: Přidělení jedné souvislé oblasti paměti

Výhody:

- jednoduchost správy,
- nevelké nároky na technické vybavení (funguje to prakticky všude).

Nevýhody:

- nemožnost mít spuštěno více procesů najednou (jednoprogramový systém),

- velká část paměti může zůstat nevyužitá, pokud ji jeden běžící proces nepotřebuje, také ostatní prostředky výpočetního systému jsou nedostatečně využívány (procesor).

S mírným zvýšením složitosti lze i v tomto případě používat omezený běh více procesů (víceprogramový systém), a to tak, že když má být spuštěn další proces, celá paměť od mezního registru (přidělená původnímu procesu) se uloží do dočasného souboru na pevný disk, pak je přidělena nově spuštěnému procesu a až po jeho ukončení je obnovena do stavu před „zálohováním“ při spouštění dalšího procesu. Jestliže je takto postupně spuštěno více procesů, může být pro organizaci odložených procesů použit princip zásobníku.

3.2.2 Přidělování bloků pevné velikosti

Správce paměti při spuštění operačního systému rozdělí operační paměť na bloky pevné délky a ty pak přiděluje procesům. Procesu je při jeho spuštění přidělen paměťový blok, adresové prostory jednotlivých procesů jsou tedy odděleny. Tento typ adresování již dnes není moc používán, objevil se například u OS MFT (běžel na strojích IBM 360).

Bloky mohou být buď všechny stejně velké nebo různé velikosti. Druhá možnost znamená trochu složitější správu, ale umožňuje lépe pracovat s pamětí – procesu přidělujeme takový blok, který je volný a jeho velikost nejvíce odpovídá potřebám procesu (ale je zároveň větší nebo rovna požadované velikosti).

Protože počet bloků je konstantní po celou dobu běhu systému, je možné evidovat bloky v tabulce. Každý blok má jeden řádek tabulky, může být zde uložena počáteční adresa bloku, délka bloku a vlastník (proces) nebo informace že jde o volný blok.

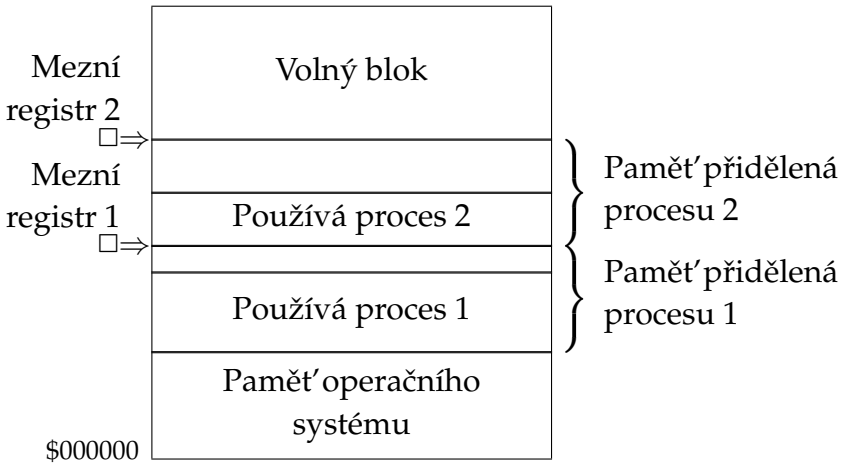
Metoda umožňuje implementaci multitaskingu za předpokladu, že je použita vhodná ochrana paměti (například dva mezní registry pro nejnižší a nejvyšší adresu právě běžícího procesu).

Výhody:

- jednoduchost správy,
- možnost implementace multitaskingu.

Nevýhody:

- proces požadující více paměti, než je délka největšího volného bloku, nelze spustit,
- velká pravděpodobnost fragmentace.



Obrázek 3.2: Přidělování bloků pevné velikosti (běží proces 2)

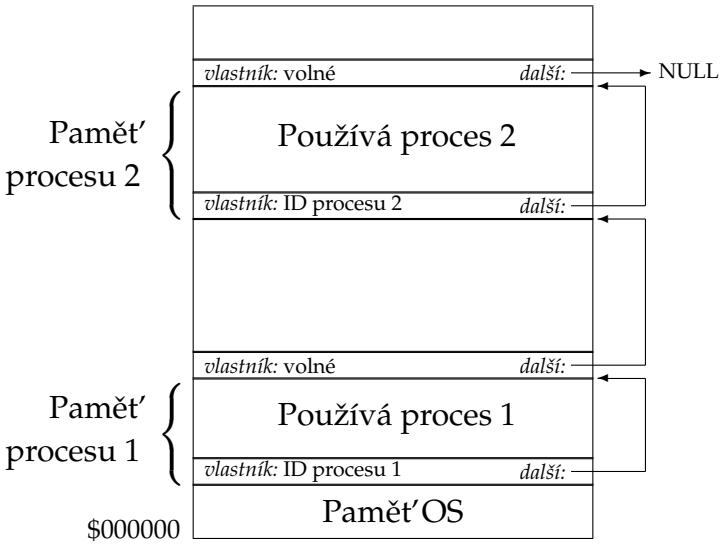
3.2.3 Dynamické přidělování bloků paměti

Nevýhody předchozí metody vyplývají především z nemožnosti určovat velikost bloků průběžně, při běhu systému. Tento problém můžeme řešit tak, že velikost přiděleného bloku určujeme až při žádosti procesu o paměť.

Proces při svém spuštění požádá o určité množství paměti. Správce paměti vyhledá volný blok s délkou větší nebo rovnou požadavkům procesu a tuto paměť přidělí. Pokud se ale nepodaří najít vhodný volný blok, proces nelze spustit. Před ukončením své činnosti proces musí vrátit přidělenou paměť a ta může být přidělena dalšímu procesu.

Protože počet a délka bloků se mění během práce systému, evidence bloků v tabulce není vhodná. Při neustálých změnách délky tabulky není možné předem odhadnout její maximální velikost. Řešení je více, například vytvoření hlaviček bloků paměti nepřístupných samotnému procesu (třebaže je mu tento blok přidělen), v hlavičce pak uložíme informaci o vlastníkovi nebo o tom, že se jedná o volný blok, a dále adresu počátku následujícího bloku (tedy ukazatel na další blok). Tímto způsobem zřetězíme bloky do dynamického seznamu, se kterým se již dá jednoduše pracovat. Paměť vyhrazená pro hlavičku bloku není procesu přístupná (ani o ní neví).

Pokud proces žádá o přístup do své paměti, k jeho adresovému prostoru se dostaneme jednoduše tak, že od prvního bloku postupujeme po ukazatelích na následující bloky, to tak dlouho, dokud podle informací v hlavičkách nenalezneme ten blok, který hledáme. Stejně postupujeme, když hledáme volný blok paměti pro přidělení nově spouštěnému procesu.



Obrázek 3.3: Dynamické přidělování bloků paměti

Při uvolňování bloku paměti postupujeme následovně: když je blok obklopen přidělenými bloky, změníme pouze informaci o vlastníkovi bloku. Jestliže však před nebo za tímto blokem je volný blok, musíme uvolňovanou oblast k tomuto bloku připojit. Tady je třeba jen dát pozor na to, aby nebyl narušen řetěz bloků, tedy zvolit vhodnou posloupnost přesměrování a uvolnění ukazatelů.

Výhody:

- všechny výhody předchozí metody, i když správa paměti je o něco náročnější a vyhledávání konkrétního bloku pomalejší,
- částečně odstraňuje nevýhody předchozí metody.

Nevýhody:

- počet procesů, které lze spustit, je limitován požadavky již spuštěných procesů, a pokud je paměť fragmentovaná, je maximální velikost požadavku na paměť limitovaná velikostí největšího volného bloku,
- určitá pravděpodobnost fragmentace.

Uvedené nevýhody se dají snížit metodami defragmentace paměti, které jsou probírány v kapitole 3.3 na str. 31.

3.2.4 Segmentace

Každému procesu je přiřazeno několik (různě dlouhých) bloků paměti, *segmentů*. Pokud je to potřeba a je v daném směru volná oblast paměti, segmenty lze prodlužo-

vat. Každý segment obvykle mívá určitý účel, například segment pro kód procesu (code segment), datový segment (data segment, jeden nebo více, segment pro proměnné s dynamickou délkou se obvykle nazývá *halda*), zásobníkový segment (stack segment, obsazuje se od nejvyšších adres k nejnižším), překryvný segment (overlay segment, například pro dynamické knihovny).

Některé segmenty jsou plně konstantní (nemění se jejich délka ani obsah, například kód procesu a globální konstanty), jiné mají konstantní délku, ale proměnný obsah (globální proměnné), další mají proměnnou délku i obsah (zásobník). To lze zohlednit při umísťování segmentů v paměti a řešení fragmentace. Procesy, které jsou instancemi téhož programu, mohou sdílet plně konstantní segmenty (pokud to systém umožňuje).

Procesy používají relativní adresy, adresy začátku jednotlivých segmentů jsou uloženy v *segmentových registrech* procesoru (tedy je to opět hardwarově závislé řešení, každý procesor má jiné registry). Absolutní adresa je pak vypočtena pomocí obsahu segmentových registrů. Adresa objektu z hlediska procesu má tedy dvě části – *segment* (určení, ve kterém segmentu se objekt nachází) a *offset* (relativní adresa v rámci segmentu, první Byte v segmentu má offset $= 0$).

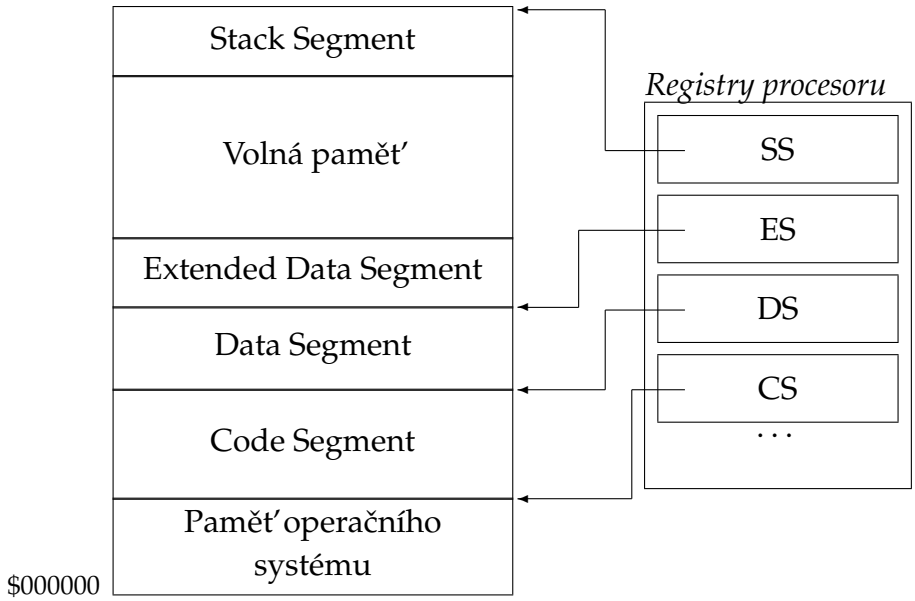
Výhodou tohoto řešení je, že případné přesouvání segmentu nezpůsobí procesu problémy s adresami. Je nutné zajišťovat mapování relativní adresy v segmentu na absolutní adresu. Pokud je implementován multitasking, je nutné při „výměně“ procesů na procesoru uložit obsah segmentových registrů odstavovaného procesoru a při znovupřidělení procesoru tomuto procesu znovu tyto hodnoty do registrů načíst.

Výhody:

- velikost segmentů může být různá, podle potřeby procesu,
- segmenty je možné prodlužovat a přesouvat,
- pokud to správce paměti umožní, některé segmenty lze sdílet.

Nevýhody:

- nutnost hardwarové podpory (segmentové registry),
- ochrana paměti je komplikovanější, protože segmenty mají proměnnou délku,
- paměť, kterou lze dalšímu procesu přidělit, je omezena velikostí největšího souvislého bloku volné paměti,
- určitá pravděpodobnost fragmentace, ta se ale dá řešit přesouváním segmentů.



Obrázek 3.4: Segmentace paměti (segmenty jediného procesu)

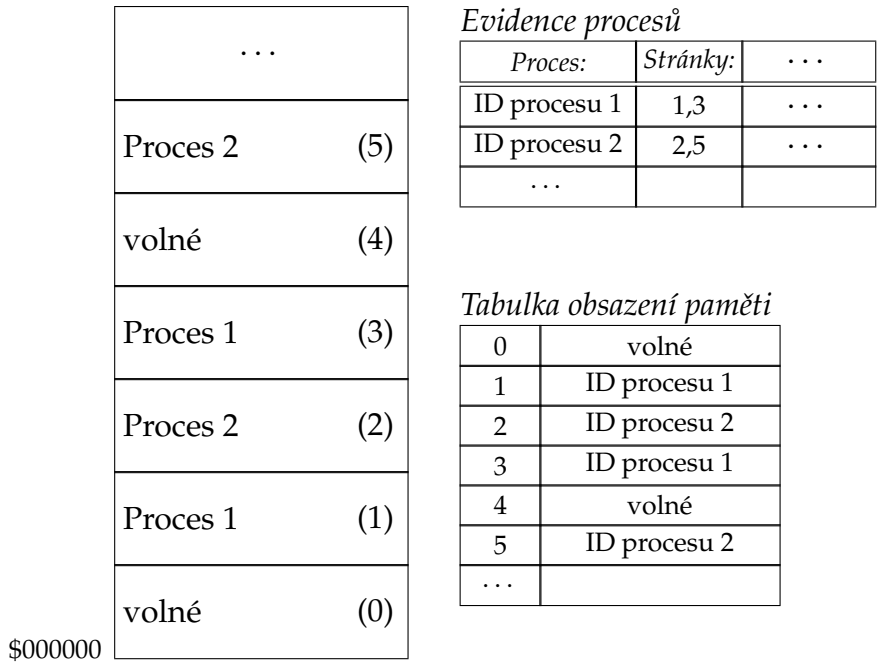
3.2.5 Jednoduché stránkování

Tato metoda rozlišuje fyzickou adresu objektu v paměti (to je absolutní adresa objektu) a logickou adresu tohoto objektu (s tou pracují procesy). Paměťový prostor je rozdělen na stejně dlouhé úseky – *stránky*, pokud možno spíše menší velikosti (obvykle jeden nebo více kB), procesu je přiděleno tolik úseků, kolik potřebuje. Procesu se jeho adresový prostor jeví jako spojitý, třebaže fyzicky spojitý nemusí být, používá z jeho hlediska „absolutní adresy“, které jsou ve skutečnosti pouze logickými adresami (od nuly) nebo se jako v případě segmentace paměti skládají ze dvou části – čísla stránky a relativní adresy uvnitř stránky.

Protože máme konstantní počet stránek (paměť je rozdělena již při spuštění operačního systému) a navíc jsou všechny stejně dlouhé, může být evidence stránek vedena v jednoduché tabulce, kde každé stránce je přiřazen vlastník nebo informace o tom, že jde o volnou stránku (nemusíme ani ukládat velikost stránky). U každého procesu je pak evidován seznam přidělených stránek. Při jakémkoliv přístupu do paměti správce paměti provádí překlad adres: číslo stránky $*$ délka stránky $+$ offset.

Výhody:

- proces může dostat tolik stránek, kolik potřebuje (pokud jsou volné), stránky nemusí na sebe navazovat,
- nejsou problémy s fragmentací.



Obrázek 3.5: Stránkování paměti

Nevýhody:

- fragmentace uvnitř stránek (proces nemusí potřebovat celou poslední stránku),
- omezení daná velikostí fyzického adresového prostoru.

3.3 Řešení fragmentace paměti

Paměť (vnější, například pevný disk, i vnitřní, tedy operační paměť) *je fragmentovaná, pokud volné oblasti paměti netvoří souvislý blok.*

Fragmentace na vnějším paměťovém médiu vzniká tehdy, když smažeme jeden soubor, do takto uvolněného místa je uložen nový soubor, ten se rozhodneme prodloužit a on se po tomto prodloužení do tohoto místa nevejde, tedy je nutné na konci volného místa vytvořit odkaz (ať už jakkoliv) na další volné místo, ve kterém soubor pokračuje. Aby byla paměť fragmentovaná, ale stačí i mnohem méně – pokud je při ukládání souboru vybíráno zbytečně velké místo.

U operační paměti je situace podobná, jen místo souborů pracujeme s paměťovými prostory jednotlivých procesů. Paměťové prostory procesů obvykle nebývají rozdrobeny na více částí, ale přesto k fragmentaci dochází. V případě, že o paměť žádá nově spouštěný proces, musí být procházena paměť a hledán vhodně velký

volný blok paměti, a v případě, že není nalezen dostatečně velký blok, proces nelze spustit.

Fragmentace se u reálných metod přidělování paměti dá snížit dvěma způsoby – vhodnou metodou výběru bloku paměti při žádosti procesu (alokační strategie) nebo setřásáním paměti.

Výběr vhodného bloku paměti: Když nově spouštěný proces požádá o paměť, stejným způsobem také hledáme vhodně velký volný blok. Základním předpokladem je, aby se do nalezeného bloku požadavek procesu vešel, což nám může dát více možností výběru bloku:

- metoda *first fit* – správce paměti prochází bloky od začátku uživatelské oblasti a přidělí paměť z prvního vhodného bloku, je to nejrychlejší metoda, i když ne nejoptimálnější (větší pravděpodobnost fragmentace),
- metoda *best fit* – správce paměti projde všechny bloky a hledá takový blok, který je vhodný (požadavek se do něho vejde) a zároveň je co nejmenší, je to nejoptimálnější metoda (je co nejmenší „zbytek“), ale časově náročnější než ta předchozí,
- metoda *last fit* – správce paměti vyhledá poslední vyhovující, tuto metodu použijeme, pokud paměť má být obsazována směrem od nejvyšších adres k nejnižším (práce s pamětí typu zásobník).

Pokud pouze volíme vhodnou metodu výběru bloku paměti, řešíme fragmentaci jen částečně. Společnou výhodou těchto metod je, narozdíl od následujícího řešení, že adresový prostor procesu se po celou dobu jeho běhu nemění.

Setřásání paměti (přesouvání bloků paměti): Abychom mohli spojit volné bloky do jednoho velkého adresového prostoru přidělitelného procesu s velkými paměťovými nároky, musíme obsazené bloky „setřást“ k nižším adresám, aby volné bloky na sebe navazovaly. Je však nutné vyřešit dva problémy: samotné přesouvání je časově náročné, a navíc se adresový prostor procesu, kterému je paměť přesouvána, mění (nemůže používat absolutní adresy).

První nevýhodu vyřešíme jednoduše tím, že k přesouvání bude docházet pouze tehdy, když to bude nutné, tedy ve chvíli, kdy o paměť bude žádat proces s nároky vyššími než je délka největšího paměťového bloku, a přesouvat budeme jen tak dlouho, dokud nevytvoříme dostatečně velký blok. Navíc základní desky bývají vybaveny možnostmi, jak procesor zbavit tohoto typu úloh (například čip *blitter* – block bits transfer, pomocný procesor pro přesuny paměťových bloků).

Druhou nevýhodu lze řešit několika způsoby:

1. Stanovení pravidel pro adresování na nižší úrovni, například používání relativních adres a vztahování k určitému registru, ve kterém je uložena adresa momentálního začátku adresového prostoru procesu.

   Výhodou je poměrně jednoduchá správa paměti a malá časová náročnost, nevýhodou je hardwarová závislost (není použitelné pro systémy portované na různé hardwarové architektury) a nutnost spolupráce programátorů aplikací (musí používat pouze relativní adresy).

2. Stanovení pravidel adresování na vyšší úrovni, například používat mechanismus zamykání bloku paměti po dobu jejího používání.

   Výhodou je jednoduchá správa paměti, nevýhodou je nutnost spolupráce programátorů aplikací a také jejich dobrá vůle (programátor také může zamknout blok hned při spuštění procesu a odemknout ho až při ukončování jeho běhu, tím znemožní veškeré přesouvání).

3. Před každým přesouváním správce paměti informuje každý proces, jehož adresový prostor je přesouván, o nové adrese začátku bloku, a proces si pak přepočítá všechny své absolutní adresy (obvykle ukazatele). Zpráva o přesouvání musí mít nejvyšší prioritu, aby se k procesům dostala včas.

   Tento způsob řešení klade vysoké nároky na systém i procesy, proto se používá pouze jako doplněk prvního uvedeného způsobu, a to pro programy, které musí používat absolutní adresy (ovladače, antivirové programy, apod.).

Metody setřásání paměti jsou dvě:

- kooperativní setřásání – použití druhého řešení, procesy na přesunech spolupracují se systémem, ovlivňují je (musí zamykat bloky), používalo se například v Apple MacIntosh do verze 9,
- transparentní setřásání – kombinace prvního a třetího řešení, procesy na přesunech nespolupracují, používalo se v systémech Epoc (když ještě šlo o operační systém pro osobní počítače).

3.4 Virtuální paměť

Virtuální paměť znamená rozšíření vnitřní paměti o oblast na vnějším paměťovém médiu, obvykle pevném disku. Důvodem je odstranění základní nevýhody všech

reálných metod přidělování paměti, nemožnosti spustit proces, jehož požadavky na paměť jsou vyšší než množství momentálně volné (fyzické) operační paměti.

Existuje více metod pro práci s virtuální pamětí, obvykle vycházejí z reálné metody stránkování (případně v kombinaci s jinou metodou). Fyzická vnitřní paměť je rozdělena na *rámce* a logická paměť je rozdělena na *stránky*. Všechny rámce a stránky mají stejnou velikost. Protože logický adresový prostor bývá rozsáhlejší než fyzický, bývá stránek obvykle více než rámců. Evidence paměti je vedena v *tabulce stránek*, tam kromě vlastníka stránky je uvedeno, kde se momentálně nachází (číslo rámce nebo adresa na disku).

Mnohé přidělené stránky nejsou zrovna používány, ať už proto, že proces, který je vlastní, zrovna nemá přidělen procesor, tedy „nic nedělá“, nebo tento proces zrovna nepotřebuje pracovat s obsahem této stránky (pracuje s jinou stránkou). Stránky mají buď přiřazen některý rámec, pak se nacházejí přímo ve fyzické paměti, nebo jsou odloženy na vnější paměťové médium, odkud v případě potřeby mohou být znovu načteny do některého rámce.

Je obvyklé, že některé stránky nelze odložit. Týká se to některých systémových stránek, kde by odkládání buď vedlo ke zpomalování systému nebo komplikovalo správu periferií (například stránky s ovladači zařízení). Pro systém bývají rezervovány nejvyšší logické adresy, a tedy proces používá adresy „od nuly“.

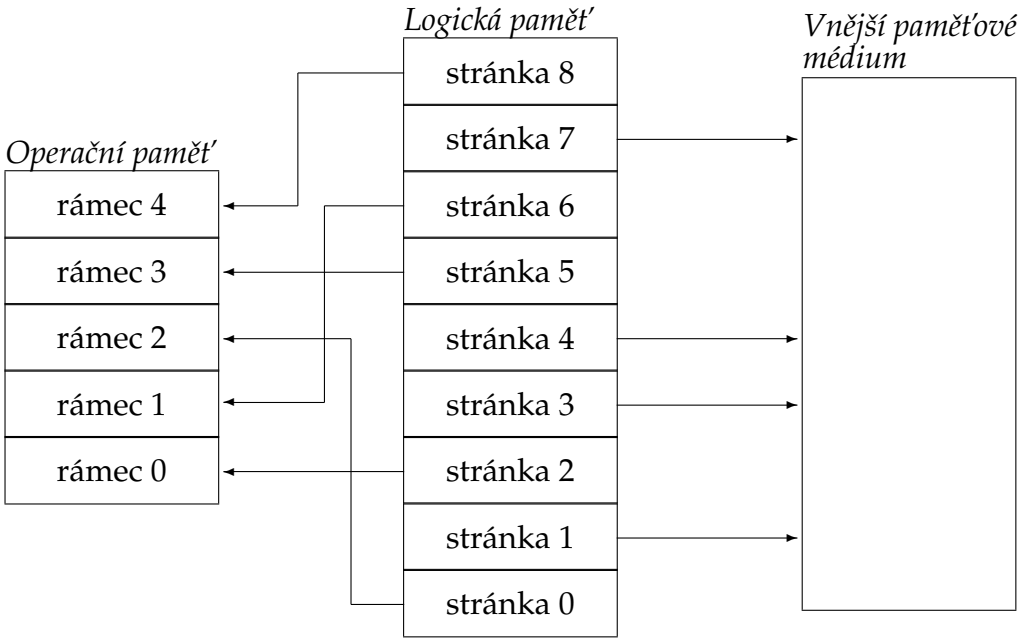
Pro odkládání stránek, které nemají přiřazen žádný rámec, se používá buď speciální soubor (odkládací soubor, swap soubor, stránkovací soubor) nebo celá oblast na disku. Protože pracujeme s vnějším paměťovým médiem, pro které obvykle existuje určitá hodnota stanovující délku nejmenšího možného bloku paměti, se kterým lze na tomto médiu pracovat (číst, zapisovat), pak velikost stránky se odvíjí od této hodnoty (celočíselný násobek, často přímo tato hodnota), obvyklá je velikost stránky 1024 B (1 kB) nebo 4096 B (4 kB).

3.4.1 Stránkování na žádost

Každý proces má přidělenu jednu nebo více stránek logické paměti. Oproti řešení základního stránkování je trochu složitější přepočítávání logických a fyzických adres, protože se musí řešit i případ, kdy je stránka odložená na disku. Proces ke svým stránkám přistupuje takto:

a) Stránka nachází ve fyzické paměti: pak této stránce odpovídá některý rámec. V tabulce stránek je uvedeno číslo tohoto rámce a pak se absolutní adresa vypočte z adresy začátku rámce (počet rámců krát velikost rámce) a přičte se offset (relativní adresa uvnitř stránky).

b) Stránka je odložena na disku: proces vyvolá přerušení nazvané *výpadek stránky* (page fault), čímž se přeruší zpracovávání jeho úlohy, správce paměti najde buď volný rámec nebo stránku, která sice má přiřazen rámec, ale lze ji odložit (tuto stránku odloží na disk), načte do rámce žádanou stránku a pak už k ní proces přistupuje stejně jako v prvním případě.



Obrázek 3.6: Stránkování na žádost

Při určování, který rámec má být uvolněn v případě, že je nutné některou stránku přesunout z disku do operační paměti, používáme některou z *metod výběru oběti*:

- FIFO (First In, First Out) – je odložena ta stránka, která má přiřazen rámec nejdéle (nejdéle nebyla odložena). Nevýhodou této metody je, že pokud je některá stránka používána hodně často, bývá také hodně často načtena v operační paměti a proto je podle této metody také nejčastěji odkládána. Správce paměti vede frontu umístěných stránek, vždy když je stránce přidělen rámec, je zařazena na konec fronty. Při nutnosti odložit stránku je odložena stránka ze začátku fronty.
- LFU (Laft Frequently Used) – je odložena nejméně používaná stránka. Komplikací metody je určení, která to vlastně je, musí se například u každé stránky vést čítač zvyšovaný o 1 při každém přístupu na stránku. Tato metoda bohužel postihuje nejvíc právě spuštěné procesy (jejich stránky dosud nebyly hodně používány).

- LRU (Last Recently Used) – je odložena stránka, která byla nejdéle nepoužívaná, tedy nejdéle „ležela ladem“. Tento algoritmus je z hlediska procesů nejlepší, implementace je však náročná (i časově), protože se předpokládá evidence času posledního přístupu na jednotlivé stránky. Proto je používanější zjednodušená verze této metody, pseudoLRU (NUR).
- NUR (Not Used Recently, pseudoLRU, hodinový aloritmus) – každá stránka má *used bit*, jeden bit, který je vždy při přístupu na stránku nastaven na 1. Správce paměti pak pořád dokola prochází tabulku stránek a used bity těch stránek, které mají přiřazeny rámce, nuluje (stránka bez rámce má logicky used bit nastaven na 0, proto byla vybrána jako oběť). Když při tomto nulování zjistí, že bit byl nastaven na 0, znamená to, že od posledního nulování stránka nebyla používána a je tedy lepším kandidátem na odložení na disk. V okamžiku vyvolání přerušení výpadku stránky může být při tomto procházení na kterékoliv stránce. Pak je za oběť vybrána nejbližší následující stránka s bitem nastaveným na 0. Této metodě se také říká *hodinový algoritmus*, protože správce cyklicky v daných intervalech prochází tabulku stránek.

Ochrana paměti je na vyšší úrovni především proto, že díky nutnosti překladu mezi logickou a fyzickou adresou se procesory běžným způsobem nedostanou mimo své paměťové stránky. Dále správce paměti má možnost označit některé (zejména systémové) stránky pouze pro čtení (na to stačí jediný bit, což není problém, pokud používáme used bity v metodě NUR) a při pokusu o zápis na takovou stránku je proces násilně ukončen.

Aby metoda mohla být realizována, usí být přítomna jednotka řízení paměti (dnes již bývá integrována v procesoru) zajišťující ochranu paměti, překlad adres apod., a dále procesor při výpadku stránky musí být schopen po přidělení rámce této stránce zopakovat poslední prováděnou instrukci procesu (tu, která vyvolala výpadek stránky). Metoda není zatížená fragmentací, může vznikat pouze *vnitřní fragmentace* – uvnitř stránek.

Výhody:

- všechny výhody metody základního stránkování,
- proces má k dispozici tolik paměti, kolik potřebuje, není limitován volným prostorem ve fyzické paměti.

Nevýhody:

- hardwarově závislé řešení.

3.4.2 Segmentace se stránkováním na žádost

Proces má přiděleno několik segmentů, každý segment se může rozkládat na několika stránkách. Paměť je tedy opět rozdělena na stránky. Adresa objektu v logickém adresovém prostoru je dána určením segmentu, číslem stránky a offsetem na stránce. Správce paměti vede u každého procesu zvlášť tabulku segmentů, u každého segmentu zde eviduje seznam stránek, na kterých se segment rozkládá. V tabulce stránek pak je zachyceno, zda má konkrétní stránka přidělen rámec nebo je odložena na disku.

Také zde je možné sdílet segmenty, jejichž délka ani obsah se nemění. Pro takový segment může být vedena jediná tabulka segmentů, i když tento segment využívá více procesů. Fyzická (absolutní) adresa v paměti se počítá tak, že v tabulce segmentů pro daný proces a segment najdeme číslo stránky uvedené v adrese, podle čísla poznáme, jaká je adresa začátku stránky (číslo krát délka stránky) a pak jen přičteme offset.

Výhody:

- všechny výhody metody stránkování na žádost,
- je možné sdílet segmenty.

Nevýhody:

- složitost implementace,
- hardwarově závislé řešení.

3.4.3 Swapování procesů

Swapování procesů je jednoduchá metoda virtualizace, která spočívá v tom, že se neodkládají jednotlivé stránky paměti, ale vždy celý paměťový prostor odkládaného procesu. Paměť vlastně ani nemusí být rozdělena na stránky či rámce (ale může), protože se stejně vždy pracuje s celým adresovým prostorem procesu.

Princip metody je podobný jako u stránkování: při vyhodnocování instrukce vyžadující přístup do paměti je buď tento přístup umožněn (pokud má proces svou paměť v operační paměti) nebo je vyvoláno přerušení. Při ošetření tohoto přerušení je nalezen vhodně velký volný prostor v operační paměti nebo je vytvořen (stejně jako u stránkování na žádost), paměť přesunuta a pak zopakována poslední instrukce.

Výhody:

- je to poměrně jednoduchá metoda,

- V celém časovém intervalu, po který je procesu přidělen procesor, je přerušení související s přesunem paměti vyvoláno nejvýše jednou.

Nevýhody:

- přesouvané bloky paměti jsou obecně různě velké, nejsou navzájem jednoduše zaměnitelné, tedy pro umístění dat jednoho procesu je někdy nutné odložit paměť několika „méně náročných" procesů a navíc je nutné nejdřív tento prostor najít,
- přesouvají se zbytečně velké paměťové bloky,
- hardwarově závislé řešení.

3.5 Správa paměti v některých operačních systémech

3.5.1 MS-DOS a Windows

MS-DOS používá základní metodu segmentace paměti, běžícímu procesu je přiděleno několik segmentů, z nichž každý má stanovený účel (segment kódu, datový, zásobníkový, překryvný pro načítané knihovny).

Neexistuje prakticky žádná možnost ochrany paměti. Spuštěný proces může přistupovat do kterékoliv části paměti včetně paměti vyhrazené pro operační systém, čehož se využívá především při přístupu k přerušením (pro řízení periferních zařízení apod.). Správa paměti je prováděna kombinací segmentace a přidělení jedné souvislé oblasti (je spuštěn jeden běžný proces a jsou mu přiděleny segmenty paměti). Jde o jednoprogramový systém, tedy může být spuštěn vždy jen jeden program.

Ve skutečnosti sice může běžet více programů současně, ale pouze tak, že nejdřív jsou spuštěny tzv. *rezidentní programy* (programy zůstávající v paměti po ukončení své nerezidentní části – TSR, Terminate and Stay Resident[2]) a pak (třeba i jejich prostřednictvím) běžný program. Při jeho běhu rezidentní programy nepracují, kromě zpracování přerušení, na která jsou napojeny.

Rezidentní programy často slouží k nahrazení některé funkce operačního systému (místo některé standardní znakové sady pro obrazovku přesměrovávají na takto přidanou speciální znakovou sadu), napojují se na přerušení (například antivirový program může být napojen na přerušení související s přístupem k souborům),

[2]TSR programy mají dvě části – rezidentní a nerezidentní (dočasnou). Při startu programu jsou načteny obě části, nerezidentní část provede „jednorázové" inicializační akce a je pak odstraněna z paměti, zůstává rezidentní část obsahující funkce napojené na ošetřovaná přerušení.

vylepšují uživatelské prostředí (grafické nebo pseudografické menu ke spouštění programů), zabezpečují některé důležité části systému (zabránění přístupu do některých částí paměti nebo do MBR disku), rezidentně pracují ovladače zařízení (myš), atd.

Napojení se provádí zajímavým, i když nepříliš bezpečným způsobem. Na začátku paměti je uložena posloupnost adres (*vektorů přerušení*) jednotlivých rutin zajišťujících obecné ošetření určitého přerušení, každá adresa odpovídá jednomu typu přerušení. Rezidentní program nebo běžný proces sem může místo původní adresy uložit adresu některé své funkce či procedury, která pak bude v případě vyvolání tohoto přerušení automaticky spuštěna. Je zvykem, že proces si uschová adresu původní rutiny a při svém ukončení ji načte zpět, případně ji spouští uvnitř své vlastní funkce pro ošetření přerušení (proč nevyužít to, co už je naprogramováno, když chceme pouze provést něco navíc). Tímto způsobem může být vytvořeno „zřetězení“ více funkcí různých TSR programů a samotného procesu.

Windows používají virtuální metodu segmentace se stránkováním na žádost. Každému procesu je přiděleno několik segmentů podobně jako v MS-DOSu. Informace o každém segmentu jsou uloženy v jeho *deskriptoru* (popisovači) (velikosti 8 B – počátek segmentu, délka, oprávnění, atd.), deskriptory segmentů jsou uloženy v *tabulce deskriptorů* procesu (LDT – Local Descriptor Table). Existuje také globální tabulka deskriptorů (GDT – Global Descriptor Table) obsahující deskriptory tabulek LDT jednotlivých procesů. Aby při adresaci nebylo nutné používat deskriptory, proces používá pro určení segmentu *selektory*, 16-bitové ukazatele do tabulky LDT tohoto procesu. Selektor je tedy obdobou adresy segmentu v MS-DOSu, má také stejnou délku, ale místo ukazatele na začátek segmentu jde o ukazatel do LDT. Logická adresa je dvouhodnotový vektor (selector,offset).

V 16-bitových Windows (do verze 3.x) bylo sdílení paměti zcela běžné, a to včetně dynamicky linkovaných knihoven. Pokud některý proces chtěl využívat funkce nebo objekty uložené v některé knihovně, při první žádosti o nalinkování obsahu této knihovny se její obsah načetl do operační paměti a proces dostal odkaz na tuto adresu. Při žádosti dalšího procesu se pak knihovna nenačítala do paměti znovu, ale další proces jen dostal odkaz na tutéž adresu v paměti, tedy oba procesy mohly přistupovat k téže oblasti paměti. Toho procesy využívaly také k meziprocesorové komunikaci. v 32-bitových Windows byly již tyto aktivity omezeny. Pokud proces požádá o přístup k dynamicky linkované knihovně (nebo jakémukoliv jinému souboru), je obsah knihovny *namapován* do adresového prostoru tohoto procesu. Tak je knihovna v paměti načtena i vícekrát.

Odkládací (stránkovací) soubor se jmenuje `pagefile.sys` a obvykle se nachází v kořenovém adresáři systémového disku. Standardně se jeho velikost mění, což při větší fragmentaci disku může způsobovat vážné zpomalení práce v systému. Základní konfigurace odkládání se provádí v nástroji *Systém*[3], kde (v případě Windows 2000) na kartě Upřesnit zvolíme tlačítko Možnosti výkonu a pak tlačítko Změnit[4].

Možnost nastavit umístění odkládacího souboru v systémech s NT jádrem je výhodná v případě, že máme nainstalováno více systémů rodiny Windows (například Windows 98 a XP) a chceme, aby používaly tentýž odkládací soubor. Je možné mít více odkládacích souborů a rozložit tak zátěž na více pevných disků. Pokud máme více pevných disků (fyzických, ne logických), doporučuje se umístit odkládací soubor na ten disk, na kterém není nainstalován systém, zrychlíme tím odkládání (na systémový disk se přistupuje hodně často, což zdržuje). Používání odkládacího souboru můžeme zakázat nastavením počátační i maximální velikosti na 0.

Procesory řady Intel, na kterých Windows obvykle běží, mají od x386 zabudovánu základní ochranu paměti, kterou Windows využívají, pokud běží v chráněném módu (režimu jádra). K paměti v režimu jádra mohou přistupovat pouze procesy běžící v režimu jádra. Dále se provádí kontrola při překladu logické adresy, zda proces přistupuje ke svým stránkám. Každá stránka má nastaven příznak určující, jak k ní lze přistupovat (pouze pro čtení, čtení a zápis, pouze spouštění kódu, spouštění, čtení a zápis, atd.).

3.5.2 Unixové systémy včetně Linuxu

Původní Unix běžel na hardwaru, který nepodporoval ochranu paměti, segmentaci ani virtualizaci (počítač PDP-7). Správa paměti probíhala formou podobnou metodě přidělování jedné souvislé oblasti paměti s vylepšeným multiprogramováním blízkým pravému multitaskingu. Při spuštění dalšího procesu byl celý paměťový prostor dosud běžícího procesu odložen (swapován) na disk.

V poměrně krátké době, jak byl Unix portován (přeložen, přepsán) na další hardwarové platformy, byla implementována podpora virtuální paměti se seg-

[3] K tomuto nástroji se dostaneme z Ovládacích panelů nebo v kontextovém menu ikony Tento počítač, položka Vlastnosti. To, jak proces využívá virtuální paměť, také zjistíme ve Správci úloh (Ctrl+Shift+Esc), pokud v menu Zobrazit – vybrat sloupce vhodně nastavíme zobrazení.

[4] Je možné zde nastavit umístění stránkovacího souboru (u Windows s NT jádrem) a nejmenší a největší velikost tohoto souboru. Aby se předešlo změnám velikosti souboru zpomalujícím systém, doporučuje se tato dvě čísla nastavit na stejnou hodnotu (a defragmentovat disk).

mentací i ochrany paměti, ale způsob odkládání zůstal podobný – odložen je vždy paměťový prostor celého odkládaného procesu, ne pouze jedna stránka, tedy je používána virtuální metoda swapování procesů. Přesto bývají používány i segmenty, mohou být sdíleny.

Některé dnešní Unixové systémy včetně Linuxu zvolily jinou formu virtualizace – stránkování na žádost (Linux) nebo stránkování se segmentací. Bylo zavedeno rozdělení paměťového prostoru na stránky, při přerušení výpadku stránky se pracuje s jednotlivými stránkami a ne s celými paměťovými prostory procesů. Přestože již nejde o swapování, i nadále se používá pojem swapovací soubor nebo swapovací oblast na disku.

Výběr oběti je na Unixech prováděn pomocí hodinového algoritmu (pseudo-LRU).

Tyto systémy podporují několik zajímavých prvků správy paměti, například sdílení kódu programů: pokud je spuštěno více procesů – instancí jednoho programu, mohou sdílet část paměti, ve které je načten kód programu. Podobně pracuje funkce *mapování souborů*, kdy do adresového prostoru může být namapován kterýkoliv soubor. Většina Unixů také umožňuje v případě přístupu na odloženou stránku v režimu pouze pro čtení přečíst data přímo z odložené stránky (nebo paměti procesu v případě swapování), což urychluje přístup k odloženým datům (není nutné vyvolat přerušení, hledat oběť, přesouvat bloky paměti).

Unixy jsou portovány na mnoha různých hardwarových platformách, proto je ochrana paměti na různé úrovni. Obvykle však Unixy využívají všechny možnosti, které daná hardwarová architektura nabízí, přinejmenším podporu privilegovaného a uživatelského režimu a ochranu segmentů.

3.5.3 MacOS

U systémů MacOS se od počátku počítalo s velkými nároky na operační paměť ze strany systému i procesů, proto byla implementována virtuální paměť metodou stránkování (také procesory, na kterém MacOS běžel, byly vybrány takové, které obsahovaly podporu virtuální paměti).

Procesory Motorola 680 00, na kterých běžely první operační systémy této řady, původně obsahovaly prostředky pro velmi pokročilou a na svou dobu rozsáhlou ochranu paměti, ale nevyužívaly ji[5]. Tato ochrana v jednotce řízení paměti zahrnovala podporu dvou režimů – privilegovaného (režim Supervisor) a uživatelského, ochranu systémové části paměti, podporu vyjímek, atd. Protože však operační

[5]Některé verze MacOS dokonce standardně spouštěly všechny procesy v privilegovaném režimu.

systém běžící na této architektuře nevyužíval jednotku řízení paměti a v ní implementovanou ochranu paměti, tato jednotka přestala být k procesorům 680 00 dodávána. U novějších pak již byla dodávána integrovaná přímo na čipu procesoru.

Virtuální paměť bylo možné na MacOS System 7 spravovat tak, že fyzická paměť sloužila jako pracovní „cache paměť". Veškerá logická paměť (stránky) byla odložena na disku a přímo v operační paměti byl vždy načten adresový prostor toho procesu, který měl zrovna přidělen procesor. Tento způsob práce s virtuální pamětí byl velice jednoduchý, ovšem použitelný pouze proto, že tyto verze nebyly plně multitaskové.

Systém NeXT Step příkladně využíval všech možností ochrany paměti počítačů NeXT, na kterých běžel, a tuto vlastnost pak přejaly i systémy MacOS X na něm postavené. Dnešní MacOS X je plně multitaskový systém a správa paměti je prováděna stránkováním pomocí tří modulů – *pagerů*, z nichž práci s běžnými stránkami paměti má na starosti jeden z nich. Tento modul pracuje pomocí procesu zvaného *dynamický pager*, stránkovací soubory se nazývají swapfile0, swapfile1, . . . , a jsou uloženy obvykle v adresáři nazvaném `/private/var/vm`. Je nastavena určitá maximální velikost stránkovacího souboru a při jejím dosažení je vytvořen další s vyšším číslem.

KAPITOLA 4

Procesy

V této kapitole se seznámíme se základy správy procesů. Definujeme proces a jeho vlastnosti, probereme základní formy multitaskingu, budeme se zabývat problematikou přidělování procesoru a možnostmi synchronizace procesů.

4.1 Evidence procesů

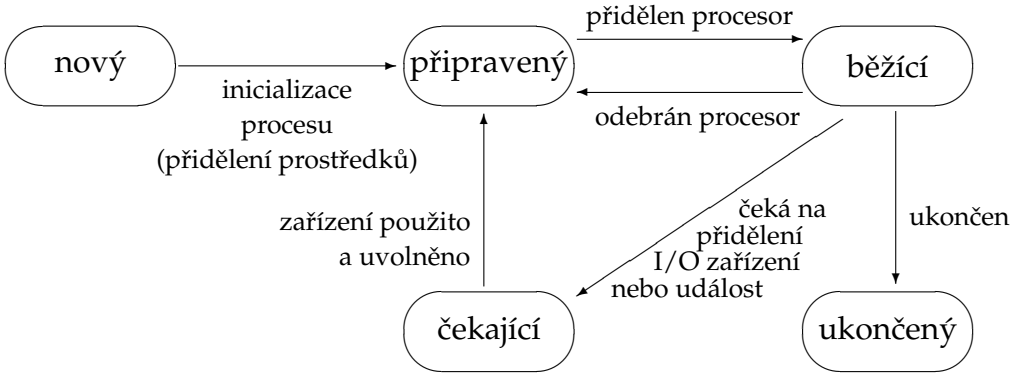
Zatímco program je jen soubor na vnějším paměťovém médiu obsahující kód (instrukce) a případně nějaká konstantní data, proces je instance programu určená nejen jeho kódem, ale i dalšími vlastnostmi, jako je jeho stav, priorita, identifikační číslo, programový čítač, přidělené prostředky (včetně paměti), atd.

Proces se tedy může nacházet v různých stavech:

- *nový* (new) – proces byl právě vytvořen, jsou mu přidělovány prostředky,
- *běžící* (running) – proces má právě přidělen procesor, tedy jeho kód je vykonáván,
- *připravený* (ready) – čeká na přidělení procesoru,
- *čekající* (waiting) – čeká na přístup k I/O prostředku, o který požádal nebo čeká na událost (například stisknutí klávesy),
- *ukončený* (terminated) – proces byl ukončen.

Proces mezi těmito stavy přechází tak, jak je naznačeno na obrázku 4.1 na str. 44.

Správce procesů vede tabulku procesů, záznam v této tabulce o konkrétním procesu se nazývá *Process Controll Block* (PCB). Je to souhrn všech dat, která operační



Obrázek 4.1: Přechody mezi stavy procesu

systém potřebuje k řízení procesů. Součástí PCB pro každý proces obvykle bývají tyto informace:

- PID (identifikační číslo procesu), případně další identifikační čísla určující například přístupová práva nebo vztah k jiným procesům,
- stav procesu,
- programový čítač určující, která instrukce se právě provádí (nebo má být provedena),
- hodnoty registrů,
- ukazatele do front, ve kterých proces čeká (procesor, I/O zařízení),
- informace pro správce paměti (tabulky obsazení paměti, evidence stránek, segmentů procesu),
- účtovací informace (týkající se přidělování procesoru),
- další momentálně přidělené prostředky (zařízení, otevřené soubory),
- atd. podle potřeb systému.

Proces může vzniknout několika způsoby:

- spuštěním programu jiným procesem (kromě prvního spuštěného procesu v systému samozřejmě), pak každý z procesů má jiný programový kód,
- klonováním již spuštěného procesu (fork) – celý paměťový prostor původního procesu je zkopírován do nového adresového prostoru, pak je novému procesu vytvořen vlastní záznam v tabulce procesů (PCB) s některými údaji původními a některými novými, pak oba procesy pokračují souběžně od stejného místa.

Obě možnosti se ve většině operačních systémů provádějí prakticky stejně, jen v prvním případě se po operaci fork (a před změnou v novém PCB) provede další volání jádra, tentokrát pro načtení kódu jiného programu do paměti spuštěného procesu.

Některé operační systémy (například Unixy) vytvářejí stromovou strukturu procesů, ve které je zachyceno, který proces byl kterým spuštěn. Spouštějící proces (nadřízený uzel) se nazývá rodič (parent), spouštěný proces je jeho synovským (dceřiným, child) procesem. V kořeni stromu je „praproces“, který buď přímo nebo zprostředkovaně spustil všechny ostatní běžící procesy. Každý další proces má kromě svého vlastního identifikačního čísla (PID) také uloženo identifikační číslo rodičovského procesu (PPID – Parent PID).

Mezi rodičovským a synovským procesem existuje jistý vztah závislosti. Obvykle po ukončení rodičovského procesu bývají ukončeny všechny procesy jeho podstromu, tedy všechny jeho synovské procesy, až na vyjímky, které z dobrých důvodů mají pokračovat v práci (například zálohování nebo dlouhodobé výpočty). Typickým příkladem je ukončení všech procesů spuštěných uživatelem po jeho odhlášení (všechny jsou v podstromě jeho přihlašovacího či inicializačního procesu).

Rodičovský proces může počkat na dokončení práce synovského procesu (použije volání jádra *wait*) nebo pokračovat ve své činnosti. Může taktéž v kterémkoliv okamžiku synovský proces ukončit (*abort*), například tehdy, když synovský proces splnil svůj úkol, pro který byl spuštěn.

4.2 Běh procesů a multitasking

Procesy mohou běžet několika způsoby:

- *sekvenčně* – další proces může být spuštěn až po ukončení činnosti předchozího,
- *sekvenčně-paralelně* – je spuštěno více procesů, které se dělí o čas procesoru (například se v určitých intervalech střídají při jeho využívání) ⇒ multitaskový systém,
- *paralelně* – procesy pracují souběžně, každý může běžet na jiném procesoru ⇒ multiprocesorový systém s multitaskingem.

Když se procesy střídají na jednom procesoru (k tomu může dojít v druhém nebo třetím případě), dochází k *přepínání kontextu*, tedy změně běhových informací

o procesu uložených na „globálních“ místech (například registry procesoru), proto je nutné kontext dosud běžícího procesu při každém přepnutí uložit, například do PCB (tedy tabulky procesů) nebo do paměťového prostoru příslušného procesu – do jeho zásobníku. Při přepínání kontextu se *uloží kontext* původně běžícího procesu a *obnoví kontext* následujícího procesu.

Kontext procesu je souhrn běhových informací o procesu. Při různých typech multitaskingu zde řadíme různé informace, obvykle jsou součástí kontextu tato data:

- obsah adresových registrů (programový čítač[1], segmentové registry, zásobníkový registr[2]),
- registr příznaků[3],
- pokud program není psán tak, aby počítal s případnými změnami v datových registrech při přepnutí kontextu, uložíme zde i obsah datových registrů,
- stav koprocesoru, pokud ho proces používá,
- stav dalších zařízení, která proces používá a nejsou řízena systémem.

Druh informací, které jsou součástí kontextu procesu, záleží především na typu multitaskingu, ve kterém procesy pracují.

Multitasking je postaven na *pseudoparalelismu* – prostředky (včetně paměti a času procesoru) jsou vyhrazeny více procesům, procesům je procesor přidělován střídavě podle určitého algoritmu, na uživatele tato metoda působí dojmem paralelní práce těchto procesů (jsou spuštěny a uživatel si může vybrat, se kterým bude pracovat).

Pseudomultitasking (multiprogramování, které není přímo multitaskingem, ale jeho předchůdcem) může být několika druhů:

- vzájemné volání – implementují procesy, nikoliv systém, procesu je přidělen procesor, pokud je volán jiným (právě běžícím) procesem, určitou formu této metody najdeme u systému MS-DOS, kdy TSR programy (viz str. 38) po

[1] Programový čítač (Instruction Pointer, IP) je adresa následující instrukce v kódu procesu, která má být zpracována.

[2] Zásobníkový registr (Stack Pointer, SP) obsahuje adresu vrcholu zásobníku. Do zásobníku se ukládají především data související s voláním funkcí – skutečné parametry funkce, lokální proměnné, návratové hodnoty apod.

[3] V registru příznaků jsou příznaky důsledků poslední provedené instrukce kódu, například zda je výsledkem výpočtu 0, bit znaménka výsledku, maskování přerušení, běh v režimu trasování (krokování) atd., u některých procesorů je zde také indikace běhu v režimu jádra (Motorola).

inicializaci své rezidentní části předávají aktivitu příkazovému interpretu (obvykle `command.com`), aby mohl být spuštěn další TSR program nebo běžný proces,

- omezené přepínání – systém přepíná mezi jedním běžným procesem a speciálními programy, které se nazývají *pomůcky* (accessories), pomůcky musí být pro tento účel speciálně programovány a bývají dodávány s operačním systémem jako drobné pomocné prográmky zjednodušující uživateli práci (jednoduchý textový editor, grafický editor, kalkulačka apod.), tuto možnost používaly nejstarší verze Apple MacOS, kde přepínání realizoval modul Finder,
- neomezené přepínání – je možné přepínat mezi jakýmikoliv běžícími procesy, tuto možnost používaly o něco novější verze Apple MacOS, kde přepínání realizoval modul MultiFinder.

Při přepínání, at'už omezeném nebo neomezeném, proces dává systému na vědomí, že může být ve své činnosti přerušen, a pokud uživatel rozhodne, že chce pracovat s jiným procesem, pak také tento proces přerušen je. Aby byly procesy „nuceny" dostatečně často sdělovat systému, že jim zrovna může být odebrán procesor, bývá tento stav (možnost odebrat procesor) často spojena s jinými službami systému, například čekání na stisk klávesy na klávesnici (proces může čekat na stisk klávesy jen tehdy, když umožní odebrání procesoru).

U vzájemného volání není potřeba používat kontext, u přepínání je součástí kontextu především vrchol zásobníku a další údaje závisí na konkrétním řešení (procesy samy určují, kdy mohou být přepnuty, mohou včas dokončit práci se zařízeními a uložit potřebné informace).

Kooperativní multitasking je vylepšením neomezeného přepínání. Jeden proces běží *na popředí*, ostatní procesy jsou spuštěny *na pozadí*. Proces na popředí má přidělen procesor, ale pokud ho zrovna nevyužívá (například čeká na událost, třeba vstup z klávesnice), může být procesor přidělen některému procesu na pozadí, ale jen na krátkou dobu. Po uplynutí této doby je procesor vrácen procesu na popředí a nebo, pokud tento proces pořád čeká na událost, opět některému procesu na pozadí. Uživatel určuje, který proces bude na popředí (například přesune se z textového editoru ke kalkulačce).

Narozdíl od neomezeného přepínání je při kooperativním multitaskingu dovoleno běžet procesům na pozadí, když proces na popředí nevyužívá procesor. Procesy musí na multitaskingu spolupracovat, a to odevzdávat procesor voláním

služby systému (proces na popředí, když nepotřebuje procesor, procesy na pozadí po uplynutí vyhrazeného krátkého času přidělení procesoru). Je však na samotném procesu (jeho programátorovi), zda příslušnou službu systému zavolá, a pokud se tím nebude obtěžovat, dostáváme se zpět na úroveň neomezeného přepínání. O obsahu kontextu platí totéž co u neomezeného přepínání.

V kooperativním multitaskingu pracovaly například Windows s DOS jádrem verze 3.x nebo novější verze Apple MacOS před verzí X.

Výhody:

- možnost spuštění více procesů,
- možnost spolupráce a komunikace procesů,
- lepší využití prostředků v systému (paměť, čas procesoru, atd.),
- možnost implementovat víceuživatelský systém (ale pouze na primitivní úrovni, plnohodnotně pracovat může pouze jeden uživatel),
- procesy „vědí“, kdy jsou přepnuty (samy vyvolávají přerušení vedoucí k odebrání procesoru), a proto kontext nemusí být tak obsáhlý.

Nevýhody:

- větší nároky na hardware,
- nutnost řešit problémy s bezpečností a stabilitou systému,
- pokud dojde k chybě v procesu běžícím na pozadí (zacyklení v nekonečné smyčce), nedojde k volání přerušení, není odevzdán procesor a celý systém „zamrzne“, totéž platí i pro proces běžící na popředí,
- pokud programátor nevolá službu systému umožňující přidělit procesor jinému procesu, systém přestává být multitaskový a jde pouze o pseudomultitasking s neomezeným přepínáním,
- náročnější realizace než u následujícího typu multitaskingu.

Preemptivní multitasking spočívá v neustálém přepínání mezi procesy. Procesy na multitaskingu nespolupracují (a dokonce o něm ani nemusí vědět), každý proces může být kdykoliv přerušen. Přerušení odebrání procesoru je vygenerováno při každé události v systému.

Kontext procesu musí obsahovat i takové údaje jako stav registrů procesoru a koprocesoru, protože proces po odebrání a znovupřidělení procesoru nemusí být informován o tom, že jeho činnost není časově souvislá a před chvílí registry

využíval jiný proces. Dále je třeba vyřešit přidělování prostředků, což obvykle bývá řešeno architekturou klient-server pro přístup k ovladačům zařízení (procesy – klienti přistupují k zařízením přes speciální procesy – servery, servery dokážou spolupracovat s kterýmkoliv klientem).

Preemptivní multitasking se sdílením času (time slicing) je vylepšením předchozí metody. K přepnutí kontextu dochází nejen při vygenerování nějaké události, ale navíc i v daných časových intervalech, a to velmi malých (jednotky až desítky milisekund). Procesy se ve využívání procesoru střídají, a to tak rychle, že na uživatele to působí dojmem paralelního zpracování úloh. Proces je přerušen po uplynutí určitého časového intervalu a nebo ještě dřív, pokud v jemu přiděleném intervalu došlo k přerušení generujícímu událost, a nebo když svou práci dokončí před koncem intervalu.

Tuto metodu používají prakticky všechny moderní operační systémy – Unixové systémy včetně Linuxu (pro běžné procesy neběžící v režimu jádra), Windows NT a Windows s DOS jádrem od verze 4 (95), MacOS X.

Výhody:

- možnost spuštění více procesů,
- možnost spolupráce a komunikace procesů,
- lepší využití prostředků v systému (paměť, čas procesoru, atd.),
- možnost implementovat víceuživatelský systém,
- možnost implementovat moderní interaktivní grafické rozhraní,
- snadnější implementace bezpečnostních mechanismů,
- poměrně snadná implementovatelnost (ve srovnání s kooperativním multitaskingem),
- metoda není závislá na běhu procesů a dobré vůli programátorů.

Nevýhody:

- větší nároky na hardware,
- kontext musí být o něco rozsáhlejší než u kooperativního multitaskingu.

4.3 Multithreading

Multithreading je vlastně paralelní zpracování více částí v rámci jednoho procesu, tedy něco jako multitasking uvnitř procesu. Rozdělení procesu na více takových

částí, podprocesů, vláken (thread) je výhodné, pokud se proces skládá z více nezávislých kusů kódu (navzájem se neovlivňují, je jedno, v jakém pořadí budou provedeny).

Typickým příkladem je aplikace, která komunikuje s uživatelem, umožňuje mu práci nebo ho baví (jedno vlákno) a „na pozadí“ třeba kopíruje soubory (druhé vlákno). Každé z těchto vláken pracuje samostatně, jedno nemá vliv na činnost druhého kromě případné komunikace (první vlákno může uživateli na vhodném grafickém prvku ukazovat, jak daleko je druhé vlákno v kopírování, druhé vlákno prvnímu vždy po zkopírování jednoho souboru nebo určitého kvanta Bytů zasílá zprávu).

V operačních systémech podporujících multithreading se proces (úloha) skládá z jednoho nebo více podprocesů nazývaných *vlákna*, jedno vlákno bývá hlavní a je spuštěno při spuštění procesu. Proces je pouze pasivní vlastník paměťového prostoru, veškerou činnost provádějí vlákna. Proto proces, který nemá žádné vlákno, může být ukončen.

Procesor není přidělován procesům, ale vláknům. Každé vlákno má svůj kód (nebo může být sdílený v rámci procesu, záleží na implementaci) a ukazatel do něho (programový čítač), zásobník, čas procesoru, kontext. Vlákna mohou mít každé svůj paměťový prostor nebo mohou přistupovat ke společnému paměťovému prostoru (to je obvyklejší), není nutné mezi vlákny uplatňovat mechanismy ochrany paměti ani další bezpečnostní metody (vlákna patří témuž procesu, spolupracují, nekonkurují si).

Protože vlákna obvykle přistupují ke stejnému paměťovému prostoru, je v některých případech potřeba jejich práci s paměťovými místy synchronizovat, viz jedna z následujících kapitol.

Každé vlákno pracuje zvlášť, ale spolupráce mezi vlákny jednoho procesu je užší než spolupráce s vláknem „cizího“ procesu. Tato spolupráce se netýká jen sdílené části paměťového prostoru, ale také času procesoru – když vlákno přestane používat procesor ještě dřív než vyprší jeho časový interval přidělení procesoru, nemusí zbylý čas „zahodit“, ale může ho přenechat jinému vláknu téhož procesu.

Vlákna mohou být implementována několika způsoby:

1. Model 1:1 – vlákna jsou implementována v jádře systému. Jádro s vlákny zachází jako s procesy, tedy přepínání kontextu se provádí na úrovni vláken. Toto řešení zvyšuje propustnost systému (systém reaguje pružněji, může být zpracováno více systémových volání zároveň, pokud je jádro také vícevláknové), ale je náročnější řešit problémy související se synchronizací systémových vláken (týkající se sdílených systémových dat).

Tato metoda je používána například systémem OS Mach, a proto také MacOS X, dále také v Linuxu (v Linuxu je však samotné jádro jednovláknové). Tato specifikace je součástí normy POSIX.

2. Model N:1 – vlákna jsou realizována na uživatelské úrovni. Podpora vláken je realizována v knihovnách, jádro je pouze jednovláknové. Systém vlákna nepodporuje, implementace je pouze na straně procesů. Vlákna jednoho procesu se dělí o čas procesoru přidělený tomuto procesu. Vlákna jednoho procesu nemohou běžet na více procesorech, proto tento model není vhodný pro víceprocesorové systémy.

   Protože přepínání vláken téhož procesu není realizováno „centrálně“, je mnohem rychlejší, proto je lepší odezva jednotlivých aplikací. Výhodou jsou menší komplikace při přístupu k systémovým datům, nevýhodou je v rámci jádra možnost najednou zpracovat jen jediné systémové volání. Když některé vlákno provede volání služeb jádra (systémové volání), zastaví se všechna vlákna procesu.

3. Model N:M – hybridní přístup. Vlákna jsou implementována na úrovni jádra (kernel-thread) i na úrovni běžných procesů (user-thread). Tento model odstraňuje nevýhody předchozích dvou modelů – přepínání vláken je rychlé, vlákna mohou běžet na více procesorech, jádro může najednou obsluhovat více systémových volání).

   Každé uživatelské vlákno procesu, které provádí nějaké systémové volání, je napojeno na některé vlákno jádra, ostatní uživatelská vlákna, která s jádrem nekomunikují, toto napojení nepotřebují. Tento model je použit ve většině komerčních Unixů (například Solaris).

4.4 Správa front procesů

Správa front procesů je základem pro spoustu dalších úkolů, které se v systému musí řešit. Fronty obvykle slouží k čekání procesů na prostředky v systému. Správce front především udržuje fronty – vytváří fronty a případně je ruší (při odebrání prostředku ze systému), přidává procesy do fronty, odebírá procesy z fronty (přidělení prostředku již má na starosti jiný modul systému). Existuje více různých druhů front:

Běžná fronta je fronta fungující způsobem *First-in first-out*.

Prioritní fronta je fronta, ve které jsou zohledňovány priority procesů. Procesy jsou zařazovány podle své priority před všechny procesy s nižší prioritou, za všechny s větší nebo stejnou prioritou.

Fronta typu delta-list je používána pro procesy, které čekají na uplynutí určitého časového intervalu. U každé položky fronty tedy máme dva údaje – první je ukazatel na proces (nebo složitější datová struktura reprezentující konkrétní proces), druhý je doba, po kterou má proces čekat. Používá se například v Unixových systémech pro spící (sleeping) procesy.

Aby u fronty typu delta-list nebylo nutné v pravidelných intervalech měnit dobu čekání u všech procesů ve frontě, používá se tato forma evidence času: dobu čekání evidujeme pouze u prvního procesu ve frontě, u ostatních je zachycen pouze rozdíl oproti předchozímu procesu ve frontě, pravidelně se zkracuje pouze údaj u prvního procesu.

Například máme ve frontě čtyři procesy: P_1 má čekat 30 ms., P_2 má čekat 65 ms., P_3 má čekat 14 ms., P_4 má čekat 142 ms.

Procesy seřadíme podle doby čekání, máme toto pořadí:

P_3	P_1	P_2	P_4
14	30	65	142

Fronta bude obsahovat tyto údaje:

P_3	P_1	P_2	P_4
14	16	35	77

Udržování takové fronty je jednoduché. V určitých časových intervalech je snižován údaj u prvního procesu ve frontě, a když dosáhne nuly, proces je „probuzen“, odstraněn z fronty. Protože u druhého procesu v pořadí byl evidován rozdíl vlastního času čekání a času čekání prvního procesu, který je teď 0, rozdílové číslo se teď stává skutečným časem čekání procesu.

Fronty také můžeme dělit podle postředků, na které v nich procesy čekají – fronta připravených procesů (čekají na přidělení procesoru), blokovaných (pro určité zařízení, např. tiskárnu nebo disk), spících procesů (je implementována frontou typu delta-list).

Jeden proces ve skutečnosti nemůže být ve více než jedné frontě (není v žádné frontě, pokud zrovna používá některý prostředek včetně procesoru), proto v jednodušším případě, kdy nechceme v různých frontách evidovat různé typy informací, stačí vést tabulku spuštěných procesů, a u každého identifikátor fronty, ve které čeká.

Častým způsobem implementace je také sada ukazatelů v PCB procesů, kdy v PCB jednoho procesu je ukazatel na následující proces v dané frontě, samotná fronta je pak reprezentována pouze ukazatelem na PCB prvního (pro odebírání z fronty) a posledního (pro přidávání) procesu ve frontě. Protože proces může být

v jednom okamžiku jen v jedné frontě, může být tento ukazatel v každém PCB jen jeden.

4.5 Přidělování procesoru

Na přidělování procesoru se podílejí dva moduly systému:

- *plánovač procesoru* (CPU Scheduler) používá frontu (fronty) připravených procesů a určuje, kterému procesu je přidělen procesor a na jak dlouho,
- *dispetcher* provádí vlastní přepnutí kontextu a přidělení procesoru, tedy uloží kontext dosud běžícího procesu včetně programového čítače, načte kontext procesu, kterému je procesor právě přidělován, zjistí hodnotu programového čítače a podle něho určí, od kterého místa v kódu procesu má tento proces běžet, a pokud je podporováno více režimů práce procesoru (privilegovaný, uživatelský), pak dispetcher provádí přepínání mezi těmito módy.

Doba, na kterou je procesu přidělen procesor, se nazývá časové kvantum. Na vhodné délce časového kvanta závisí funkčnost multitaskingu a tedy i kvalita zvolené metody přidělování procesoru. Pokud je příliš krátké, je časová režie spojená s přepínáním vysoká ve srovnání se skutečnou dobou běhu procesů, a tedy systém je neúměrně pomalý. Jestliže je naopak časové kvantum zbytečně velké, pak procesy hodně používající I/O zařízení využívají jen malou část přiděleného kvanta a procesor musí být stejně přepínán častěji, a navíc je systém méně interaktivní.

Procesy můžeme rozdělit do dvou skupin na ty, které více využívají procesor (*CPU-bound*) a procesy, které více používají I/O zařízení (*I/O-bound*). Každý z nich má trochu jiné nároky na procesor, CPU-bound procesy obvykle využití celé přidělené kvantum, u I/O-bound procesů je zase mnohem větší pravděpodobnost přerušení běhu. Plánovač procesoru by měl rozlišovat mezi těmito dvěma skupinami procesů, aby bylo využití procesoru optimální a aby byl přidělován procesům z obou skupin rovnoměrně.

Plánování procesů můžeme rozdělit do tří oblastí:

1. *Dlouhodobé plánování* souvisí se samotným návrhem multitaskingu, s určením toho, co se má provést při vytvoření procesu a plánováním zacházení s CPU-bound a I/O-bound procesy.

2. *Střednědobé plánování* provádí správce paměti, jde například o rozhodování, který proces bude odložen (suspended) a tedy mu není přidělován procesor.

3. *Krátkodobé plánování* představuje samotné plánování procesoru, kdy se určuje například časové kvantum procesů, frekvence přerušení generovaných časovačem pro přerušení běhu procesu, atd.

Plánování může být preemptivní nebo nepreemptivní. Jestliže je použita metoda *nepreemptivního plánování*, pak běžící proces využívá procesor tak dlouho, jak sám potřebuje nebo do vygenerování přerušení, tedy procesor je odejmut až po některém systémovém volání, u *preemptivního plánování* může být procesor odebrán plánovačem dříve, například při přerušení generovaném časovačem.

Dále probereme základní metody plánování procesoru.

4.5.1 Fronta (FCFS)

(First Come First Served) Při použití této metody je fronta připravených procesů organizována jako klasická FIFO struktura, tedy procesy jsou řazeny na konec fronty a vybírány ze začátku.

Je to nepreemptivní metoda, procesy používají procesor tak dlouho, dokud není vygenerováno přerušení nebo pokud samy neodevzdají procesor. Do fronty jsou procesy řazeny i z jiných front (například předtím mohl proces využívat I/O zařízení).

Nevýhodou je, že CPU-bound procesy si vyhrazují příliš mnoho času procesoru a tedy I/O-bound procesy jsou znevýhodněny. Tato metoda je proto implementovatelná pouze v kombinaci s prioritami procesů (I/O-bound procesy by měly mít vyšší prioritu).

4.5.2 Cyklické plánování (RR)

(Round Robin) Tato metoda je podobná předchozí, také používáme frontu s organizací FIFO. Rozdíl je v tom, že každý proces může běžet na procesoru jen po stanovenou dobu, časové kvantum, je to tedy preemptivní metoda. Po ukončení běhu procesu je tento proces zařazen na konec fronty připravených procesů, pokud není (například při přerušení jemu určením) zařazen do jiné fronty nebo převeden do stavu sleeping či suspended[4].

Každému procesu je procesor přidělen na stejnou dobu (časové kvantum). Pokud je časové kvantum příliš velké, metoda svou funkčností odpovídá předchozí

[4]Proces se dostane do stavu sleeping (spící), pokud je zařazen do fronty typu delta-list, do stavu suspended se dostává například při odložení všech částí svého adresového prostoru do odkládací oblasti.

metodě. Opět jsou zvýhodněny CPU-bound procesy, protože předbíhají I/O-bound procesy čekající na přidělení zařízení.

Metodu lze brát jako základ pro další pokročilejší preemptivní metody plánování a lze ji vylepšit například tak, že pokud byl běžícímu procesu procesor odejmut po přerušení souvisejícím s I/O zařízením, je pak proces s nevyužitou částí časového kvanta zařazen místo hlavní fronty připravených procesů do pomocné fronty, která má přidělenu vyšší prioritu, a tedy zbylé časové kvantum může vyčerpat dříve než kdyby byla metoda uplatněna v základní verzi. Po vyčerpání zbytku časového kvanta je proces zařazen opět do hlavní fronty připravených procesů.

4.5.3 Nejkratší úloha (SPN)

(Shortest Process Next, SJF – Shortest Job First) Procesor je přidělen tomu procesu, u kterého se předpokládá nejkratší doba jeho využívání (nejmenší časové kvantum). Fronta je vedena jako prioritní s tím, že priority jsou zde určeny velikostí časového kvanta.

Metoda má preemptivní i nepreemptivní verzi. Pokud je do fronty připravených řazen proces, u něhož se předpokládá menší časové kvantum než u právě běžícího procesu, při nepreemptivním plánování běžící proces může běžet i dále a teprve když (nepreemptivně) přijde o procesor, pak může běžet nově řazený proces, při preemptivním plánování je v takovém případě běžící proces okamžitě přerušen a procesor je přidělen dalšímu procesu.

Je nutné co nejlépe odhadnout časové kvantum pro tento úsek běhu procesu. Pro odhadnutí časového kvanta existuje více možností (označme n počet dosavadních přidělení procesoru danému procesu, R pole hodnot skutečných časových kvant, zatím o délce n, a S pole odhadů časových kvant):

1. Následující časové kvantum bývá často stejné jako předchozí, tedy budeme předpokládat, že při následujícím přidělení procesoru bude proces potřebovat asi tolik času kolik využil při posledním přidělení.

$$S[n+1] = R[n] \tag{4.1}$$

2. Exponenciální průměrování – u každého procesu zaznamenáváme délku skutečně využité doby přidělení procesoru v minulosti a vhodné časové kvantum odhadujeme výpočtem aritmetického průměru všech předchozích skutečných časových kvant. Aby nebylo nutné vždy počítat aritmetický průměr

všech hodnot, lze vzorec zjednodušit využitím předchozího odhadu a odpovídajícího skutečného časového kvanta.

$$\begin{aligned} S[n+1] &= \frac{1}{n} \cdot \sum_{i=1}^{n} \cdot R[i] && (4.2) \\ &= \frac{1}{n} \cdot R[n] + \frac{1}{n} \sum_{i=1}^{n-1} \cdot R[i] \\ &= \frac{1}{n} \cdot R[n] + \frac{n-1}{n} \cdot S[n] && (4.3) \end{aligned}$$

3. Zkombinujeme oba přístupy (podle vzorců 4.1 a 4.3), tedy budeme předpokládat, že další časové kvantum se nebude příliš lišit od předchozího, ale vezmeme v úvahu i předchozí délky úseků, třebaže s menší vahou.

   Volíme vhodnou konstantu c, $0 < c < 1$. Pokud je tato konstanta blíže jedničce, má výrazně větší váhu délka posledního skutečného časového kvanta, a čím blíže je nule, tím větší váhu mají rozdíly v dřívějších kvantech. První odhad ($S[1]$) je obvykle nastaven na 0.

   $$S[n+1] = c \cdot R[n] + (1-c) \cdot S[n] \tag{4.4}$$

   Význam konstanty c je zřejmý z rekurzivního rozložení vzorce:

$$\begin{aligned} S[n+1] &= c \cdot R[n] + (1-c) \cdot \Big(c \cdot R[n-1] + (1-c) \cdot S[n-1] \Big) \\ &\vdots \\ &= c \cdot R[n] + (1-c) \cdot c \cdot R[n-1] + \ldots \\ &\quad \ldots + (1-c)^{n-1} \cdot c \cdot R[1] + (1-c)^n \cdot S[1] && (4.5) \end{aligned}$$

Tato metoda zvýhodňuje I/O-bound procesy, jejichž časové kvantum bývá menší, a výrazně znevýhodňuje CPU-bound, zvláště když jde o dlouho běžící procesy. Déle běžící procesy mohou stárnout, tedy jejich běh přestává být aktuální (ztrácejí význam). Pokud je v procesu chyba (nekonečný cyklus), pak chybně běžící proces neblokuje procesor a lze ho snadno detekovat (zůstává na konci fronty, je neustále odstavován).

4.5.4 Priority

Při uplatnění této metody přidělujeme procesor procesu s nejvyšší prioritou, tedy používáme prioritní frontu. Metoda má opět preemptivní i nepreemptivní variantu,

stejně jako předchozí. Za variantu této metody můžeme považovat také metodu SPN (předchozí), kde se priorita odvíjí od časového kvanta procesu (čím menší kvantum, tím vyšší priorita).

Priorita procesu může být určena staticky (*statická priorita*, stanoví se předem, při spuštění procesu) nebo dynamicky (*dynamická priorita*, mění se za běhu procesu). Dynamická priorita při vhodném použití snižuje nebezpečí stárnutí procesů s nízkou prioritou, priorita může být u déle čekajících procesů postupně zvyšována.

Moderní systémy používají priority v kombinaci s jinými metodami (viz následující metoda), a je to také podmínka alespoň základní podpory real-time procesů.

4.5.5 Kombinace metod s více frontami

Je vedeno více front připravených procesů, procesy jsou rozdělovány dle určitého algoritmu. Pro každou frontu je stanovena některá metoda plánování, a dále jedna z metod je uplatňována při rozhodování mezi frontami.

Jednoduchý algoritmus řazení procesů do jednotlivých front spočívá v tom, že každá fronta je určena pro určitý typ procesů (systémové, interaktivní, dávkové, ostatní) s tím, že jednotlivé fronty jsou organizovány metodou RR (cyklické plánování), FCFS (fronta) nebo použitím dynamické priority. Každá fronta má stanovenu prioritu (netýkající se jednotlivých procesů, ale celé fronty), a přednostně jsou brány procesy z fronty s nejvyšší prioritou.

Efektivnější algoritmus umožňuje přesouvání procesů mezi frontami, fronty nejsou určeny pouze pro konkrétní typ procesů. Fronty jsou uspořádány do posloupnosti s klesající prioritou (tj. první fronta v posloupnosti má nejvyšší prioritu). Proces je nejdřív zařazen do první fronty, když vyčerpá své časové kvantum, je pro čekání na přidělení dalšího časového kvanta zařezen do druhé fronty, pak do třetí, atd. Při přerušení běhu I/O zařízením se po opětovném přechodu procesu do stavu připravený proces zařadí do první fronty. Plánovač procesoru začne pracovat s následující frontou až tehdy, když jsou všechny předchozí fronty prázdné. Poslední fronta je organizována metodou RR, ostatní metodou FCFS.

Tento algoritmus zvýhodňuje I/O-bound procesy, protože ty se po každém použití I/O zařízení vracejí do první fronty, CPU-bound procesy s časem postupují do následujících front s menší prioritou.

4.6 Komunikace procesů

Jednou z výhod multitaskingu je možnost snadné komunikace procesů – meziprocesové komunikace (IPC, Interprocess Communication). Rozlišujeme proces odesílající (odesílatel, sender) a proces přijímající (příjemce, receiver). Odesílatel může poslat

- data či textový řetězec (případně s délkou omezenou určitou konstantou) nebo
- odkaz na data (adresa v paměti nebo na pevném paměťovém médiu, může jít o dočasný soubor) nebo
- signál (číslo s určitým významem, například informace o tom, že má proces ukončit svou činnost).

Rozlišujeme dva základní typy komunikace:

- přímá – příjemce je předem znám (zasílání zpráv),
- nepřímá – příjemce není určen odesílatelem při odesílání dat, ale až během samotného přesunu (schránka, sdílená paměť) nebo navázáním spojením zvnějšku (roury – pipes).

Pokud je příjemce pouze jeden a odesílatel ho přímo adresuje, model komunikace nazýváme *unicast*, jestliže je zpráva (nebo jakákoliv data) určena všem, kdo mohou komunikovat, a to bez konkrétní adresace, pak je to model *broadcast*, pokud je více konkrétních adresovaných příjemců, jde o model *multicast*.

Přímá komunikace (zasílání zpráv) má výhodu především v širších možnostech použití. Zprávy lze zasílat také procesům běžícím na jiném procesoru nebo počítači, nejsme vázáni podmínkou existence sdíleného paměťového prostoru pro odesílatele a příjemce. Zasílání zpráv je realizováno následujícími funkcemi:

- `send(P,zpráva)` – proces odešle příjemci – procesu `P` – zprávu,
- `receive(Q,zpráva)` – proces přijme (vyzvedne si) zprávu od odesílatele `Q`, zpráva se načte do druhého parametru.

Přímou komunikaci dělíme na

- symetrická – odesílatel a příjemce zprávy se navzájem dokážou identifikovat, každý ví, s kým komunikuje, lze realizovat například prioritní frontou, ze které se přednostně vybírají zprávy určitého odesílatele,

- asymetrická – příjemce nemusí znát odesílatele, jen odesílatel ví, komu zprávu posílá. Potom příjemce nezadává identifikaci odesílatele do prvního parametru funkce `receive`, ale tento údaj je do tohoto parametru načten stejně jako samotná zpráva (odpovídá jednoduchému vybírání zpráv z fronty).

Přímou komunikaci dále dělíme na

- asynchronní – odesílající proces nemusí čekat na odpověď,
- synchronní – odesílající proces musí čekat na potvrzení zprávy nebo odpověď (do té doby je blokován, obvykle ve stavu suspended).

Zasílání zpráv lze implementovat mnoha způsoby, například tak, že odesílaná zpráva je uložena odesílatelem do fronty ve společné či systémové paměti, z které jsou zprávy k tomu určeným modulem správy procesů postupně vybírány a odesílány, tedy kopírovány do paměťového prostoru příjemce (jeho fronty zpráv).

Synchronní komunikace bývá někdy implementována třemi funkcemi – kromě dříve uvedených `send` a `receive` ještě `reply(P,zpráva)` pro potvrzení přijetí zprávy od procesu `P`. Odesílatel je po odeslání zprávy suspendován a může pokračovat až po obdržení potvrzení vyslaného příjemcem pomocí funkce `reply`.

Tak funguje například *RPC* (Remote Procedure Call, volání vzdálené procedury, tedy procedury nepatřící do kódu volajícího procesu). Odesílatel je proces volající vzdálenou proceduru, příjemce je proces, v jehož kódu se tato procedura nachází. Odesílatel je blokován až do chvíle, kdy příjemce odešle `reply` o provedení volané procedury.

Nepřímá komunikace probíhá přes rozhraní představované bodem spojení, nazývaným obvykle *port* (brána, socket, schránka). Port může být vytvořen kterýmkoliv procesem nebo operačním systémem. Vlastníkem portu je ten proces, který ho vytvořil, nebo může být vlastnictví převedeno na jiný proces. Do portu může zapisovat jen vlastník (odesílatel), ostatní procesy, kterým je k portu dovolen přístup, mohou jen číst (příjemci). Komunikace probíhá pomocí funkcí

- `send(ID_portu,zpráva)` – odesílatel uloží do zadaného portu zprávu,
- `receive(ID_portu,zpráva)` – příjemce vyzvedne zprávu z daného portu.

Speciální typ socketu (portu) je *pipe* (roura). Jde o soubor pevné délky (v Unixových systémech), který obvykle ani nebývá uložen na disk, zůstává v operační paměti, nebývá ani stránkován. Odesílatel ho vytvoří (systém pro tento účel obvykle nabízí některé systémové volání) a postupně naplňuje daty. Po naplnění je odesílatel blokován, dokud příjemce nepřečte celý tento soubor, jeho obsah je pak smazán a odesílatel může pokračovat.

KAPITOLA 5

Synchronizace procesů

V multitaskovém systému se běžně stává, že více procesů potřebuje přistupovat ke stejnému prostředku. Tímto prostředkem může být běžné I/O zařízení, jako je obrazovka, klávesnice či tiskárna, ale také sdílená oblast paměti. V této kapitole si popíšeme základní problémy související se synchronizací procesů a metody, kterými je lze řešit.

5.1 Úvod do problematiky

Při přístupu více procesů k témuž prostředku je hlavním problémem zajištění *konzistentního stavu prostředku*. V případě sdílené paměti jde o konzistenci dat, tedy pokud některý proces zapisuje do této paměti, jiný by neměl číst, dokud zapisující proces nedokončí svou práci, protože by mohl načíst jen zpoloviny modifikovaná data. Data jsou v konzistentním stavu před začátkem zápisu a po dokončení zápisu.

Nejpřísnějším kritériem pro přístup k prostředkům jsou *Bernsteinovy podmínky*. Označme

- *read(P,t)* množinu všech prostředků včetně paměťových míst, ze kterých se proces P pokouší číst v čase (okamžiku) t,
- *write(P,t)* množinu všech prostředků, na které se proces P pokouší v čase t zapisovat (provádět jakékoliv změny).

Bernsteinovy podmínky pro jakékoliv dva procesy P, Q jsou následující:

$$read(P,t) \cap write(Q,t) = \emptyset \quad (5.1)$$

$$write(P,t) \cap write(Q,t) = \emptyset \quad (5.2)$$

Znamená to, že je zakázáno přistupovat k témuž bodu (portu, socketu, zařízení, místu v paměti), ať už pro čtení nebo zápis, pokud zde v té chvíli provádí zápis jiný proces.

Bernsteinovy podmínky jsou hodně silné a je těžké je dodržet. Proto se obvykle používají jiné, jemnější podmínky, třebaže jejich implementace je náročnější. Řešení pro přístup k periferním zařízením si ukážeme v jedné z následujících kapitol, zde se budeme zabývat synchronizací přístupu ke sdílené paměti a synchronizací běhu procesů při jejich komunikaci.

5.2 Petriho sítě

Petriho sítě jsou vizualizační prostředek, který přehledně zachycuje tok dat nebo jakékoliv paralelní či pseudoparalelní postupy na abstraktní úrovni. Zde je budeme používat pro první fázi návrhu řešení problémů vznikajících při synchronizaci prostředků.

Petriho síť je orientovaný graf s dvěma typy uzlů:

místa, představují stavy procesu nebo stavy systému,

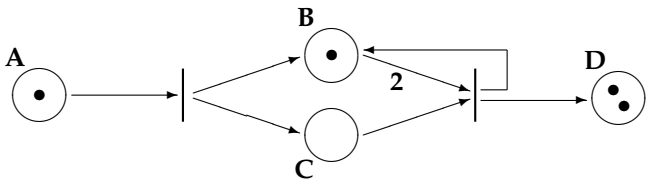
přechody, představují určitou činnost procesu nebo systému probíhající mezi dvěma stavy (představovanými místy).

Místa a přechody se v síti střídají, nesmí být přímo za sebou dva uzly stejného typu. V místech mohou být *tečky* (tokeny) představující „povolení“ pokračovat v grafu dále. Každá hrana je ohodnocena přirozeným číslem (pokud není číslo uvedeno, je to 1), toto číslo znamená násobnost hrany.

Aby přechod mohl být proveden, musí být v každém místě, z něhož do přechodu vede hrana, nejméně tolik teček, jaké je ohodnocení této hrany. Provedení přechodu probíhá takto:

1) z každého místa, z něhož do přechodu vede cesta (šipka) ohodnocená číslem **n**, ubere **n** teček,
2) do každého místa, do kterého z něj vede cesta ohodnocená číslem **m**, přidá **m** teček.

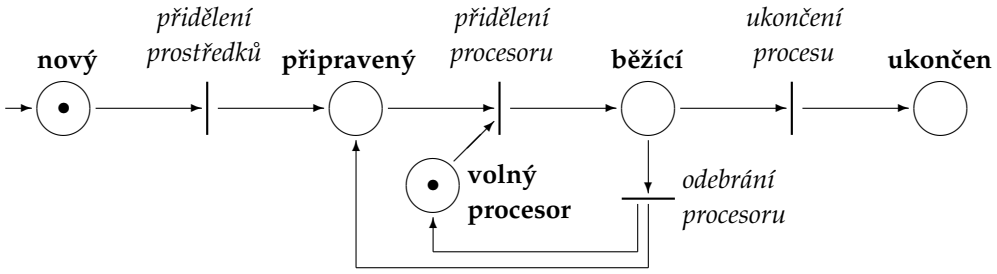
Na obrázku 5.1 jsou dva přechody, z nichž je v tomto stavu sítě proveditelný pouze ten první, druhý není proveditelný, protože v místě **C** není žádná tečka a v místě **B** je pouze jedna, musí být dvě. Při provedení prvního přechodu se



Obrázek 5.1: Příklad Petriho sítě

z místa **A** odebere tečka (pouze jedna, hrana není označena číslem) a do míst **B** a **C** se přidá po jedné tečce. Teď už je proveditelný druhý přechod. Při jeho provedení se z místa **B** odeberou dvě tečky a z místa **C** jedna tečka a přidá se tečka do míst **B** a **D**. V místě **D** teď budou tři tečky.

Na obrázku 5.2 je ukázka petriho sítě popisující zjednodušený běh procesu využívajícího pouze procesor s tím, že žádný jiný proces neběží.



Obrázek 5.2: Petriho síť popisující běh velmi jednoduchého procesu

Tečku v místě označeném **nový** můžeme chápat jako stav, ve kterém se momentálně nachází vykonávání procesu. Všechny přechody jsou ohodnoceny číslem 1 (číslo 1 se nemusí uvádět).

Místo **volný procesor** představuje stav systému, ve kterém může být přidělen procesor. Obsahuje tečku pouze tehdy, když je procesor volný a může proběhnout jeho přidělení. Po provedení přechodu *přidělení procesoru* se odebere tečka z míst **připravený** a **volný procesor** a přidá se tečka do místa **běžící**. Pak může být proveden přechod *odebrání procesoru*, přičemž se odebere tečka z místa **běžící** a přidá se do míst **volný procesor** a **připravený**.

V případě, že je spuštěno více procesů, všechny tyto procesy využívají místo **volný procesor**. Kdykoliv se některý proces dostane do stavu běžící, odebere tečku z tohoto místa a ostatní procesy musí počkat ve stavu připravený, tedy v místě **připravený** daného procesu, dokud běžící proces tečku nevrátí.

Petriho síť plně popisující běh a synchronizaci procesů by byla příliš složitá a rozsáhlá, proto budeme používat zjednodušená schémata, kde jednotlivá místa

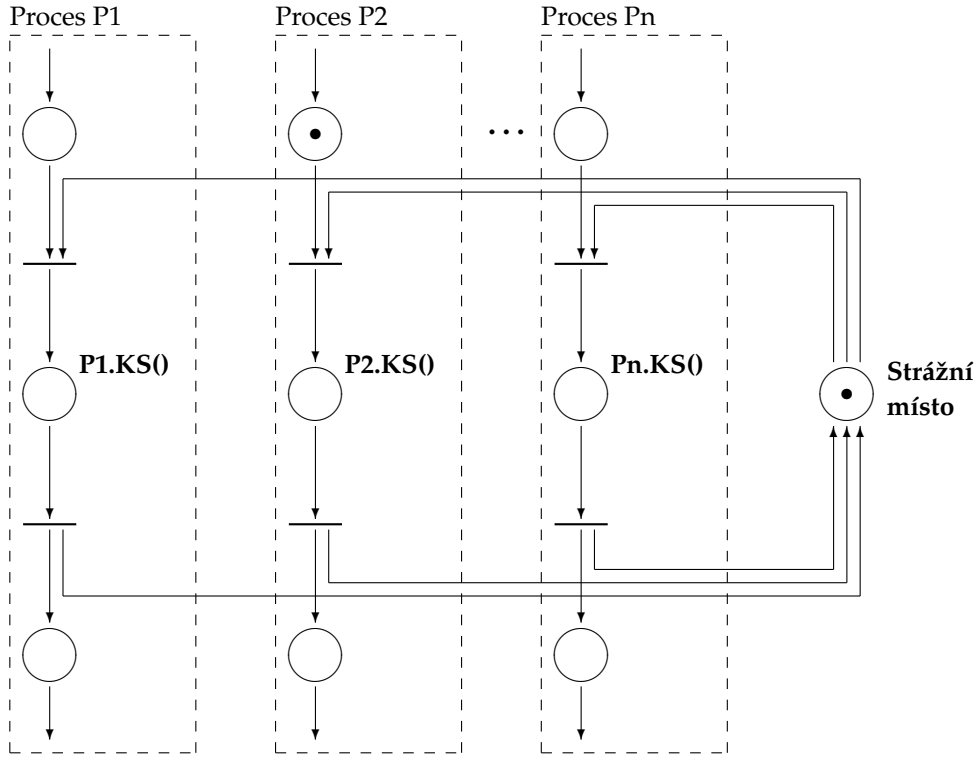
a přechody mohou představovat podsítě, jejichž výpočet a stavy nepotřebujeme rozlišovat.

5.3 Základní synchronizační úlohy

Postupně probereme základní úlohy, které se řeší při synchronizaci procesů, a naznačíme jejich řešení na abstraktní úrovni pomocí petriho sítí.

5.3.1 Kritická sekce

Tuto úlohu je třeba vyřešit, pokud chceme umožnit výlučný přístup ke sdílenému prostředku – kritické sekci (například sdílenému místu v paměti). Je třeba zajistit, aby k tomuto prostředku v jednom okamžiku přistupoval nejvýše jeden proces a aby tento prostředek mohl bez přerušení využívat po potřebnou nebo předem stanovenou dobu. Na obrázku 5.3 je problém ukázán na petriho síti.



Obrázek 5.3: Petriho síť pro úlohu Kritická sekce

Aby proces mohl provést svou část kódu přistupující ke kritické sekci, musí být ve strážním místě tečka. V našem případě k přechodu pro vstup do kritické sekce

přichází proces P2, a protože je ve strážním místě tečka, znamená to, že ke sdílenému prostředku zrovna nepřistupuje jiný proces a tedy P2 může dál pokračovat.

Po vyhodnocení vstupního přechodu je odebrána tečka nejen z místa procesu před kritickou sekcí, ale také ze strážního místa, a přidána do místa uvnitř vyhodnocení kritické sekce procesem P2. Pak je proces ve stavu vyhodnocování kritické sekce, v místě P2.KS(). Po provedení přechodu znamenajícího opuštění kritické sekce je tečka vrácena do strážního místa a také přidána do místa procesu P2 za kritickou sekcí, tedy proces pokračuje ve své činnosti a do kritické sekce může vstoupit další proces.

Kdyby další proces chtěl vstoupit do kritické sekce v době, kdy se v ní nachází proces P2 (a tedy ve strážním místě není tečka), musí počkat, dokud proces P2 nevrátí tečku do strážního místa, a teprve potom pokračovat.

Požadavky na řešení jsou:

- v kritické sekci smí být nejvýše jeden proces,
- proces čeká na vstup do kritické sekce konečnou dobu,
- proces se nachází v kritické sekci konečnou dobu.

5.3.2 Producent–konzument

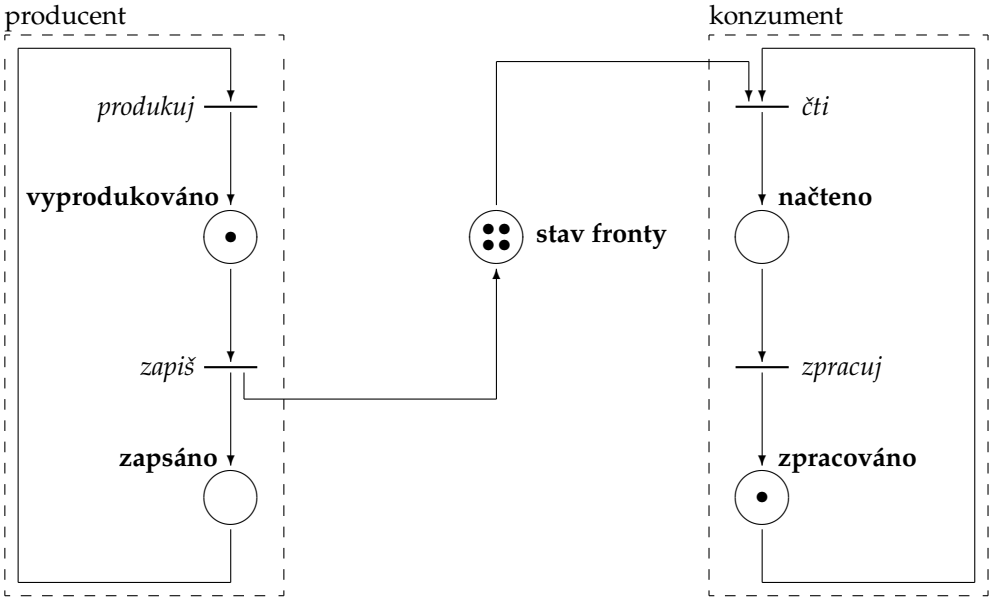
Producent je proces produkující data a *konzument* je proces, který tato data přijímá a dále zpracovává. Účelem je, aby producent a konzument (konzumenti) mohli pracovat každý jinou rychlostí, do určité míry na sobě nezávisle, tedy obvykle *asynchronně*.

Úlohu lze řešit několika způsoby v závislosti na tom, kolik máme k dispozici sdílené paměti, do které mají přístup všechny zúčastněné procesy:

1) Neomezený buffer – máme k dispozici jakékoliv množství paměti.
2) Omezený buffer – máme k dispozici určitý počet paměťových míst, je stanovena horní hranice.
3) Synchronizace zprávami – žádná sdílená paměť, nutnost synchronní komunikace.

ad. 1) Neomezený buffer: řešení je naznačeno petriho sítí na obrázku 5.4.

Máme k dispozici frontu položek, jejíž délka se dynamicky mění, požadavkem je zajistit, aby se konzument zastavil ve chvíli, kdy je fronta prázdná, a aby data za každých okolností zůstala konzistentní. Na obrázku 5.4 je systém ve stavu, kdy



Obrázek 5.4: Petriho síť pro úlohu Producent - konzument, neomezený buffer

ve frontě jsou čtyři položky a jsou proveditelné přechody *zapiš* a *čti* (tedy pracovat mohou oba procesy).

ad. 2) Omezený buffer: řešení je naznačeno petriho sítí na obrázku 5.5. Omezený buffer může být implementován jako statická kruhová fronta.

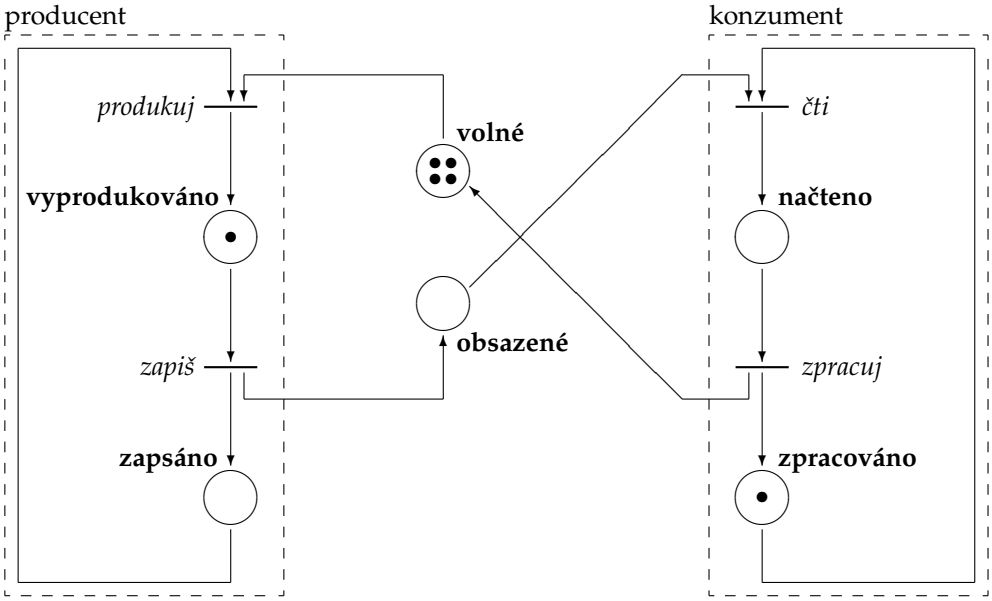
Oproti řešení neomezeným bufferem zde jsou tři požadavky:

- producent se zastaví, když je buffer plný,
- konzument se zastaví, když je buffer prázdný,
- data musí být neustále v konzistentním stavu.

Na obrázku 5.5 je proveditelný pouze přechod *zapiš*, konzument musí čekat před přechodem *čti*. Po provedení přechodu *zapiš* budou proveditelné přechody *produkuj* a *čti*. Celkový počet míst ve frontě je součet teček v místech **volné**, **obsazené**, **vyprodukováno** a **načteno**.

ad. 3) Synchronizace zprávami: tato metoda je použitelná v případě, že procesy nemohou sdílet žádnou paměť, například v distribuovaných systémech nebo v případě víceprocesorových systémů bez společné paměti.

Producent a konzument si navzájem posílají zprávy, producent posílá položky a konzument potvrzení o zpracování (žádost o další položku). Jedná se tedy o sy-



Obrázek 5.5: Petriho síť pro úlohu Producent - konzument, omezený buffer

metrickou synchronní komunikaci. Řešení petriho sítí je na obrázku 5.6 na str. 67, procesy na sebe navzájem čekají.

5.3.3 Model–obraz

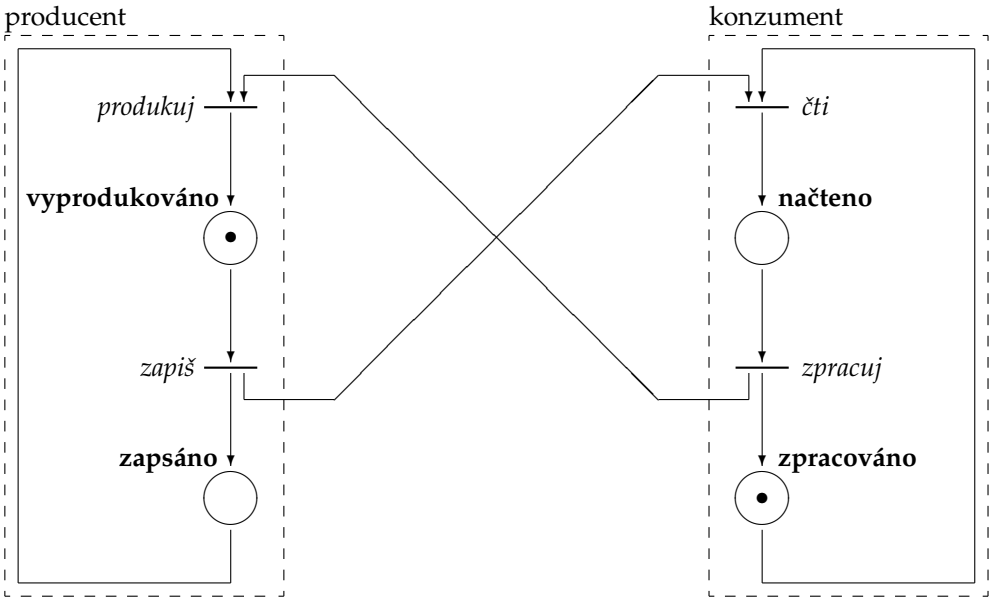
Tato úloha je podobná úloze Producent–konzument. Řešíme ji, když je potřeba sledovat a zpracovávat nikoliv všechny položky, ale vždy právě aktuální stav.

Jedna skupina procesů (nebo vláken) neustále provádí změny na datech a další skupina procesů (vláken) zobrazuje aktuální stav těchto dat. Typickým problémem je zobrazování stavu nějakého zařízení (teplota, vlhkost, apod.), kdy producenti jsou procesy sledující čidla měřící tyto veličiny.

Kdybychom trvali na zpracování všech položek, které produkující procesy vytvoří, zpracování by se zdržovalo a případné zobrazování dat na monitoru by mohlo „problikávat“ nebo by se tak rychle měnilo, že by údaje byly nečitelné. Úlohy tohoto typu jsou typické také pro realtimové systémy.

Producent může označovat pozměněná místa, pak se konzument soustřeďuje pouze na tato místa a při čtení jejich označení odstraňuje. V případě, že producent mění místo, které ještě nebylo konzumentem načteno a tedy je označeno jako pozměněné, pak nemusí nic poznamenávat, jen změní data.

Hlavním požadavkem je zachování konzistence dat, tedy k danému paměťovému místu nesmí přistupovat více než jeden proces. Řešení odpovídá řešení kritické sekce.



Obrázek 5.6: Petriho síť pro úlohu Producent - konzument, synchronizace zprávami

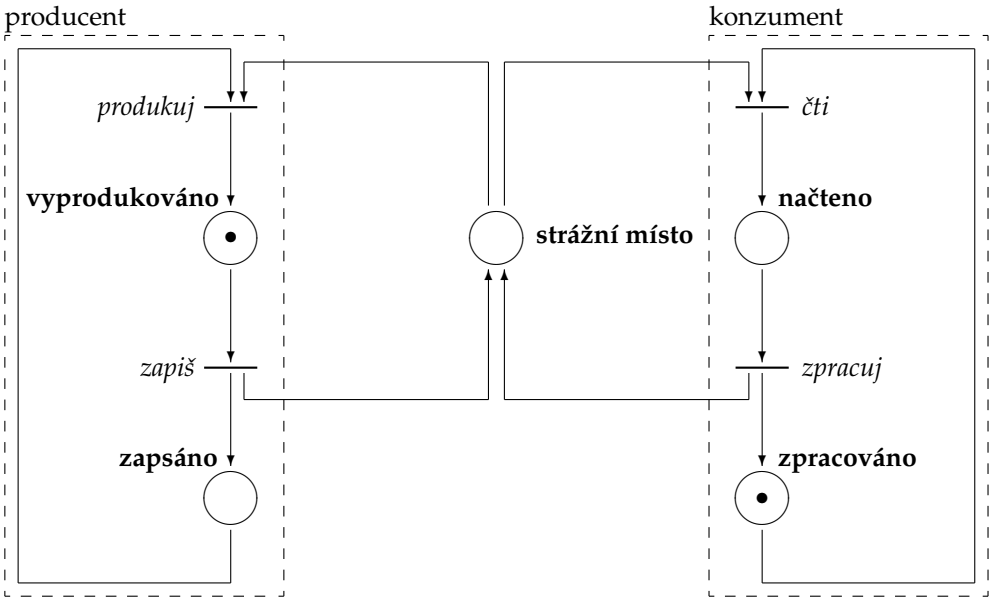
Na obrázku 5.7 je případ jednoho producenta a jednoho konzumenta, kteří přistupují k paměťovému místu určujícímu momentální stav dat (*strážní místo*). Pokud je v strážním místě tečka, znamená to, že data jsou v konzistentním stavu a právě k nim nepřistupuje producent ani konzument, v našem případě k datům přistupuje producent, který ze strážního místa tečku odebral. Každý proces může pracovat jiným tempem.

5.3.4 Čtenáři–písaři

V této úloze jsou procesy rozděleny vzhledem k přístupu ke sdílenému prostředku (paměti) do dvou disjunktních skupin – skupiny čtenářů a skupiny písařů (proces může přecházet mezi skupinami, nesmí však být v obou). Čtenáři zde mohou číst, písaři mohou zapisovat. Musíme mít neustále přehled o tom, kolik je čtenářů (čtoucích procesů).

V době, kdy některý proces zapisuje, nesmí probíhat žádné čtení ani zápis jiným procesem, zatímco operací čtení může probíhat více zároveň (nenarušují konzistentnost dat). Požadavky na řešení jsou následující:

- data musí zůstat v konzistentním stavu,
- operace zápisu je vyloučena s jakoukoliv jinou operací (čtení i zápisu),
- operace čtení nejsou navzájem vyloučeny.



Obrázek 5.7: Petriho síť pro úlohu Model - obraz

Na obrázku 5.8 na str. 69 je úloha řešena pro čtyři čtenáře a jednoho písaře. Každý čtenář si před čtením vyzvedne tečku ze strážního místa. Zapisující proces (písař) si při pokusu o zápis musí vyzvednout čtyři tečky, tedy tolik, kolik je celkem čtenářů. Tím je zajištěno, že zapisovat lze pouze tehdy, když žádný čtenář nečte (ve strážním místě jsou všechny tečky). Pokud některý písař zapisuje, musí všichni čtenáři počkat, až písař dokončí zápis a vrátí tečky do strážního místa.

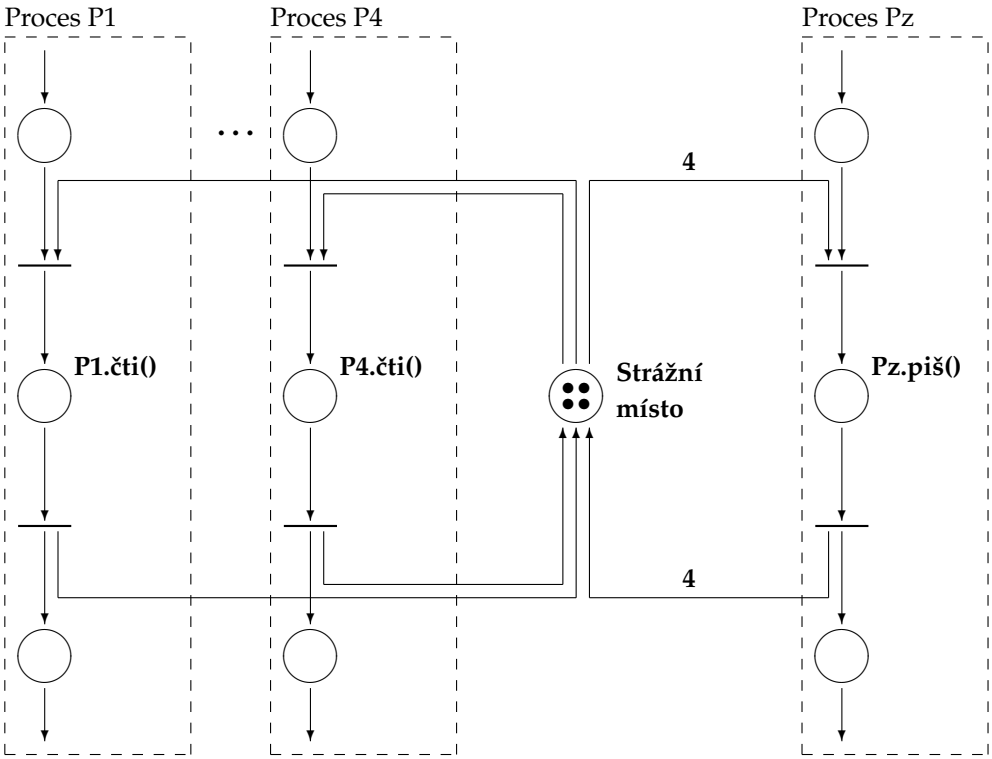
Kdyby bylo více písařů, opět by kterýkoliv písař byl zablokován jak v případě, že některý čtenář čte, tak i v případě, že některý jiný písař zapisuje.

5.3.5 Pět hladových filozofů

Je to typická úloha paralelního programování. Název úlohy je odvozen ze známého problému: u kulatého stolu sedí pět filozofů a střídavě přemýšlí a jí. Každý k jídlu potřebuje dvě hůlky, ale na stole je pouze pět hůlek, mezi každou sousedící dvojicí filozofů jedna. Pokud zrovna filozof nemá k dispozici hůlku po pravé i levé ruce, nezbývá mu než přemýšlet.

Filozof, jehož sousedé mu střídavě berou hůlky, nemá šanci se najíst, dochází ke stárnutí. Pokud všichni najednou zvednou hůlku po své pravé ruce, dojde k uváznutí, protože všichni drží v pravé ruce hůlku a čekají na levou, která je však zrovna držena levým sousedem, a tedy nedostupná.

Po aplikaci na procesy máme pět (obecně n) procesů využívajících určité prostředky (hůlky), o které se dělí s jinými.



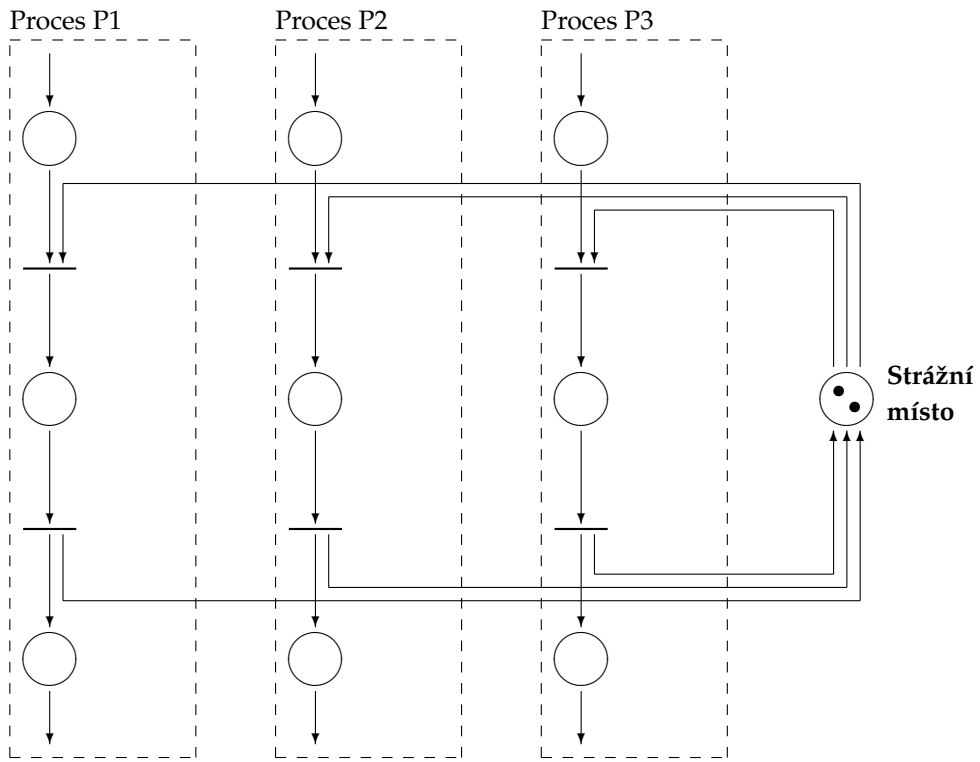
Obrázek 5.8: Petriho síť pro úlohu Čtenáři - písaři

Řešení problému spočívá v tom, že ke stolu nepustíme všech pět filozofů najednou (a tedy k prostředkům nepustíme všechny procesy najednou), ale maximálně čtyři ($n-1$, případně $k-1$ pro $k > 0$, ke k je počet sdílených prostředků). Důsledkem je, že alespoň jeden filozof se nají v každém případě, tedy i kdyby všichni najednou vzali hůlku pravou (nebo levou) rukou, u procesů alespoň jeden proces může použít potřebné prostředky a uvolnit pak místo dalšímu procesu. Jinou možností je nařídit jednomu filozofovi, aby bral hůlky v opačném pořadí než ostatní.

Na obrázku 5.9 je řešení naznačeno na skupině tří procesů a tří prostředků. Je dovoleno najednou pracovat pouze dvěma procesům, aby mohly využít ty prostředky, které potřebují.

5.3.6 Souběh procesů

Úloha je řešena v paralelním systému, kdy dva procesy běžící na různých procesorech (a tedy souběžné) je třeba sesynchronizovat tak, aby určitou část kódu prováděly společně.



Obrázek 5.9: Petriho síť pro úlohu Pět filozofů (pro tři procesy a tři prostředky)

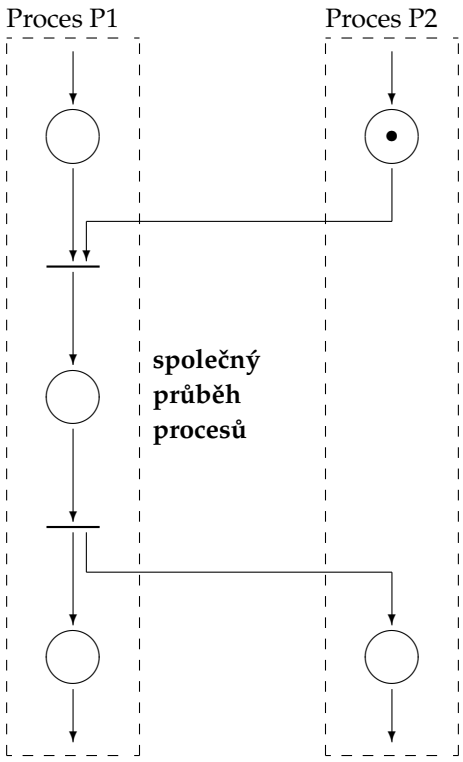
Pokud se jeden proces ke společné části kódu dostane dřív než druhý, musí počkat (na obrázku 5.10 na str. 71 musí čekat proces P2), tento kód lze provést až ve chvíli, kdy k přechodu na začátku společné části dospějí oba procesy.

Tato metoda synchronizace procesů se může používat například v mechanismu RPC (volání vzdálené procedury, na obrázku 5.10 volá proces P2 proceduru procesu P1) nebo při jakékoliv synchronní výměně dat paralelních procesů.

5.4 Implementace čekání před kritickou sekcí

Jak vyplývá z popisu řešení synchronizačních úloh pomocí petriho sítí, procesy musí trávit určitou dobu čekáním. Toto čekání lze implementovat různými způsoby, které můžeme rozdělit do dvou skupin:

- pasivní čekání,
- aktivní čekání.



Obrázek 5.10: Petriho síť pro úlohu Souběh procesů

Pasivní čekání: proces je blokován nebo suspendován, sám se nepodílí na čekání. Zde můžeme zařadit tyto metody:

1. *Zákaz přerušení* (maskování přerušení) – proces, který využívá daný prostředek, zakáže přerušení a tím znemožní přepínání kontextů, procesor nemůže být přidělen jinému procesu (tedy ani žádnému z čekajících). Jde o hardwarové řešení použitelné pouze na jednoprocesorovém systému.

   Nevýhodou je, že ne všechna přerušení lze zakázat (například přerušení při dělení nulou) a navíc přerušení vygenerovaná během zákazu přerušení se mohou ztratit (není vyvolána funkce ošetřující toto přerušení). Důsledkem je kromě jiného i to, že proces, který zakázal přerušení, nelze obvyklým způsobem ukončit ani přerušit, může dojít k uváznutí a stárnutí procesů. Toto řešení snižuje propustnost systému.

2. *Mutex* (mutual exclusion, vzájemné vyloučení) na úrovni jádra (operace Lock, zámek) – operační systém může nabízet systémové volání zakazující přepnutí kontextu. Oproti předchozímu řešení se neztrácejí přerušení (jsou obvykle určena běžícímu procesu), navíc je to softwarové řešení na úrovni operačního

systému, tedy není hardwarově závislé. Nevýhoda nebezpečí uváznutí a stárnutí procesů zůstává.

Aktivní čekání: proces se podílí na čekání, obvykle jde o některou variantu cyklu s prázdnou operací.

1. *Sdílená zamykací proměnná* – proměnná `obsazeno` může být nastavena na 0 (false, volno) nebo 1 (true, obsazeno). Pokud je nastavena na 1, proces provádí prázdnou smyčku.

```
shared int obsazeno = 0;
...
while(obsazeno) {};
obsazeno = 1;
KS();
obsazeno = 0;
```

   V kritické sekci se může nacházet pouze jeden proces, ale není vyřešena podmínka konečnosti průběhu kritické sekce ani konečnost čekání ostatních procesů (záleží na tom, v jakém pořadí se provádí test `while(obsazeno)` čekajících procesů, do kritické sekce se dostane jednoduše ten proces, jehož test se provádí jako první po nastavení proměnné `obsazeno` na 0, což je do určité míry náhoda).

2. *Střídání procesů* – prostředek je střídavě přiřazován všem procesům. Pokud proces prostředek zrovna nepotřebuje, zbytečně zdržuje ostatní procesy, tedy metoda snižuje propustnost systému a dochází k uváznutí. Řešení je použitelné v případě, kdy máme jen málo procesů (pokud možno dva), což je v operačním systému velmi nepravděpodobné. Pro i-tý proces:

```
shared int pridelen = 0;
...
while(pridelen != i) ;
KS();
pridelen ++;
if(pridelen == n)
  pridelen = 0;
```

   Malou změnou lze odstranit uváznutí v případě, že některý proces nechce prostředek využívat – přidáme pole hodnot 0, 1 (false, true), kde pro každý

proces bude jedna položka. V tomto poli dá proces na vědomí, že chce využívat daný prostředek, nastavením své položky na 1, a místo přidělení následujícímu procesu se ve smyčce přeskočí ty procesy, které o prostředek nestojí. Pro i-tý proces:

```
shared int pridelen = 0;
shared int priznaky[n] = (0,0,...,0);
...
priznaky[i] = 1;
while(pridelen != i) {};
KS();
priznaky[i] = 0;
do {
  pridelen ++;
  if(pridelen == n)
    pridelen = 0;
} while(!priznaky[pridelen]);
```

I po této úpravě však dochází ke zbytečnému zdržování časovou režií přiřazování prostředku.

3. *Bakery algorithm* (pekařův algoritmus) – procesu je při žádosti o přístup do kritické sekce přiděleno pořadové číslo. Přednost má proces s nejnižším pořadovým číslem, a v případě, že dva procesy mají stejné pořadové číslo, rozhoduje se mezi nimi podle dalšího kritéria (PID nebo porovnání jmen procesů podle abecedy).

```
shared int poradi[n] = (0,0,...,0);
shared int prirazuje[n] = (0,0,...,0);
...
prirazuje[i] = 1;
poradi[i] = nejvyssi(poradi)+1;
prirazuje[i] = 0;
for(j=0; j<n; j++) {
  while(prirazuje[j]) {};
  while((!poradi[j])        % poradi[j]!= 0
        &&((poradi[j]<poradi[i])
           ||((poradi[j]==poradi[i])&&(j<i)) )) {};
}
KS();
poradi[i] = 0;
```

Algoritmus pracuje takto: v poli `poradi` má každý proces čekající na prostředek přiřazeno pořadové číslo (pokud o prostředek nežádá, je zde číslo 0), v poli `prirazuje` je u daného procesu číslo 1, pokud zrovna probíhá přiřazování pořadí pro tento proces, po provedení přiřazení je hodnota vrácena zpět na 0. Tím je zajištěno, že sice je možné přidělit dvěma procesům stejné pořadí, ale během tohoto přiřazování, kdy číslo není „konzistentní “, není zjišťována hodnota tohoto čísla jiným procesem (čekání `while((prirazuje[j])`, tedy dokud je daná hodnota rovna 1).

Čekající proces pak prochází pole `poradi`, a pokud narazí na proces, jehož pořadové číslo je menší (nebo stejné, ale má nižší PID), pak v prázdném cyklu ({}) čeká, až tento proces ukončí čekání a zpracuje svou část kódu v kritické sekci. Když proces takto projde celé pole, pak už žádný jiný proces nemá nižší pořadové číslo (žádný z nich už nečeká na tento prostředek), a proto i tento proces může provést kód kritické sekce. Pak nastaví své pořadí na 0 a tím dá najevo, že už prostředek nepoužívá.

Pekařův algoritmus je použitelný i pro víceprocesorové systémy. Je to jednoznačný a přitom jednoduchý algoritmus, který zamezuje stárnutí procesů. Je použitelný například tehdy, když je procesor plánován metodou FCFS.

4. *Hardwarové řešení* – některé procesory nabízejí hardwarové řešení pro aktivní čekání, instrukci `TSL` (Test and Set Lock), `swap` nebo `XCHG`. Všechny tyto instrukce nějakým způsobem provádějí výměnu hodnoty zámku kritické sekce a dané proměnné. Používají se tak, že neustále nastavují zámek na 1 (zamčeno) a zjišťují, jaká byla původní hodnota před tímto nastavením. Mohou být použity jako součást složitějšího prostředku aktivního čekání.

5.5 Synchronizační nástroje operačního systému

Prostředky popsané výše v kap. 5.4 jsou většinou buď hardwarově závislé nebo se implementují na straně procesu, což omezuje možnosti jejich použití. Operační systémy ve svém jádře obvykle nabízejí komplexnější synchronizační nástroje dostupné pomocí systémových volání. Jsou to především tyto nástroje:

- semafory,
- mechanismus zpráv,
- monitory,
- volání vzdálené procedury (RPC).

5.5.1 Semafory

Semafory povolují nebo zabraňují přístupu do kritické sekce. Semafor je obvykle implementován strukturou obsahující proměnnou se stavem semaforu a další proměnnou pro implementaci fronty procesů. Čekající procesy jsou blokovány (suspendovány) a zařazeny do této fronty, tedy jde o pasivní čekání, které je zajišťováno operačním systémem. Fronta může být klasická FIFO nebo s prioritami. Pro řešení úloh z kap. 5.3 se používá jeden nebo více semaforů. Existují dva typy semaforů – binární a obecné.

Semafor je kromě inicializační funkce obsluhován dvěma funkcemi:

- funkci `wait` (příp. `down`, `lock`) spouští proces žádající o vstup do kritické sekce; pokud do kritické sekce nelze vstupit, je v rámci této funkce proces blokován (suspendován) a zařazen do fronty, jinak funkce končí bez tohoto důsledku,
- funkci `signal` (příp. `send`, `up`, `unlock`) spouští proces vystupující z kritické sekce a slouží k vybrání následujícího procesu z fronty a jeho poslání do kritické sekce, tedy ukončí jeho blokování a zařadí do fronty připravených.

Proces nejdřív zavolá funkci `wait` (a případně je blokován), pak provede kód kritické sekce a potom zavolá funkci `signal` povolující dalšímu prostředku přístup do kritické sekce. Posloupnost operací pro proces žádající o vstup do kritické sekce a daný semafor je následující:

```
...
wait(semafor);
KS();
signal(semafor);
...
```

Funkce `wait` a `signal` by měly být atomické (nedělitelné, nepřerušitelné), a také k frontě semaforu by měl být výhradní přístup (při přidávání nebo vybírání z fronty)[1].

[1]Nepřerušitelnost funkcí wait a signal lze jednoduše zařídit na jednoprocesorovém systému (například zakázáním přerušení během vykonávání kódu funkce), ale ve víceprocesorovém systému tuto jednoduchou metodu nelze použít. Obvykle se tento problém řeší označením těchto funkcí samotných za kritické sekce a jejich softwarovým řešením.

Taktéž výhradní přístup k frontě se těžko řeší, a to v případě, kdy je fronta dynamická. Proto se v takovýchto případech (sdílená paměť) používají spíše statické paměťové struktury přes řadu jejich nevýhod oproti dynamickým.

Binární semafor má dva stavy: 0 (červená) pro zákaz vstupu, 1 (zelená) pro povolení vstupu. V těchto stavech se reaguje takto:

0 (červená): Pokud nyní některý proces spustí funkci `wait`, je blokován a zařazen do fronty semaforu.

Při volání funkce `signal` se zkontroluje, zda některý proces čeká ve frontě. Jestliže ano, je první čekající proces zařazen do fronty připravených a tím je mu povoleno vstupit do kritické sekce, když je fronta prázdná, semafor se pouze nastaví na 1.

1 (zelená): Na proces volající funkci `wait` tato funkce nemá žádný vliv, jen nastaví semafor na 0. Funkce `signal` v tomto stavu není volána.

Obecný semafor (proměnná zachycující jeho stav, čítač) může nabývat různých celočíselných hodnot.

```
typedef struct TSemafor {
  int stav;
  TFrontaProcesu fronta;
}
extern struct TSemafor semafor;
```

Z hlediska procesu se obecné semafory používají stejně jako binární, vnitřně jsou však implementovány jinak a oproti binárním semaforům uchovávají další informace navíc. Obvykle záporná hodnota semaforu určuje, kolik procesů čeká ve frontě, kladná naopak znamená, že semafor je „předplacen“, určuje, kolikrát je možno kolem semaforu projít bez čekání.

Obecné semafory se používají ve dvou variantách:

a) čítač nabývá hodnot ≥ 0 (nezáporných) – oproti binárnímu přidává jen funkci „předplacení“. Funkce `wait` a `signal` jsou implementovány následovně:

- `wait` při hodnotě čítače semaforu > 0 sníží hodnotu čítače o 1 a ukončí se (proces není blokován), při hodnotě $= 0$ zablokuje proces a zařadí ho do fronty.
- `signal` zkontroluje frontu. Když je fronta prázdná, zvýší čítač o 1, a když není prázdná (tedy čítač je $= 0$), odblokuje první čekající proces a pošle ho do fronty připravených.

b) čítač nabývá i záporných hodnot – funkce `wait` a `signal` jsou implementovány takto:

- `wait` sníží čítač o 1. Pokud před tímto snížením byl čítač ≥ 0 (žádný čekající proces), hned se ukončí (a proces může pokračovat do kritické sekce), když byl čítač < 0, zablokuje proces a zařadí ho do fronty.
- `signal` zvýší čítač o 1. Pokud byl čítač před zvýšením < 0 (po zvýšení ≤ 0), pak zkontroluje frontu; pokud fronta není prázdná, první čekající proces odblokuje a pošle do fronty připravených.

Použití obecných semaforů si ukážeme na řešení synchronizační úlohy Producent - konzument s omezeným bufferem. Potřebujeme dva semafory:

- semafor `volne` zakazuje producentovi překročit velikost bufferu, iniciujeme ho na počet položek, které se vejdou do sdílené paměti (tedy „předplatíme“),
- semafor `obsazene` zakazuje konzumentovi vybírat z bufferu, když je zrovna prázdný, iniciujeme ho na 0.

Producent:

```
do {
  produkuj(data);
  wait(volne);
  zapis(data);
  signal(obsazene);
} while(1);
```

Konzument:

```
do {
  wait(obsazene);
  cti(data);
  signal(volne);
  zpracuj(data);
} while(1);
```

Další příklad je řešením úlohy Pět hladových filozofů. Pro N prostředků máme N kritických sekcí, tedy budeme mít pole N semaforů. Všechny iniciujeme na 1. Další semafor bude hlídat, aby prostředky využívalo pouze N-1 procesů (je inicializován na N-1). Následující kód je pro i-tý proces:

```
#define  N  5

struct TSemafor sem[N];
struct TSemafor S;
for(i=0; i<N; i++)
  sem[i].stav = 1;
S.stav = N-1;
...
```

```
do {
  mysli();
  wait(S);
  wait(sem[i]);
  wait(sem[(i+1)%5]);
  jez();
  signal(sem[(i+1)%5]);
  signal(sem[i]);
  signal(S);
} while(1);
```

5.5.2 Mechanismus zpráv

Procesy mohou být synchronizovány také mechanismem zpráv. Pod tímto pojmem rozumíme nejen přímé zasílání zpráv (`send` a `receive`), ale také nepřímou komunikaci posílání zpráv přes porty (sockety).

Metoda je vhodná i pro víceprocesorové prostředí včetně synchronizace v rámci počítačové sítě. Pak se při adresaci udává také adresa počítače v síti (uzel), protože PID procesu nemusí být v rámci sítě jedinečné. Používá se tedy adresování přímé (uzel.PID) nebo nepřímé (uzel.číslo_portu, IP_adresa.číslo_portu, atd.).

U nepřímého adresování bývá port představován sdílenou pamětí – *schránkou* – pevné nebo proměnné délky.

Procesy se zprávami pracují pomocí funkcí (systémových volání) obvykle nazvaných `send` a `receive`. Při použití přímého adresování komunikace funguje výše popsaným způsobem (kap. 4.6), u nepřímého adresování tyto funkce pracují následovně:

- funkce `send` zkontroluje, zda schránka není plná; pokud není plná, odesílající proces pošle zprávu, jinak je proces zablokován a zpráva je poslána až po jeho odblokování (po uvolnění místa ve schránce),
- funkce `receive` volaná adresátem zkontroluje, zda schránka není prázdná; pokud není prázdná, přijme zprávu, jinak je proces zablokován až do doby, kdy je do schránky doručena zpráva, a ta je pak přijata.

Následující příklad je řešením úlohy Producent - konzument pro buffer pevné délky. Jsou definovány dvě schránky:

- `volne` – když se producentovi podaří z této schránky získat zprávu, může produkovat, na začátku je celá naplněna zprávami informujícími o možnosti produkovat, konzument zde zašle zprávu po každém zpracování položky,
- `obsazene` – zde producent zasílá zprávy s vyprodukovanými položkami.

```
#define  MAX  20
struct TSchranka volne, obsazene;
for(i=0; i<MAX; i++) send(volne,NULL);
```

Producent:

```
do {
  receive(volne,data);
  produkuj(data);
  send(obsazene,data);
} while(1);
```

Konzument:

```
do {
  receive(obsazene,data);
  zpracuj(data);
  send(volne,NULL);
} while(1);
```

Pokud má schránka velikost 0, jde vlastně o variantu přímého adresování a tento případ se nazývá *dostaveníčko* (randez-vous). Samotné volání funkce `send` nebo `receive` způsobí zablokování volajícího procesu, proces je odblokován až tehdy, když je volána párová funkce, tedy až když jsou volány obě funkce `send` a `receive`, může komunikace proběhnout (vzájemné čekání). Následující kód je řešením úlohy Producent - konzument bez sdílené paměti. Oproti předchozím řešením musí být komunikace synchronní.

Producent:

```
do {
  produkuj(data);
  send(konzument,data);
  receive(konzument,ok);
} while(1);
```

Konzument:

```
do {
  receive(producent,data);
  zpracuj(data);
  send(producent,ok);
} while(1);
```

5.5.3 Monitory

Monitor je synchronizační prostředek na vyšší úrovni. Zapouzdřuje v sobě skupinu datových struktur, procesy k nim mohou přistupovat pouze přes rozhraní určené přístupovými funkcemi. Funkcí může být jakýkoliv počet a určují různé způsoby práce s daty monitoru.

Datové struktury zapouzdřené v monitoru bývají označovány jako *podmínky*. Tyto podmínky mohou být implementovány pomocí semaforů a semaforové operace `wait` a `signal` jsou používány přístupovými funkcemi monitoru (nikoliv procesy), každá přístupová funkce využívá jednu nebo více různých podmínek.

Jde o to, aby každá podmínka mohla být v jednom okamžiku využívána pouze jedním procesem (jedinou funkcí). Proto přístupová funkce okamžitě po svém spuštění zavolá funkci `wait` každé podmínky, kterou bude používat, a tím zabrzdí spuštění kterékoliv další funkce, která by tuto podmínku také chtěla používat. Těsně před svým ukončením pak funkce opět rezervované podmínky odblokuje jejich funkcemi `signal`.

5.5.4 RPC

RPC (Remote Procedure Call, volání vzdálené procedury) rozumíme postup, kdy proces volá proceduru jiného procesu. Může se jednat nejen o volání procedury (funkce) naprogramované v spustitelném souboru, ale také o proceduru (funkci) nacházející se v knihovně (včetně knihoven API rozhraní operačního systému).

RPC se obvykle realizuje pomocí synchronních zpráv, kdy žádající proces pošle procesu - majiteli dotyčné procedury zprávu obsahující také případné skutečné parametry pro danou proceduru a čeká na odpověď. Druhý proces obdrží zprávu, vyvolá proceduru a volajícímu procesu odešle zprávu s výsledkem provedení procedury.

KAPITOLA 6

Uváznutí procesů (Deadlock)

K uváznutí procesů dochází, když některý proces čeká na prostředek, který je přidělen jiným čekajícím procesům.

Uváznutí je samozřejmě nežádoucí, proto je vhodné této situaci předcházet nebo, pokud nastane, ji řešit co nejšetrněji vzhledem k systému i procesům.

6.1 Základní pojmy

Pokud proces chce používat prostředek, musí o tento prostředek nejdřív *požádat*. Jeho žádost může být vyplněna, a potom je procesu tento prostředek *přidělen*, proces ho může *používat* v rámci jeho možností a bezpečnostních opatření, když už proces tento prostředek nepoužívá, měl by ho *uvolnit*. Pokud žádost procesu o prostředek z nějakého důvodu nemůže být vyplněna, proces *čeká* na prostředek.

Prostředky rozdělíme do *tříd*, v jedné třídě mohou být pouze prostředky navzájem zaměnitelné, tedy proces bude vždy žádat prostředek z určité třídy a může mu být jedno, který konkrétní prostředek z této třídy dostane. Konkrétní prostředky z určité třídy budeme nazývat *instance*.

Například třída *operační paměť* má jako instance jednotlivé bloky (stránky, segmenty) paměti, třída *tiskárny* obsahuje různé tiskárny, které jsou v „rozumné“ vzdálenosti od daného uživatele a tedy zaměnitelné, třídě *čas CPU* příslušejí jako instance časové cykly procesoru, které mohou být procesům přidělovány, . . .

Množina procesů je ve stavu uváznutí, pokud každý proces v této množině čeká na událost, kterou může vyvolat pouze některý z procesů v téže množině. Jedná se

zde především o událost uvolnění prostředku používaného některým procesem, případně některé typy komunikace (čekání na potvrzení zprávy).

Problém uváznutí se řeší buď některou metodou předcházení uváznutí nebo předpovídání uváznutí, nebo se neřeší vůbec a nanejvýš jsou implementovány postupy zjištění uváznutí.

6.2 Popis stavu přidělení prostředků

Stav přidělení prostředků lze popsat *grafem přidělení prostředků*. Je to orientovaný graf, jehož vrcholy jsou dvojího druhu:

- procesy – každý proces systému má zde svůj vrchol, tyto vrcholy mají tvar kruhový,
- prostředky – každá třída prostředků má zde svůj vrchol tvaru obdélníku, v něm je pro každou instanci třídy (tj. konkrétní prostředek) jedna tečka.

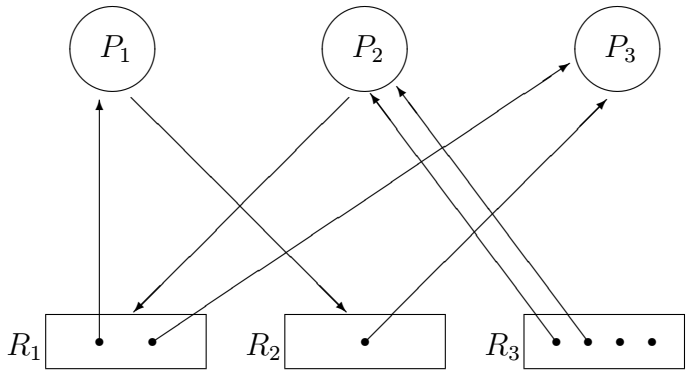
Orientované hrany jsou také dvojího druhu:

- hrana žádosti o prostředek vede od procesu, který žádá o prostředek, k vrcholu třídy prostředku, o který žádá,
- hrana přidělení prostředku vede od instance třídy prostředku (tedy od tečky ve vrcholu třídy) k procesu, kterému byl prostředek přidělen.

Příklad: V systému jsou procesy P_1, P_2 a P_3 a prostředky R_1 se dvěma instancemi, R_2 s jednou instancí a R_3 se čtyřmi instancemi. Proces P_1 má přidělenu jednu instanci prostředku R_1 a žádá o prostředek R_2, proces P_2 má přiděleny dvě instance prostředku R_3 a žádá o prostředek R_1, proces P_3 má přidělenu jednu instanci prostředku R_1 a jednu instanci prostředku R_2 a momentálně nežádá o žádné další prostředky.

Co z grafu na obrázku 6.1 můžeme vyčíst? Pokud v grafu není žádná kružnice, žádný proces není blokován čekáním na prostředek. Jestliže je v grafu nějaká kružnice, může, ale nemusí dojít k uváznutí. K uváznutí v případě kružnice dojde, pokud každá třída prostředků na kružnici má právě jednu instanci.

V grafu není žádná kružnice, proto žádný proces neuvázl. Kdyby prostředek třídy R_2 byl přidělen procesu P_2 místo procesu P_3, v grafu by byla kružnice $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$, ale nedošlo by k uváznutí, protože až proces P_3 nebude potřebovat přidělenou instanci prostředku R_1, uvolní ji a tato instance může být přidělena procesu P_2, který o tento prostředek žádá, a tím je kružnice odstraněna



Obrázek 6.1: Graf přidělení prostředků

(hrana žádosti o prostředek se změní na hranu přidělení prostředku, která má opačnou orientaci).

Kdyby ale při situaci v grafu na obrázku 6.1 proces P_3 požádal o další prostředek R_1, vzniklá kružnice by znamenala uváznutí, protože všechny instance prostředků patřících do kružnice drží právě uvázlé procesy P_1 a P_3, žádný z nich nemůže být uvolněn.

6.3 Podmínky vzniku uváznutí

K uváznutí dojde, pokud jsou splněny všechny následující podmínky:

1. Existence prostředků, které nelze sdílet (prostředků, které v jednom okamžiku může používat nejvýše jeden proces).

2. Existuje alespoň jeden proces, který má přidělen nějaký prostředek a čeká na přidělení jiného prostředku, který je přidělen jinému procesu.

3. Při správě prostředků je používáno nepreemptivní plánování, tedy k uvolnění přidělených prostředků dochází pouze ze strany procesů, prostředky nejsou „násilně“ odebírány.

4. Dojde ke *kruhovému čekání*, tedy existuje taková posloupnost různých procesů $P_0, P_1, \ldots, P_n, P_{n+1}$, že každý proces P_i čeká na prostředek přidělený procesu P_{i+1} v této posloupnosti, $0 \leq i \leq n$), $P_{n+1} = P_0$ (jinými slovy: každý proces čeká, až ten, který je v posloupnosti za ním, uvolní určitý prostředek, poslední je totožný s prvním).

6.4 Prevence uváznutí

Účelem je zajistit, aby nemohla nastat některá z podmínek vzniku uváznutí (stačí, když nemůže nastat jedna z nich, protože k uváznutí dojde pouze tehdy, když nastanou všechny zároveň). Postupně probereme jednotlivé podmínky.

Prostředky, které nelze sdílet: Částečným řešením je vytvoření rozhraní k prostředku, které převezme úkoly procesů, které by jinak musely o prostředek žádat. Toto řešení můžeme použít například u tiskárny, kdy vytvoříme speciální obslužný proces, který bude obsluhovat tiskovou frontu tiskárny – přijme od procesu data, která mají být vytisknuta, zařadí do fronty a pak postupně tiskárně posílá data v dávkách se stanovenou strukturou a délkou.

Obecně však tento problém nelze řešit, u některých typů prostředků neexistuje možnost takové rozhraní vytvořit.

Proces má přidělen prostředek a čeká na jiný: Tento problém lze řešit dvěma způsoby, každý z nich je použitelný pro jiný typ prostředků:

1. Před vlastním spuštěním procesu mu budou přiděleny všechny prostředky, které by mohl potřebovat (použije se pro prostředky, které lze sdílet, například paměť).

2. Umožníme procesu žádat o další prostředky až když uvolní všechny prostředky, které měl přidělené (musí uvolnit všechny prostředky, které byly přiděleny postupem z tohoto bodu).

Pro každou třídu prostředků se bude používat jeden z těchto dvou způsobů řešení.

Hlavní nevýhodou této metody je, že prostředky přidělené podle prvního bodu mohou být zbytečně málo využívány (například čas procesoru u procesu, který je interaktivní, tedy často čeká na reakci uživatele), je také velká pravděpodobnost stárnutí některých procesů (proces čeká na prostředek, který je neustále přidělován jiným procesům, které například mají vyšší prioritu).

Nepreemptivní plánování: Řešením je využití některých prvků preemptivního plánování (tedy použijeme možnost odebrat prostředek procesu, třebaže tento proces by ještě potřeboval prostředek používat). Symbolicky můžeme postup zapsat takto (označíme R, S prostředky a P, Q procesy, proces P žádá o prostředek R):

```
IF (volný(R))
  {přiděl(P,R)}
ELSE IF (EXIST Q:(používá(Q,R) AND EXIST S:(čeká(Q,S))
  {uvolni(R), přiděl(P,R)}
```

Slovně: pokud prostředek, o který proces žádá, je volný, přidělíme mu ho, ale pokud ne, zjistíme, který proces tento prostředek má přidělen a přitom čeká na přidelění jiného prostředku. Pokud takový proces (`Q`) najdeme, odebereme mu prostředek (předpokládejme, že o tento prostředek může znovu požádat, až ho bude potřebovat), a přidělíme ho žádajícímu procesu `P`. Když nenajdeme žádný proces `Q`, kterému bychom mohli „zabavit“ žádaný prostředek, proces `P` bude muset počkat.

Tato metoda je opět vhodná pouze pro některé třídy prostředků, a to pro takové, které lze odebrat bez nebezpečí poškození dosavadní činnosti procesu – buď přerušení jejich používání nemá vliv na činnost procesu a navázání činnosti po opětovném přidělení není problém, nebo lze stav používání prostředku snadno zaznamenat a po znovupřidělení pomocí tohoto záznamu navázat (například čas procesoru nebo paměť při použití stránkování).

Kruhové čekání: Vytvoříme posloupnost tříd prostředků s přesně stanoveným pořadím, každý proces může žádat pouze o takové prostředky, které jsou v posloupnosti až za těmi, které již má přidělené.

Pokud chce proces požádat o prostředek třídy, která je v posloupnosti před některým prostředkem, který již má přidělen, musí uvolnit všechny prostředky, které by ten žádaný mohly v posloupnosti následovat.

Žádosti o prostředky z téže třídy musí být sdruženy, tedy jestliže proces „špatně odhadl“ množství prostředků z jedné třídy, které bude potřebovat k řádnému dokončení své činnosti, před další žádostí o dostatečné množství prostředků této třídy musí to, co již měl přiděleno, uvolnit.

Například máme posloupnost $R = (R_1, R_2, \ldots, R_n)$ tříd prostředků. Proces má přiděleny prostředky tříd R_1, R_3 a R_8. Bez problémů může požádat o prostředky z tříd $R_9, R_{10}, \ldots$, ale pokud bude chtít požádat o prostředek z třídy R_2, musí uvolnit přidělené prostředky z tříd R_3 a R_8. Jestliže chce požádat o další prostředky třídy R_3, musí uvolnit nejen prostředky třídy R_8, ale i třídy R_3.

Efektivnost této metody je do značné míry dána pořadím tříd prostředků v posloupnosti. Pokud je pořadí špatně navrženo, procesy téměř neustále čekají a může docházet k jejich stárnutí. Pořadí je vhodné navrhovat především s ohledem na obvyklé pořadí využívání zdrojů, například vnější paměti (obecně vstupní peri-

ferie) by měly být v posloupnosti před obvyklými výstupními periferiemi včetně tiskárny.

6.5 Předpovídání uváznutí

Při použití tohoto postupu nejsou procesy nuceny (obvykle) předčasně uvolňovat přidělené prostředky, ale principem je správně odhadnout, kdy by přidělení dalších prostředků mohlo způsobit uváznutí a takové přidělení pozdržet.

Stav systému je *bezpečný*, jestliže existuje alespoň jedno pořadí přidělování prostředků, při kterém všechny procesy mohou úspěšně dokončit svou činnost bez uváznutí. Stav, který není bezpečný, ještě nemusí znamenat uváznutí, ale může k němu vést.

První možné řešení je použitelné pro systém, kde každá třída prostředků má právě jednu instanci (využíváme graf přidělení prostředků), druhé je o něco náročnější, ale je použitelné i pro systém, kde třídy mohou mít více než jednu instanci. První řešení je postaveno na použití grafu přidělení prostředků, druhé, Bankéřův algoritmus, na simulaci různých pořadí přidělování zdrojů.

6.5.1 Graf nároků a přidělení prostředků

Vytvoříme *graf nároků a přidělení prostředků* úpravou grafu přidělení prostředků. Přidáme nový typ hrany, budeme mít tedy celkem tři typy hran:

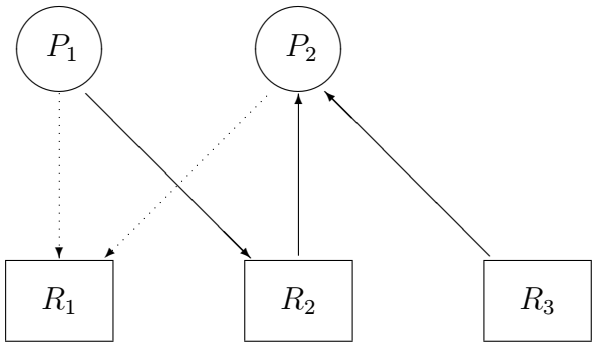
- hrana žádosti o prostředek vede od procesu, který žádá o prostředek, k vrcholu třídy prostředku, o který žádá,
- hrana přidělení prostředku vede od prostředku k procesu, kterému byl prostředek přidělen,
- hrana nároku vede od procesu k prostředku, znamená, že proces může požádat o tento prostředek.

Abychom hrany nároku odlišili od hran žádosti, budeme je značit tečkovaně.

Hrany nároku pro určitý proces vznikají při spuštění tohoto procesu, pokud proces požádá o prostředek, ke kterému od něho vede hrana nároku (nemůže požádat o prostředek, ke kterému hrana nároku nevede), tato hrana se změní na hranu žádosti o prostředek, v případě přidělení prostředku se mění na hranu přidělení prostředku (mění se orientace hrany) a po uvolnění se opět mění na hranu nároku (znovu změna orientace).

Hrana žádosti o prostředek se může změnit na hranu přidělení prostředku (a tedy prostředek je přidělen) pouze tehdy, když se touto změnou nevytvoří kružnice – změna totiž znamená změnu orientace hrany. Algoritmus tedy pouze „simuluje“ změnu orientace hrany a spustí postup detekce kružnice v grafu.

Na obrázku 6.2 je znázorněn stav systému se dvěma procesy a třemi různými prostředky. První proces nemá přiděleny žádné prostředky, ale žádá o prostředek R_2, má nárok požádat o prostředek R_1. Druhý proces má přiděleny prostředky R_2 a R_3, má nárok požádat o prostředek R_1. V grafu není žádná kružnice.



Obrázek 6.2: Graf nároků a přidělení prostředků

Když proces P_1 požádá o prostředek R_1 a ten je tomuto procesu přidělen, vznikne v grafu kružnice $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$, což znamená nebezpečný stav. Kdyby v tomto nebezpečném stavu požádal proces P_2 o prostředek R_1, došlo by k uváznutí, proto je nutné nedopustit ani vznik kružnice.

6.5.2 Bankéřův algoritmus

Při použití tohoto algoritmu musí systém evidovat informace, které mu budou pomáhat při rozhodování, zda prostředek přidělit. Každý proces musí předem oznámit, kolik kterých prostředků maximálně bude pro svou činnost potřebovat. Kdykoliv pak takový proces žádá o prostředky, systém zjistí, kolik by ještě ostatní procesy mohly potřebovat, a pokud dospěje k názoru, že přidělení žádaných prostředků nepovede do nebezpečného stavu, přidělí je.

Předpokládejme, že v systému pracuje n procesů a je k dispozici m různých tříd prostředků. Potřebujeme následující datové struktury:

`VOLNE` – vektor o délce m, je v něm pro každý prostředek uložen počet nepřidělených instancí,

PRIDELENO – matice s n řádky a m sloupci určující, kolik prostředků má který proces přiděleno, budeme chápat jako vektor vektorů o délce m, kde každý vektor přísluší jednomu procesu (tedy prostředky přidělené procesu P_i jsou ve vektoru PRIDELENO$[i]$),

POTREBUJE – matice s n řádky a m sloupci určující, kolik prostředků bude který proces ještě potřebovat k dokončení své činnosti, tedy po sečtení dvou matic PRIDELENO + POTREBUJE dostaneme matici obsahující údaj o tom, kolik kterých prostředků určitý proces maximálně potřebuje pro svou činnost od spuštění až po ukončení procesu (pro proces P_i je to vektor POTREBUJE$[i]$),

POZADAVKY – matice s n řádky a m sloupci požadavků jednotlivých procesů o prostředky, pokud jsou požadavky procesu vyplněny, data se přičtou k příslušnému řádku matice PRIDELENO.

Po spuštění procesu je příslušný řádek matice POTREBUJE naplněn údaji o tom, kolik kterého prostředku maximálně bude moci proces požadovat. Při každém přidělení prostředku je přidělený počet instancí přesunut z matice POTREBUJE do matice PRIDELENO, tedy proces postupně spotřebovává přidělené prostředky.

Dále budeme pro zjednodušení zápisu používat relaci $\leq$ pro vektory definovanou takto: nechť $V1$ a $V2$ jsou vektory o délce m. $V1 \leq V2$ právě tehdy když $\forall i(V1[i] \leq V2[i]),\ 1 \leq i \leq m$. Slovy: první vektor je menší nebo roven druhému, jestliže všechny jeho prvky jsou menší nebo rovny prvkům se stejným indexem druhého vektoru.

Když proces P_i žádá o přidělení prostředku, provede se tento algoritmus:

1. Požadavek procesu je přidán do i-tého řádku matice POZADAVKY (je přidán do vektoru POZADAVKY$[i]$).

2. Jestliže POZADAVKY$[i]$ $\leq$POTREBUJE$[i]$, pokračuj bodem 3, jinak odmítni přidělit prostředek (proces svým požadavkem překročil maximum prostředků, které ohlásil při svém spuštění).

3. Jestliže POZADAVKY$[i]$ $\leq$VOLNE, pokračuj bodem 4, jinak dej procesu P_i na vědomí, že bude čekat (proces žádá o víc, než kolik je momentálně k dispozici, proces musí počkat, až některý další proces uvolní prostředky).

4. Simuluj přidělení prostředků:

   VOLNE$=$VOLNE$-$POZADAVKY$[i]$
   PRIDELENO$[i]$ $=$PRIDELENO$[i]+$POZADAVKY$[i]$
   POTREBUJE$[i]$ $=$POTREBUJE$[i]-$POZADAVKY$[i]$

5. Vytvoř pomocné datové struktury, které budou sloužit k simulaci dalšího průběhu stavu systému v případě, že prostředky budou přiděleny:

   `SIMVOLNE` – vektor, ve kterém jsou při simulaci stejná data, jako v případě skutečného průběhu ve vektoru `VOLNE`, tento vektor inicializujeme hodnotami vektoru `VOLNE`, tedy `SIMVOLNE`=`VOLNE`,

   `KONEC` – vektor o n prvcích, které jsou inicializovány na $false$, pokud v průběhu simulace proces P_j bezpečně ukončí svou činnost, j-tý prvek tohoto vektoru se nastaví na $true$.

6. Najdi index j, pro který platí obě následující podmínky:

   (a) `KONEC`$[j] = false$ ..... ještě pracuje (neskončil)
   (b) `POTREBUJE`$[j] \leq$`SIMVOLNE` .. nebude potřebovat víc než je k dispozici

   Jestli takový index neexistuje, pokračuj bodem 8, jinak pokračuj bodem 7.

7. Proveď následující operace:

   `SIMVOLNE`=`SIMVOLNE`+`PRIDELENO`$[j]$
   ..... „uvolníme“ prostředky přidělené procesu P_j
   `KONEC`$[j] = true$ ..... „ukončíme“ proces P_j

   Pokračuj bodem 6.

8. Jestliže vektor `KONEC` obsahuje pouze hodnoty $true$, potom při simulaci nedošlo k zablokování a všechny procesy dokázaly bez problémů ukončit svou činnost, systém je v bezpečném stavu, v opačném případě (alespoň jedna hodnota $false$) by se systém po přidělení požadovaných prostředků procesu P_i dostal do nebezpečného stavu.

Jestliže po simulaci stav bezpečný, systém přidělí požadované prostředky procesu P_i (vlastně to už udělal v bodu 4), jinak proces musí čekat, než bude zase dostatek prostředků a je nutné vrátit změny z bodu 4 (vektor `POZADAVKY`$[i]$ bude nadále obsahovat nevyplněné požadavky procesu).

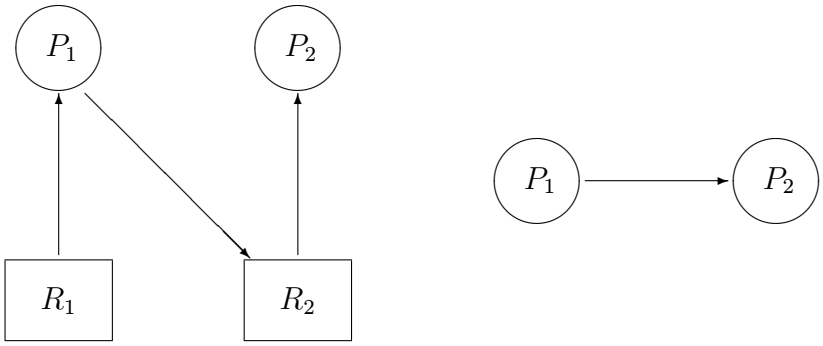
6.6 Detekce uváznutí

Opět rozlišíme dva případy: první metoda je určena pro systém, kde v každé třídě prostředků je právě jeden prostředek, druhá metoda pro systém, kde je ve třídách prostředků povoleno i více instancí.

6.6.1 Úprava grafu přidělení prostředků

Vytvoříme *graf čekání*, ve kterém bude zachyceno vzájemné čekání mezi procesy (jeden proces čeká, až jiný uvolní nějaký prostředek). Protože nás momentálně zajímá jen to, který proces uvázl, nepotřebujeme informaci o tom, na které prostředky které procesy čekají (snadněji se detekuje kružnice).

Graf čekání získáme z grafu přidělení zdrojů tak, že odstraníme všechny uzly odpovídající prostředkům a necháme hrany, které do nich a z nich vedly, zkolabovat (tedy hrana, která vedla od procesu k prostředku, se přesměruje na všechny uzly – procesy, ke kterým vedly hrany přidělení prostředku).



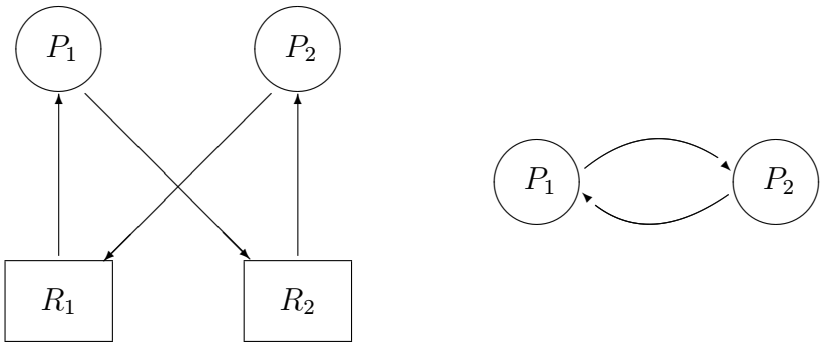
Obrázek 6.3: Graf přidělení prostředků a ekvivalentní graf čekání

Na obrázku 6.3 je ukázka vytvoření grafu čekání pro graf přidělení prostředků. Je to obdoba grafu nároků prostředků na obrázku 6.2 ze strany 87 bez prostředku R_3, který nemá vliv na uváznutí, s přidělením požadovaného prostředku R_1 procesu P_1. V grafu čekání není kružnice, proto bude tento prostředek přidělen. Kdyby byl používán postup předpovídání uváznutí, k přidělení by nedošlo, tady však neprovádíme predikci, ale pouze detekci již existujícího uváznutí.

Na obrázku 6.4 je v grafu přidělení prostředků stav, kdy proces P_2 žádá o přidělení prostředku R_1, a ekvivalentní graf čekání. V obou grafech je kružnice, v tom druhém je snáze zjistitelná (máme méně uzlů v grafu), detekovali jsme uváznutí systému.

6.6.2 Úprava Bankéřova algoritmu

Bankéřův algoritmus slouží k předvídání uváznutí, pro detekci stačí jeho zjednodušení. Budeme používat následující datové struktury definované stejně jako u Bankéřova algoritmu:



Obrázek 6.4: Graf přidělení prostředků a ekvivalentní graf čekání

- `VOLNE`
- `PRIDELENE`
- `POZADAVKY`

Algoritmus pro zjištění uváznutí je následující:

1. Vytvoř pomocné datové struktury, které budou sloužit k simulaci dalšího průběhu stavu systému v případě, že prostředky budou přiděleny:

   `SIMVOLNE` – inicializujeme hodnotami vektoru `VOLNE`, `SIMVOLNE`=`VOLNE`,
   `KONEC` – vektor o n prvcích, které jsou inicializovány na $false$.

2. Najdi index j, pro který platí obě následující podmínky:

   (a) `KONEC`$[j] = false$ ..... ještě pracuje (neskončil)
   (b) `POZADAVKY`$[j] \leq$`SIMVOLNE` ..... nežádá víc než je k dispozici

   Jestli takový index neexistuje, pokračuj bodem 4, jinak pokračuj bodem 3.

3. Proveď následující operace:

   `SIMVOLNE`=`SIMVOLNE`+`PRIDELENO`$[j]$
   ..... „uvolníme“ prostředky přidělené procesu P_j
   `KONEC`$[j] = true$ ..... „ukončíme“ proces P_j
   Pokračuj bodem 2.

4. Jestliže vektor `KONEC` obsahuje pouze hodnoty $true$, potom nedošlo k uváznutí, v opačném případě (alespoň jedna hodnota $false$) došlo k uváznutí, a to těch procesů, pro jejichž index je hodnota ve vektoru `KONEC` rovna $false$.

6.7 Reakce při zjištění zablokování

Některým z algoritmů z předchozí kapitoly bylo zjištěno uváznutí a víme také, které procesy uvázly (v případě prvního algoritmu jsou to procesy na detekované kružnici, u druhého algoritmu procesy, jejichž index ve vektoru `KONEC` je nastaven na $false$). Další reakce závisí na tom, zda s prostředky pracujeme preemptivně nebo nepreemptivně.

Při preemptivní práci s prostředky postupně uvolňujeme prostředky, které mají přiděleny uváznuté procesy, a přidělujeme je jiným procesům tak dlouho, dokud se neodstraní uváznutí. Klíčový je „výběr oběti“, tedy procesů, kterým budou prostředky postupně odebírány, mělo by být také zajištěno, aby po odstranění uváznutí tyto procesy mohly postupně prostředky opět dostávat a ukončit tak svou práci.

Pokud nepoužíváme nepreemptivní plánování, jediným řešením je ukončovat procesy tak dlouho, dokud existuje uváznutí, při takovém násilném ukončení procesu jsou jeho prostředky uvolněny a přiděleny jiným procesům. Opět je důležité, jak vybíráme „oběť“, protože násilně ukončený proces samozřejmě nemůže dokončit svou práci. Existují procesy, které takto můžeme ukončit a pak restartovat bez nebezpečí ztráty dat nebo nekonzistence dat či stavu systému, u jiných bohužel toto nebezpečí je.

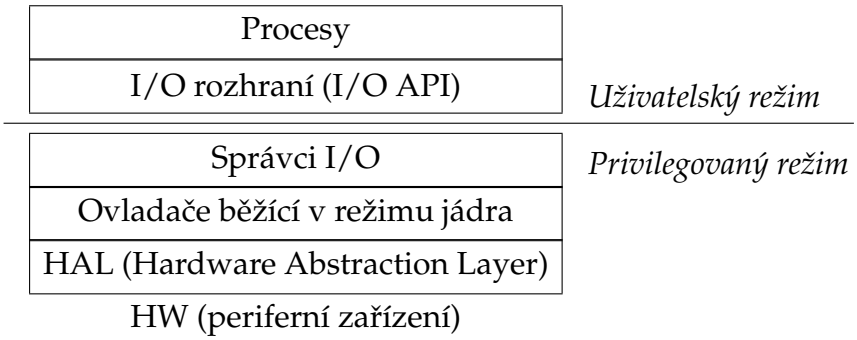
KAPITOLA 7

Správa periferií

Periferie se také nazývají vstupně-výstupní zařízení (V/V zařízení, I/O zařízení). V této kapitole se nejdřív podíváme strukturu I/O systému, druhy periferií, ovladače a potom se budeme krátce věnovat problematice nízkoúrovňového přístupu k periferiím pomocí přerušení.

7.1 I/O systém

Obvyklá struktura I/O systému je ve většině moderních operačních systémů následující:



Obrázek 7.1: Struktura I/O systému

I/O rozhraní je sada rutin (funkcí) a objektů poskytovaná operačním systémem procesům pro přístup k periferiím, jiným způsobem obvykle běžné procesy s periferiemi komunikovat nemohou (to neplatí pro některé systémové procesy).

Správci periferií jsou moduly systému, které provádějí správu určitého zařízení, například správce tisku nebo správce pro některý disk, bývají uspořádáni do stromové struktury, ve které nadřízený správce řídí činnost ostatních správců.

7.2 Druhy periferií

Zařízení dělíme na *vstupní* a *výstupní*, ale toto dělení není úplně disjunktní – existují zařízení, která patří do obou těchto skupin (pak je nazýváme vstupně-výstupní). Vstupní je například klávesnice, výstupní běžný monitor nebo tiskárna, vstupně-výstupní jsou třeba diskové paměti nebo dotyková obrazovka.

Z hlediska možností využívání procesy dělíme periferie do tří skupin:

- *vyhrazená zařízení* – tato zařízení nemohou sloužit více procesům najednou, je to například tiskárna. Správce tohoto zařízení musí zajistit, aby procesy nebyly zbytečně zdržováni. Pro to existují dvě základní možnosti:
  - vyhrazování zařízení – jednodušší možnost, která ale moc problémů neřeší – jeden proces používá zařízení, ostatní musí počkat třeba ve frontě, až proces sám zařízení uvolní,
  - virtualizace zařízení (ovladač typu server, viz dále) – proces ve skutečnosti komunikuje s jakousi „virtuální náhradou“, speciálním procesem, a teprve tento proces komunikuje se samotným zařízením, komunikace s procesem je rychlejší než se zařízením a navíc díky různým technologiím včetně multithreadingu může speciální proces komunikovat s více procesy najednou.
- *sdílená zařízení* – tato zařízení mohou sloužit najednou více procesům s tím, že každý proces má vyhrazenu svou vlastní část, typický příklad je operační paměť nebo vnější paměťová média. Správce přiděluje, odebírá a eviduje části zařízení přidělené jednotlivým procesům a musí především zajistit, aby procesy přistupovaly pouze k těm částem, které jsou pro ně vyhrazeny nebo kam je jim přístup povolen.
- *společná zařízení* – k těmto zařízením může bez problémů přistupovat více procesů najednou, jejich stav nebývá z vnějšku měněn a proto nevyžadují často ani synchronizaci přístupu. Je to například mikrofon nebo některý typ čidla (třeba teploměr či vlhkoměr).

Periferní zařízení dále dělíme podle rozsáhlosti dat, se kterými dokážou najednou pracovat jejich ovladače na

- *znaková* – klávesnice, tiskárna, myš, terminál, apod.,
- *bloková* – paměťová zařízení jako je třeba pevný disk,
- *speciální* – například časovač.

7.3 Ovladače

Ovladač zařízení je program (proces po spuštění), který slouží jako rozhraní mezi zařízením a procesy.

Hlavní úlohou ovladače je zprostředkovávat komunikaci mezi zařízením a procesy tak, aby procesy mohly stejným způsobem přistupovat k různým zařízením téhož typu, například u dvou různých tiskáren by neměl být proces nucen zjišťovat, jak přesně mají být formátována data, jaké parametry mají být tiskárně zadány a v jakém pořadí, atd. Proto správce zařízení poskytuje procesům sadu služeb (funkcí), které jsou vždy pro jakékoliv zařízení stejně nazvány, jen pokaždé jinak pracují. Jsou to například funkce:

`Init(zařízení)` inicializuje zařízení, funkce obvykle vrací jeho stav (připraveno nebo chyba),

`Open(zařízení)` otevře komunikační kanál mezi zařízením a procesem, naváže spojení, funkce vrací identifikaci komunikačního kanálu (číslo, které bude nadále pro komunikaci používáno),

`Close(zařízení)` uzavře komunikační kanál, zruší spojení,

`Read(zařízení,Blok)`, `Write(zařízení,Blok)` přenos dat mezi blokovým zařízením a procesem – čtení bloku dat ze zařízení, zápis bloku dat,

`Getc(zařízení)`, `Putc(zařízení,Zn)` – totéž pro znaková zařízení – čtení znaku ze zařízení, poslání znaku na zařízení,

`Seek(zařízení,param)` přesun na zadanou pozici v rámci poslaných dat,

`Cntl(parametry)` přístup k dalším možnostem zařízení, má různé parametry podle toho, co zařízení nabízí (všechny funkce, které se „nevejdou“ do předchozích).

Rozlišujeme dva základní druhy ovladačů:

- *klasické ovladače* – jednodušší procesy, které zprostředkovávají komunikaci, překládají volání obecně pojmenovaných služeb na konkrétní proudy dat určené zařízení,

- *ovladače typu server* – speciální procesy vytvářející dokonalejší rozhraní mezi procesy (klienty) a zařízením; proces pošle data tomuto speciálnímu procesu a nemusí čekat na jejich další zpracování ani posílat po malých dávkách podle požadavků zařízení, místo čekání může dále pokračovat ve své práci, speciální proces zařadí tato data do fronty a postupně je sám posílá zařízení (architektura klient-server).

Z mnoha důvodů je dobré rozdělit ovladač na dvě části, které mezi sebou komunikují stylem Producent - konzument:

- *horní část* je Producent, přebírá data od procesů a ukládá je do fronty (u vstupních zařízení zase přebírá data z druhé části, kompletuje je a zasílá adresátovi),
- *dolní část* je Konzument, komunikuje přímo se zařízením – vybírá z fronty data a podle požadavků zařízení mu je posílá (u vstupních zařízení přebírá data ze zařízení a řadí do fronty).

Toto rozdělení má smysl zvláště u ovladačů typu server, ale je užitečné i u klasických ovladačů. Dolní polovina je hardwarově závislá, proto když chceme napsat ovladač pro několik druhů téhož typu zařízení (např. několik různých tiskáren), stačí přepsat dolní část a do horní téměř nemusíme zasahovat.

7.4 Přerušení

Pod pojmem přerušení chápeme přerušení normálního běhu procesu (posloupnosti vykonávaných instrukcí jeho programu). V multitaskovém systému přerušení způsobí změnu stavu běžícího procesu (je odebrán procesor), ale až po dokončení právě zpracovávané instrukce[1].

Přerušení může být generováno buď hardwarově, pak hovoříme o *hardwarovém přerušení*, tato přerušení mají přidělena čísla *IRQ* (Interrupt Request – požadavek přerušení), nebo softwarově operačním systémem nebo běžícím procesem[2], pak jde o *softwarové přerušení*.

Hardwarová přerušení jsou například generována I/O zařízeními jako je klávesnice (stisk klávesy) nebo myš (pohyb či stisknutí tlačítka), ale třeba také procesorem při pokusu o dělení nulou (takováto přerušení vzniklá chybou se také nazývají

[1] Instrukce je nejmenší a dále nedělitelný povel, kterému rozumí již přímo procesor, program je posloupnost těchto instrukcí. Jednomu příkazu vyššího programovacího jazyka odpovídá obvykle celý sled instrukcí. Typicky jde o přesuny jedno- či několikabytových údajů mezi registrem procesoru a pamětí, jednoduché aritmetické operace apod.

[2] V „zabezpečenějších“ operačních systémech běžné procesy nemohou generovat přerušení přímo, ale voláním jádra.

vyjímky – exceptions), při neznámé instrukci, přerušení generovaná časovačem (v pravidelných intervalech, předem nastavených), při hardwarové implementaci ochrany paměti je procesorem generováno přerušení při neoprávněném přístupu do chráněné paměti.

Softwarová přerušení jsou generována procesem například při žádosti o prostředek (včetně výstupu na obrazovku) nebo při pokusu vyvolat určitou událost a tím i její obslužnou rutinu (uvnitř procesu nebo i u jiného procesu či operačního systému), operačním systémem například při porušení bezpečnosti systému.

Po vykonání každé instrukce procesor zjistí, zda během jejího vykonávání nedošlo k vygenerování přerušení, a jestliže ano, postupuje se následovně:

1. běžící proces je přerušen a zařazen do některé fronty (obvykle fronta připravených procesů), jeho kontext je uložen,
2. řízení převezme operační systém, resp. jeho modul pro obsluhu přerušení (Interrupt Handler), který zjistí, o jaké přerušení jde a vytvoří datovou strukturu s údaji týkajícími se přerušení (typ, jak bylo vyvoláno, související data, ...), pokud takovéto struktury jsou vyžadovány,
3. podle typu přerušení se určí, jak má být ošetřeno, tedy je spuštěna příslušná obslužná rutina (funkce nebo procedura, která je na toto přerušení napojena),
4. po provedení obslužné rutiny je procesor přidělen některému z připravených procesů (může to být tentýž, který byl přerušen).

Za určitých okolností nesmí být ošetřena (tj. musí být ignorována) přerušení určená danému procesu, např. z důvodů synchronizace, pak hovoříme o *zákazu přerušení*. Zakázaná přerušení jsou tzv. *maskována* (maskované přerušení je ignorováno), některá přerušení však nelze maskovat a proces o nich musí obdržet zprávu (například dělení nulou).

Správa periferií musí především zajistit správnou obsluhu přerušení. V nejjednodušším případě se provádí pomocí *vektorů přerušení* udávajících adresu obslužné rutiny. V systému MS-DOS je správa přerušení prováděna následovně:

- pro každé přerušení je nadefinován programový kód (obslužná rutina – program, funkce, rutina, tj. ovladač přerušení), který se má spustit v případě, že je toto přerušení generováno,
- vektor přerušení je uspořádaná dvojice (vektor o dvou prvcích) [*segment*,*offset*], která udává adresu v paměti (tj. je to vlastně pointer), na které je právě tento programový kód, a když víme, kde hledat tento vektor, pak snadno můžeme spustit obsluhu přerušení, které nastalo,

- vektory přerušení jsou uloženy od adresy, která je známá nejen systému, ale také všem programům, každý vektor obsahuje dva údaje, každý zabírá 2 B, tedy celkem vektor zabírá 4 B paměti,
- vektory jsou naskládány za sebou, proto když známe číslo přerušení, které nastalo (přerušení jsou očíslována od 0), stačí provést výpočet

$$\textit{výchozí adresa} + \textit{číslo přerušení} * 4,$$

  získáme adresu, na které je vektor přerušení s adresou kódu obsluhujícího přerušení s tímto číslem,
- pokud je generováno přerušení, je přerušen běh programu a je zpracována obsluha přerušení určená vektorem, potom může běh programu zase pokračovat (MS-DOS není multitaskový systém, plnohodnotně může běžet jen jediný proces).

Pokud je implementován multitasking, je samozřejmě nutné obsluhu přerušení rozšířit, protože generované přerušení nemusí být určeno běžící aplikaci a navíc může být nadefinováno velké množství softwarových přerušení. Potom se výše popsaným způsobem spravují pouze základní druhy přerušení a vektory přerušení ukazují na části modulu obsluhy přerušení, který shromáždí informace o typu přerušení, adresátovi a další data a to vše pošle (například jako zprávu) adresátovi. Adresát (proces, kterému je přerušení určeno) má pak definovány vlastní obslužné rutiny, které pak zprávu dále zpracují.

KAPITOLA 8

Paměťová média

Paměťová média také patří mezi periferní zařízení, ale protože jejich správa je nejnáročnější, nejfrekventovanější a klíčová pro práci celého operačního systému (operační systém je koneckonců uložen na pevném disku, což je paměťové médium), budeme se jim věnovat podrobněji zde a pak ještě v následující kapitole o správě disků.

8.1 Základní pojmy

Pod pojmem *paměťové médium* (nebo také vnější paměť) rozumíme periferní zařízení sloužící k ukládání dat. Hardwarovým rozhraním, přes které je paměťové médium připojeno k počítači, je *úložiště*. V případě disků a pásek (pevných disků, CD, disket, zálohovacích pásek, apod.) je to mechanika tohoto disku či pásky, pro USB flash disky je to USB rozhraní, své úložiště mají také paměťové karty.

Paměťová média mohou být buď napevno připojena datovým kabelem k základní desce počítače, třeba přímo ve skříni počítače (například běžný pevný disk) nebo mohou být vyměnitelná, a v tom případě je datovým kabelem připojeno pouze úložiště tohoto média (například CD mechanika).

Média mohou být buď se sekvenčním přístupem (pásky) nebo mohou umožňovat přístup na kteroukoliv svou část (především disky). V případě sekvenčního přístupu není problém s adresací, ale v případě disků používáme „vícerozměrnou“ adresaci určující přesnou polohu dat na disku.

V dalším textu budeme používat pojmy týkající se struktury disku:

Stopy jsou soustředné kružnice na disku.

Sektory jsou výseče kružnic, každý sektor obsahuje 512 B dat (tj. 1/2 kB). Protože by v takovém případě sektory u vnějšího okraje byly mnohonásobně větší než sektory blízko středu disku (vždy se ale do sektoru uloží právě 512 B dat), u novějších disků je na vnějších stopách více sektorů než na stopách vnitřních. Disk samotný dokáže pracovat vždy jen s celými sektory, neumí je rozdělit na části.

Desky a povrchy – pevný disk se skládá z více desek (kruhů) na jedné ose, každá deska má dva povrchy.

Hlava (čtecí a zápisová) je jedna pro každou dvojici protilehlých povrchů (tj. v každé „štěrbině“ jedna).

Cylindr (z angl. cylinder, válec) je tatáž stopa na všech površích (tedy ze všech povrchů vezmeme stopu s daným poloměrem, a to je také poloměr cylindru).

Fyzická adresa dat na disku závisí na typu disku (IDE nebo SCSI), je to buď [*povrch, stopa, sektor*] nebo [*cylindr, hlava, sektor*], s touto adresou nedokáže přímo pracovat operační systém, ten pracuje s *logickou adresou* (to může být třeba číslo logického sektoru, kdy sektory máme očíslovány absolutně na celém disku, sektory s nižším číslem jsou u okraje, s vyšším číslem u středu disku).

Cluster (čte se [*klastr*]) je jeden nebo více sektorů, a stejně jako samotný disk dokáže pracovat jen s celými sektory, souborový systém dokáže pracovat pouze s celými clustery (například soubor, i když zabírá třeba jen 3 B, musíme uložit do celého clusteru, zbytek clusteru zůstane nevyužit). Je to tedy nejmenší část disku, se kterou dokáže pracovat operační systém.

Aby mohlo být paměťové médium používáno, musí na něm být předpřipravena určitá struktura, musí být *formátováno*.

Nízkoúrovňové formátování je zápis značek pro sektory a stopy na magnetickém médiu (pevný disk, disketa, apod.), provádí obvykle výrobce.

Vysokoúrovňové formátování je vytvoření souborového systému, určíme, jakým způsobem budou na médiu data ukládána a vytvoříme některé nezbytné datové struktury (např. pro FAT souborový systém FAT tabulku). Pro tento úkon se pojem „formátování “ používá prakticky jen ve Windows, v jiných operačních systémech se používá pojem „vytvoření souborového systému“.

Před vytvořením souborového systému můžeme paměťové médium, typicky pevný disk, *rozdělit* na *logické disky* (svazky, oblasti, partitions podle terminologie v různých operačních systémech), na jednom fyzickém disku je vždy alespoň jeden logický disk.

Rozdělení se provádí pomocí nástrojů často nazývaných FDISK. Tento nástroj je součástí většiny dnes používaných systémů. Bohužel jsou do určité míry navzájem nekompatibilní a může se stát, že disk rozdělený fdiskem jednoho operačního systému (třeba Linuxu) dělá problémy jinému operačnímu systému (Windows, proto se doporučuje v případě, že uživatel chce mít na jednom disku Windows i Linux, pro základní rozdělení a nadefinování Windows oblastí použít fdisk Windows a teprve pro zbytek disku fdisk Linuxu).

Některé nástroje na rozdělení disku nedokážou měnit hranice oblastí bez ztráty dat (tj. data musíme zálohovat), ale některé Linuxové nástroje a některé programy pro Windows (zde většinou komerční) dokážeou s hranicemi oblastí pracovat bez ztráty dat (ale zálohovaná by pro jistotu být měla). Na logickém disku (oblasti) již můžeme vytvořit souborový systém.

8.2 Adresářová struktura

Paměťová média mohou obsahovat velmi mnoho dat, a aby bylo vůbec možné se v těchto datech vyznat, vyhledávat, používat je, přidávat další nebo některá odstraňovat, musí být vhodně organizována.

Data se obvykle nacházejí v jednotkách, které nazýváme *soubory*. Soubor je tedy posloupnost dat s vlastním významem, dat, která k sobě nějakým způsobem patří (třeba dokument, obrázek nebo tabulka).

Souborů může být opět velmi mnoho, proto také musí být tříděny. Třídění se provádí do jednotek, kterým říkáme *adresáře*. Adresář obvykle obsahuje údaje o souborech, které jsou do něho vloženy, včetně jejich fyzického umístění na disku (adresy), od toho i název. Protože adresář je vlastně souhrn dat o souborech, v mnoha operačních systémech je transparentně chápán také jako soubor, třebaže se speciálním významem.

Adresáře tvoří strukturu, která nabývá různých stupňů složitosti. Adresář, který obsahuje vše ostatní, co se na médiu nachází, se nazývá *kořenový adresář* (root). Podle toho, do jaké míry může být adresářová struktura složitá a její prvky navzájem vnořené. Rozlišujeme tyto druhy adresářových struktur:

Jednoúrovňová struktura – existuje pouze jediný adresář, root, všechny soubory jsou v něm. Tuto koncepci používal operační systém CP/M.

Dvouúrovňová struktura – v rootu mohou být odkazy na adresáře, tyto adresáře však nemohou obsahovat další adresáře, jen soubory. Je to vylepšení jednoúrovňové struktury o rozdělení souborů jednotlivých uživatelů a systému.

Stromová struktura – v adresáři mohou být další adresáře, které se nazývají *podadresáře*, v kterémkoliv adresáři mohou být soubory. Celá struktura tvoří strom s jedním kořenem – kořenovým adresářem. Tuto strukturu používá pro své souborové systémy Windows.

Acyklická struktura – oproti stromové struktuře navíc přidává možnost mít soubory a některé adresáře uloženy ve více adresářích, tedy k některým položkám může vést více než jedna cesta. Je nutné zajistit acykličnost, aby při vyhledávání nedocházelo k zacyklení vyhledávacího algoritmu.

Položka (soubor nebo podadresář) je fyzicky pouze jednou na adrese, která může být uvedena ve více adresářích. Výhodou je především snadný přístup k témuž souboru z různých adresářů (například z adresářů patřících různým uživatelům). Používá se v Unixových souborových systémech.

Cyklická struktura – k položkám může existovat více než jedna cesta, narozdíl od předchozího řešení jsou dovoleny i cykly, používá se pouze jako nástavba pro jednodušší struktury.

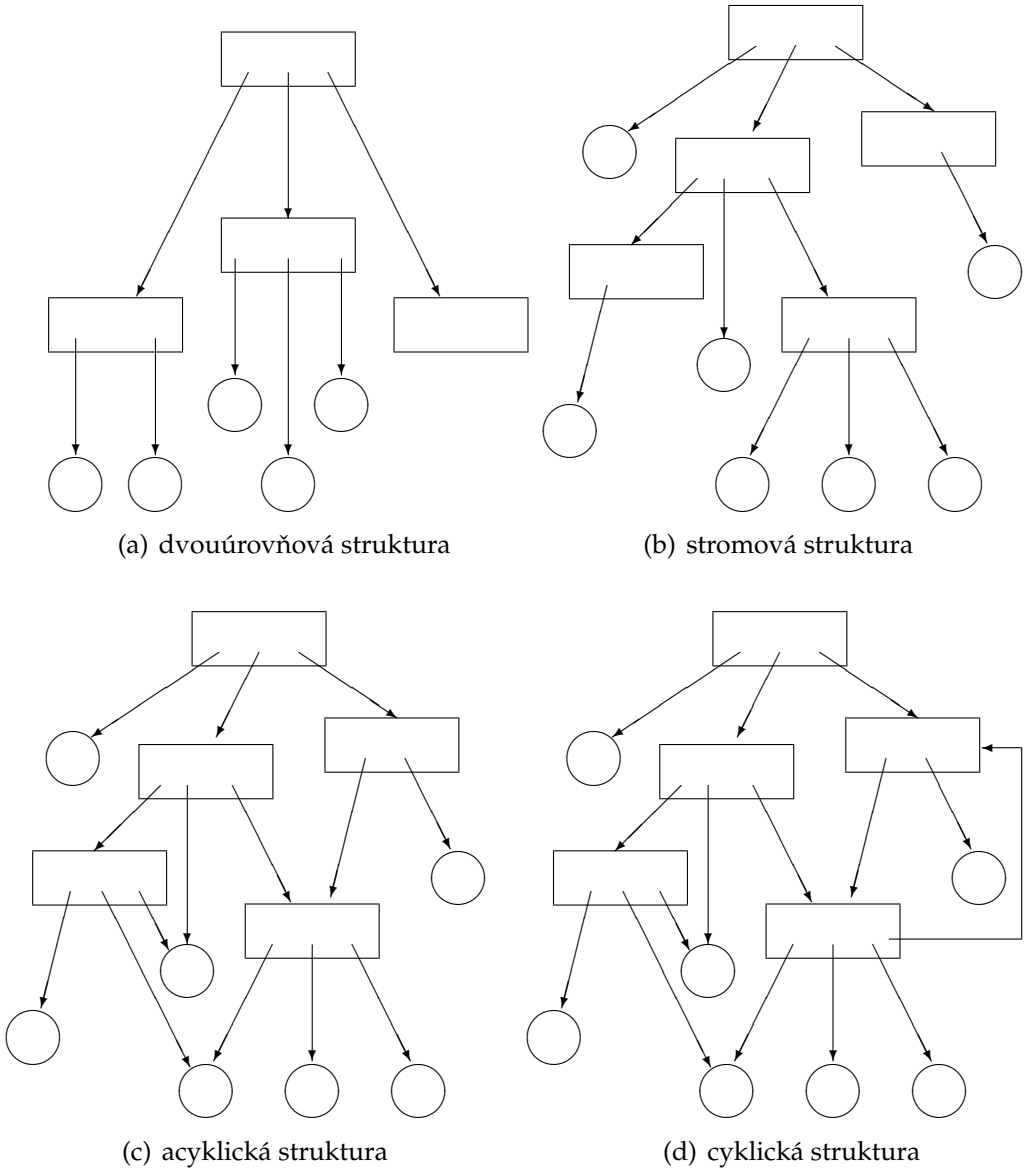
Na obrázku 8.1 na straně 103 je ukázka všech těchto adresářových struktur kromě té nejjednodušší, jednoúrovňové.

U acyklické struktury může být problémem zachování acykličnosti grafu při přidávání nových adres do adresářů. To lze řešit více způsoby. Nejjednodušším způsobem je omezení týkající se vícenásobných adres – ve více adresářích může být jen soubor, nikoliv adresář, nebo je možné „přibrat“ některé adresáře se zvláštním významem (například pro snazší pohyb ve struktuře může v adresáři být odkaz na sebe sama a na nadřízený adresář, tradičně nazvané `.` a `..`).

V této struktuře dále musí být vyřešeno rušení položek tak, aby nevznikali „sirotci“ bez jakéhokoliv umístění, a tedy nevyhledatelní, třebaže zabírající místo na disku. To lze řešit dvěma způsoby:

a) Každá položka (soubor i adresář, který může být ve více adresářích) má čítač, který zachycuje počet odkazů na tuto položku (počet výskytů adresy této položky v různých adresářích). Při rušení položky je nejdřív její čítač snížen o 1. Pokud po tomto snížení má hodnotu 0, je položka fyzicky vymazána, jinak je ponechána.

b) Kromě výskytů adres v adresářích jsou položky evidovány systémem ještě zvlášť. Při mazání položky se odstraní pouze její záznam v tom adresáři, ze kterého mažeme, tedy odstraní se pouze jeden odkaz na položku. Systém pak pravidelně prochází všechny položky a fyzicky odstraňuje ty, které nejsou v žádném adresáři, na které nevede žádný odkaz.



(a) dvouúrovňová struktura (b) stromová struktura

(c) acyklická struktura (d) cyklická struktura

Obrázek 8.1: Různé typy struktur adresářů

Cyklická struktura adresářů se, jak již bylo uvedeno, používá pouze jako nástavba jiné struktury. Obvykle jde o systém „lehkých“ (soft) odkazů (ve Windows se nazývají zástupci). Tyto odkazy jsou krátké soubory s informací o skutečné adrese položky a případně dalšími informacemi.

8.3 Soubory a systém souborů

Pod pojmem *soubor* rozumíme datový objekt uchovávaný na vnějším paměťovém médiu, logicky je to posloupnost dat s vlastním významem (dat, která k sobě patří).

Rozeznáváme tyto základní typy souborů:

- standardní (dokumenty, spustitelné soubory),
- adresáře,
- simulované (pro přístup k I/O zařízení nebo pro mechanismus pipes), nemusí být přítomny fyzicky na disku,
- odkládací soubory pro virtuální paměť.

Souborový systém (systém souborů) jsou metody a struktury dat, pomocí kterých operační systém udržuje záznamy o souborech. Data se na disk ukládají lineárně (jeden bit za druhým), souborové systémy potřebujeme jako jednoduché databáze, které umožňují přístup ke konkrétním datům, třídění (do adresářů) a udržování informací o těchto datech.

V každém operačním systému jsou u souborů evidovány trochu jiné vlastnosti. Kromě názvu souboru a jeho přípony je třeba určovat přístupová práva k tomuto souboru nebo atributy. To je realizováno různými způsoby, z nichž některé budou podrobněji popsány dále. Některé možnosti:

a) Souborové systémy FAT (Windows s DOS jádrem) – určují se pouze atributy, žádná ochrana přístupu, jsou to atributy A (k archivaci), D (adresář), L (popisek disku), S (systémový), H (skrytý), R (pouze pro čtení).

b) Souborový systém NTFS (Windows s NT jádrem) – přístupová práva `n` (žádné), `r` (právo čtení), `w` (zápisu), `c` (změny), `f` (veškerá práva), práva se přiřazují uživatelům nebo skupinám (a tedy všem členům dané skupiny). Dají se dědit, tedy není nutné definovat je pro každou položku zvlášť.

c) Multics – každý soubor obsahuje jako metadata seznam uživatelů s jejich přístupovými právy.

d) Unixové souborové systémy – práva `r` (číst), `w` (zapisovat), `x` (spouštět). Každé položce se přiřazují tato práva pro vlastníka (obvykle uživatele, který soubor vytvořil), konkrétní skupinu (obvykle skupina, do které patří vlastník) a pro ostatní, tedy ve vlastnostech souboru jsou tři údaje, každý z nich obsahuje kombinaci práv `rwx` (tři bity, pokud je právo přiděleno, je bit nastaven na 1). Evidován je také vlastník souboru a skupina.

Systémy souborů můžeme členit podle různých kritérií, uvedeme si členění podle odolnosti vůči haváriím:

1. *Souborové systémy s okamžitým zápisem* (FAT, FAT32) – pokud aplikace chce zapisovat na disk a zároveň probíhá jiná disková operace, musí počkat. Výhodou je bezpečnost (data se nemohou neočekávaně ztratit bez toho, aby to aplikace nevěděla), nevýhodou snížení výkonnosti (čekání).
2. *Souborové systémy s opatrným zápisem* (HPFS) – rozdělí zápis do posloupnosti takových operací, že když dojde s selhání při zápisu, data zůstanou konzistentní. Vlastně se jedná o jednoduchý databázový systém s definovanými transakcemi.
3. *Souborové systémy s opožděným zápisem* – používají cache paměť (vyrovnávací paměť), tedy data se nejdřív zapisují do cache paměti, zapisující aplikace může dále pracovat, z cache paměti se data zapíšou na disk až tehdy, když disk dokončí předchozí probíhající operaci (když má čas). Výhodou je zvýšení výkonnosti systému, nevýhodou je možnost ztráty dat při havárii.
4. *Žurnálovací souborové systémy* (journalized, zotavitelné – NTFS a většina Linuxových souborových systémů) si uchovávají informace o operacích, které byly provedeny, aby bylo možné v případě výpadku dostat data zpět do konzistentního stavu.

   Změny jsou evidovány podobně jako v databázích jako transakce. Transakce se skládá z jednoduchých (atomických) operací, navzájem oddělitelných, tyto operace se postupně evidují. Po provedení všech operací, ze kterých se transakce skládá, je odesláno potvrzení, které znamená úspěšné ukončení transakce, jednotlivé operace transakce se z žurnálu vymažou (už nejsou potřeba). Pokud systém „spadne“, třeba dojde k náhlému výpadku el. proudu, můžeme se u nedokončených transakcí vrátit zpět podle zaznamenaných operací. V případě NTFS žurnálování probíhá takto:

   - během každé operace na disku jsou dílčí operace zaznamenávány do log souboru, po ukončení operace jsou všechny tyto dílčí operace z log souboru vymazány,
   - po startu systému se prochází tento log soubor a opakují se všechny dokončené transakce (aby bylo jisté, že byly zapsány z cache paměti na disk) a ruší všechny nedokončené,
   - mohou se používat kontrolní body (místo, kdy jsou vždy všechny transakce provedeny, v pravidelných časových intervalech, od tohoto bodu lze provést zotavení).

Virtuální souborový systém je takový souborový systém, který nemá přímou podporu na konkrétním paměťovém médiu. Virtuální souborové systémy se používají k abstrakci přístupu k ostatním souborovým systémům (především v Unixových systémech) nebo pro snadnější přístup k datům, která přímo nesouvisejí s jedním fyzickým zařízením (například běhové údaje o stavu systému v Unixových systémech). Jde vlastně o jakési virtuální rozhraní.

Důležitým problémem je *fragmentace*. Fragmentace je způsobena především tím, že pokud je soubor příliš dlouhý, mohou být jeho části (fragmenty) uloženy na různých částech disku. Fragmentace se musí často řešit v souborových systémech, které ve snaze rychle najít volné místo při ukládání souboru vezmou první volný blok, začnou ukládat, když nestačí, najdou další volný blok, který samozřejmě může být úplně jinak umístěn, pokračují v ukládání, pak další volný blok, ...

Souborové systémy pro vyměnitelná média: Na vyměnitelných médiích (CD, DVD, USB flash disk, disketa, apod.) se používají obvykle takové souborové systémy, kterým „rozumí " pokud možno všechny operační systémy nebo alespoň ten operační systém, který máme nainstalován. Pro CD je to obvykle *CDFS* (Compact Disk File System), pro DVD, ale i pro CD, je to UDF (Universal Disk Format) nebo některý FAT systém, USB flash disky mívají některý souborový systém typu FAT, diskety FAT12 nebo ext2fs.

8.4 Souborové systémy ve Windows

8.4.1 Starší verze souborových systémů typu FAT

Souborové systémy typu FAT byly vyvinuty pro operační systémy MS-DOS a Windows. FAT je zkratka z File Allocation Table, systém je založen na evidenci umístění souborů a adresářů v tabulce na začátku disku.

FAT12 je souborový systém použitelný dnes prakticky jen na disketách (je pro disky menší než 16 MB). Platí zde to, co je řečeno v následující podkapitole s tím rozdílem, že dokáže adresovat nejvýše 212 clusterů (to je 4096 kB = 4 MB, na disketě můžeme proto zvolit 1 cluster = 1 sektor, na disketě jsou necelé 3000 sektorů). FAT tabulka obsahuje položky o délce 12 bitů (ale všechny tyto bity nejsou využity pro identifikaci obsahu clusterů).

FAT16 je určen již pro pevné disky. Délka clusteru je pro velmi malé disky obvykle 2 sektory (1 kB), se zvyšující se kapacitou disku je tato hodnota výrazně vyšší, určuje se napevno podle velikosti disku. Struktura logického disku se souborovým systémem FAT16 je následující:

- *boot sektor* (zaváděcí sektor, odkaz na zaváděcí záznam = umístění programu, který po zapnutí nebo restartu počítače zavede operační systém)
- *FAT* (File Allocation Table), tabulka obsazení logického disku
- její kopie (použitelná v případě, že se první FAT poškodí)
- *root* (hlavní adresář disku) – zvláštní struktura s pevnou délkou, proto v hlavním adresáři disku může být pouze limitovaný počet objektů (souborů nebo adresářů)
- *clustery* – zde jsou ukládány soubory a další adresáře. Adresáře jsou uspořádány do stromové struktury. Clustery jsou očíslovány (od 1), každý má podle svého pořadového čísla přiřazen jeden záznam ve FAT tabulce.

Obsah FAT tabulky. Jednotlivé clustery datové oblasti jsou očíslovány, FAT obsahuje pro každý cluster jeden záznam zabírající 2 B (od toho název FAT 16, 2 B = 16 bitů, ale ve skutečnosti se pro čísla clusterů nepoužívá všech 16 bitů, některé jsou vyhrazeny a vytvářejí speciální kódy například pro vadný cluster).

Obsah záznamů v tabulce určuje, co v příslušném clusteru najdeme. Jestliže je cluster volný, je zde číslo 0x0000, vadný – číslo 0xFFF7 (toto číslo zde zapisuje scandisk nebo podobné programy pro kontrolu povrchu disku).

Pokud je v clusteru uložena část některého souboru nebo adresáře, v tabulce je na tomto místě identifikace následujícího clusteru (tedy například pro soubor cluster, ve kterém pokračuje, jde o zřetězení). Jestliže jde o poslední cluster souboru nebo adresáře (a proto žádný cluster „nenásleduje“), je v záznamu FAT číslo 0xFFFF.

Příklad 8.1

Soubor začíná na clusteru s číslem 0x0021, pokračuje postupně na clusterech 0x0027, 0x0025, 0x0026, 0x0029. Cluster 0x0022 je poškozený, ostatní až po cluster 0x002A jsou volné. FAT tabulka od záznamu 21 po záznam 2A vypadá takto:

Záznam	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A
Obsah	0027	FFF7	0000	0000	0026	0029	0025	0000	FFFF	0000

Tabulka 8.1: Příklad struktury FAT tabulky v souborovém systému FAT16

Pokud chceme načíst určitý soubor (nebo adresář), musíme předně znát číslo clusteru, na kterém začíná. V záznamu ve FAT tabulce pro tento cluster zjistíme, na kterém clusteru pokračuje, v jeho záznamu najdeme číslo dalšího článku v řetězci, atd.

Řetězení clusterů může být výhodou (organizace nezabírá příliš mnoho místa na disku), ale také nevýhodou (poškození jednoho údaje ve FAT vede k tomu, že ztratíme celý zbytek souboru).

Datová oblast. Pod tímto pojmem budeme rozumět vše, co je za FAT tabulkami, tedy root a clustery. Root obsahuje odkazy na adresáře, adresáře mohou podle stromové struktury obsahovat odkazy na další adresáře nebo odkazy na soubory, root také může obsahovat soubory. Root také může obsahovat položku typu *label* (popisek), který představuje jméno disku (diskety).

Zatímco běžný soubor obsahuje jakákoliv data, adresář se skládá z položek o délce 32 B popisujících soubory a podadresáře pro daný adresář, v položkách jsou evidovány následující informace:

- název souboru či podadresáře (8 B),
- přípona souboru (3 B),
- pokud položka představuje label, tedy název disku, tento název zabírá celých předchozích 11 (8+3) B,
- atributy (1 B), jednotlivé bity znamenají `xxADLSHR`, kde

  `x` volné bity, nepoužívají se,
  `A` k archivaci,
  `D` directory – adresář,
  `L` label – název disku, atributům předchází samotný název,
  `S` systémový,
  `H` skrytý,
  `R` pouze pro čtení.

- čas a datum vytvoření a datum posledního přístupu (3+2+2 B),
- čas a datum poslední změny, tj. zápisu do souboru nebo změny struktury adresáře (2+2 B),
- první cluster souboru nebo adresáře (pro label nemá význam, = 0) (2 B),
- délka souboru nebo adresáře (pro label nemá význam) (4 B), zbytek rezervován.

Pokud například je v nějakém adresáři 5 podadresářů a 2 soubory, najdeme v clusteru, který je pro tento adresář přidělen, celkem 7 položek, z nich 5 má atribut nastaven na `00010000` (pokud není pro celý adresář zapnuta archivace), zbylé dvě položky jsou odkazy na soubor a atribut mohou mít například ve tvaru `00100000`.

Ve všech 7 položkách je důležitým údajem také to, co je ve výčtu uvedeno v předposlední odrážce, cluster, na kterém začínají data souboru nebo adresáře. Ve FAT tabulce pak nalezneme záznam s tímto číslem a zjistíme, jestli se jedná o poslední cluster (obsahuje číslo `0xFFFF`) nebo kterým clusterem posloupnost pokračuje.

Velikost clusteru je pro disky velikosti od 512 MB do 1 GB stanovena na 16 kB, pro větší (do 2 GB) na 32 kB.

Udává se, že FAT16 není použitelná pro logické disky větší než 4 GB, a není prakticky použitelná pro disky větší než 2 GB (pro max. velikost clusteru 32 kB, tedy 16 sektorů, která je použita v DOSu a starších Windows s DOS jádrem) nebo 4GB (ve Windows NT - umožňují využít maximální velikost clusteru 64 kB, tedy 32 sektorů).

8.4.2 VFAT a FAT32

VFAT je zkratka z Virtual FAT a je to nástavba pro FAT16, která k vlastnostem tohoto souborového systému přidává především podporu dlouhých názvů souborů (týká se také delších přípon souborů, jako je třeba HTML) a možnost používat v názvech některé další znaky (jako třeba znaky národních abeced nebo mezery). Jde o virtuální ovladač, přes který jde komunikace se systémem FAT16, najdeme ho od Windows 95. Tedy pokud ve Windows od této verze, v řadě NT od verze 3.5, používáme FAT16, jde o VFAT. Tímto termínem bývá také označován systém FAT32, který má podobné vlastnosti, ale již interně.

Název souboru nebo adresáře ve VFAT maximálně 255 znaků, některé zdroje uvádějí, že tato délka je včetně cesty k souboru. Omezení je nutné, protože název souboru (víceméně často včetně cesty k souboru) je používán jako parametr mnoha funkcí při programování, musí se vejít do paměťového prostoru vymezeného daným datovým typem.

Samotná podpora dlouhých názvů je realizována tak, že pro delší název je využita následující položka (položky) v adresáři. Položky adresáře mají trochu jinou formu, rozeznáváme oproti třem typům v původní FAT čtyři typy:

- položky pro soubory,
- položky pro adresáře,

- položky (-a) pro label (jmenovku) disku,
- položka pro rozšířený název souboru nebo adresáře.

Položka pro rozšířený název souboru nebo adresáře má specifickou formu. Délka je stejná jako u ostatních (32 B), ale obsahuje některé další parametry a 13 symbolů pro rozšířený název, stejně jako u souborů jsou tyto položky zřetězeny (ve FAT tabulce), následující položka v řetězci obsahuje dalších 13 znaků, ...

V původní položce souboru nebo adresáře je název souboru pro DOS ve formě 8.3 odvozený z dlouhého jména konverzí (vypuštění mezer a dalších v DOSu „nedovolených znaků“, případně jejich nahrazení, konec je odříznut a nahrazen identifikací rozlišující soubory nebo adresáře se stejným zkráceným názvem.

Microsoft uvádí, že dlouhé jméno lze použít i pro label disku, ovšem reálně mohou nastat problémy s kompatibilitou disku pro různé operační systémy.

FAT32 je použitelný v operačních systémech Windows 95 OSR2, 98, ME, 2000, XP a novějších. Verze Windows 95, Windows NT do 4.x včetně a starší s ním nedokážou pracovat.

FAT32 přejímá všechny vlastnosti VFAT včetně podpory dlouhé názvy souborů. Lze nadefinovat různou velikost clusterů v rozmezi MIN - 16 (nebo 32) sektorů, podle verze Windows, kde hodnota MIN se řídí velikostí disku:

Velikost disku	Nejmenší velikost clusteru
512 MB – 8 GB	4 kB
8 GB – 16 GB	8 kB
16 GB – 32 GB	16 kB
32 GB – 2 TB	32 kB = 16 sektorů

Tabulka 8.2: Nejmenší velikost clusteru pro souborový systém FAT32

Při vytváření souborového systému tedy můžeme volit kteroukoliv hodnotu v tomto rozmezi. Doporučuje se nevolit příliš nízkou hodnotu, protože to zvyšuje nároky na správu souborového systému (velká FAT tabulka, pomaleji se v ní hledá), vyšší než potřebná hodnota zase není výhodná, pokud máme hodně malých souborů (každý soubor zabírá nejméně jeden cluster). Proto se doporučuje zvolit nějaký vhodný kompromis.

FAT32 má oproti FAT16 tyto výhody:

- je možné stanovit při formátování i menší velikost clusteru, takže pokud máme mnoho „malých“ souborů, je disk optimálněji využit,

- je použitelná pro disky větší než 2 GB (ale pro logické disky menší než 512 MB ji nelze použít),
- velikost FAT tabulky může být jakákoliv, interně se s ní zachází jako se souborem, proto je možné ji prodlužovat,
- root se skládá z běžných clusterů, může být proto jakkoliv dlouhý a taktéž přesouván na jiné místo,
- systém reaguje rychleji a je lépe chráněn proti chybám,
- podporuje dlouhé názvy souborů.

Ve FAT tabulce se tedy nacházejí záznamy o clusterech a jde o 32bitová čísla. Pokud jde o záznamy určující, na kterém clusteru pokračuje soubor či adresář, ve skutečnosti jsou uloženy v 28 bitech tohoto čísla, zbytek je opět vyhrazen speciálním kódům. Například:

- `0x00000000`, `0x10000000`, `0xF0000000` znamenají, že cluster je volný (spodních 28 bitů je 0, zbylé mohou obsahovat cokoliv),
- `0x0FFFFFF7` je chybný cluster,
- `0xFFFFFFFF` znamená poslední cluster souboru nebo adresáře.

8.4.3 Souborový systém NTFS

NTFS (New Technology File System) je žurnálovací souborový systém vyvinutý pro Windows řady NT, byl používán již v prvních verzích (3.x). Hlavním požadavkem při jeho vyvíjení bylo zajištění větší bezpečnosti dat, především možnost definování přístupových práv pro různé uživatele. Je určen pro velké disky, lze ho použít i na malé disky, ale ne na diskety.

V souborovém systému NTFS máme možnost řídit přístup k souborům a složkám definováním přístupových práv pro různé uživatele a skupiny. Každému souboru nebo složce je přiřazen *Access Control List* (ACL, seznam řízení přístupu) se seznamem uživatelů a skupin a jejich přístupovými právy.

Druhy přístupových práv jsou `n` (není dovolen žádný přístup), `r` (právo čtení), `w` (také právo zápisu), `c` (právo změny), `f` (úplné řízení), vždy pro určitého uživatele nebo skupinu.

Přístupová práva se definují v grafickém rozhraní ve Vlastnostech souboru (složky), karta Zabezpečení, nebo v Příkazovém řádku příkazem `cacls` (existují ještě další možnosti).

Aby nebylo nutné definovat plný ACL pro každý soubor nebo adresář, přístupová práva se mohou dědit. Pro určení, jak má dědění fungovat, se používá u složek parametr `/t`, který způsobí změnu i u podsložek zpracovávané složky. U souborů, které nejsou složkami, se samozřejmě dědění nepoužívá.

Když příkazem `cacls složka` vypíšeme ACL této složky, dědění je zachyceno těmito zkratkami:

OI platí pro tuto složku a všechny soubory v ní (ne pro podsložky),
CI platí pro tuto složku a všechny podsložky v ní (ne pro soubory v ní),
IO neplatí pro tuto složku.

Zkratky jsou ve výpisu zkombinovány takto:

(OI)(CI) platí pro tuto složku a celý její obsah (podsložky i soubory),
(OI)(CI)(IO) platí pro celý její obsah – podsložky i soubory (ale ne pro samotnou složku),
(CI)(IO) platí jen pro podsložky v ní obsažené,
(OI)(IO) platí jen pro soubory v ní obsažené.

Vlastnosti:

- Všechno je soubor (tedy také všechny implicitní struktury na disku jsou implementovány jako speciální soubory).
- Možnost řídit přístup k souborům a složkám definováním přístupových práv pro různé uživatele a skupiny.
- Podpora *násobných proudů dat* – každý soubor může obsahovat více datových proudů (nejméně jeden). Jeden z nich je hlavní, není pojmenován, jde vlastně přímo o data souboru, ostatní proudy jsou pojmenované (například stream s názvem STREAM5 u souboru SOUBOR.XYZ je SOUBOR.XYZ:STREAM5). V proudech může být cokoliv, ve Windows 2000 se v sekundárních proudech například ukládá autor a informace o obsahu souboru, celkově ale záleží na programátorovi aplikace vytvářející soubor.

  Datový proud můžeme vytvořit například pro dokumenty ve vlastnostech souboru, záložka Souhrn (nebo Vlastní).
- Názvy souborů jsou v UNICODE (sice zaberou více místa na disku, ale je možné používat i znaky nepatřící do anglické národní znakové sady).
- Možnost *indexace* podle různých typů dat (nejen název souboru, ale také přístupová práva, čas vytvoření souboru, ...), zrychluje vyhledávání dat na disku (NTFS implementuje vpodstatě databázové funkce). Indexace může mít ale také negativní efekt, protože celkově zpomaluje výkonnost systému

(udržování indexů vyžaduje, aby při každé změně určitých údajů byl změněn i indexový soubor). Pokud nastane tento problém, je možné indexování vypnout (vypneme službu Indexing Services).

- *Dynamické přemapování vadných sektorů* (vadný sektor se nahradí jiným, pokud jsou data redundantní, pak se při poškození zkopírují „ze zálohy“).
- *Šifrování a komprese*. Šifrování je podporováno až od Windows 2000, používá EFS (Encrypting File System) založený na symetrických klíčích, je prováděno „za běhu“, při práci uživatele.
- *Pevné odkazy* – tyto odkazy zůstávají funkční i po přesunu objektu, na který ukazují (souboru, adresáře), ale narozdíl od pevných odkazů na Unixových souborových systémech nejsou rovnocenné s původním objektem. Mohou být definovány pouze v rámci jednoho svazku, a to například příkazem `fsutil hardlink create`.
- *Řídké soubory* – soubory, které obsahují rozsáhlejší oblasti s nulovou informační hodnotou (oblasti vyplněné 0), mohou být uloženy tak, že tyto „prázdné“ oblasti na disku nezabírají žádné místo.

V terminologii NTFS hovoříme o logických discích jako o *svazcích*. Velikost (implicitní) clusterů je stejně jako v FAT systémech odvozena od velikosti svazku podle tabulky (tab. 8.3 na str. 113), ale můžeme při vytváření souborového systému stanovit prakticky jakoukoliv (udává se do 64 kB, tj. 32 sektorů).

Velikost svazku	**Velikost clusteru**
512 MB nebo méně	512 B
512 MB - 1 GB	1 kB
1 GB - 2 GB	2 kB
2 GB nebo více	4 kB

Tabulka 8.3: Velikost clusteru pro souborový systém NTFS

Na disku jsou mimo samotná data také implicitní struktury, které zde označujeme jako metadata (jde o soubory). Jsou to například:

`$MFT` (Master File Table) – obdoba FAT tabulky ve FAT systémech. Záznam v této tabulce má obvykle 1 kB, ale může být jakkoliv prodloužen. Najdeme zde záznamy pro všechny soubory na disku (MFT je také soubor, proto jsou zde informace i o ní), v každém záznamu je především odkaz za umístění začátku souboru, bezpečnostní nastavení, atributy, ...

`$LOGFILE` – log soubor (žurnál), do kterého se ukládají transakční informace.

`$BITMAP` – je to pole bitů, pro každý cluster na disku je zde vyhrazen jeden bit. Pokud je bit nastaven na 0, je cluster volný, 1 znamená, že je obsazený.

`$BADCLUS` – obdobným způsobem jsou zachyceny vadné clustery. Atd.

V běžných souborových manažerech, ve kterých pracujeme se soubory, jsou tyto speciální soubory neviditelné, i když existuje způsob, jak je zviditelnit (přes Příkazový řádek). Neviditelné jsou také všechny datové proudy souboru kromě hlavního, zobrazovaná délka souboru se také týká hlavního proudu, takže po smazání jednoho malého souboru by se teoreticky mohlo stát, že na disku je najednou o několik kB více volného místa.

NTFS se brání fragmentaci tak, že pro uložení souboru hledá vždy ne nejbližší, ale nejbližší vhodnou posloupnost navazujících clusterů (ve které je tolik místa, že se tam soubor vejde, obdoba metody BestFit pro operační paměť, viz kap. 3.3, str. 32), takže fragmentace vzniká pouze tehdy, když je na disku příliš málo volného místa (není žádná „vhodně velká“ posloupnost clusterů) nebo když je soubor po změně prodloužen a za jeho clustery není volný cluster. Fragmentace by byla problémem především u MFT, protože ta se může libovolně prodlužovat s tím, jak roste počet a délka v ní obsažených záznamů. NTFS to řeší tak, že kolem MFT nechává některé clustery volné, nedovoluje nikomu je zabrat a vyhrazuje je pro MFT.

NTFS ve své implicitní podobě snižuje výkon systému. Na rychlejších počítačích to nevadí, ale jinak existují způsoby, jak jeho práci zrychlit. Užitečný a celkem logický je tento způsob: NTFS i při procházení adresářovou strukturou aktualizuje datum a čas posledního přístupu. To můžeme vypnout tak, že v registru najdeme hodnotu `NtfsDisableLastAccessUpdate` a změníme ji na 1.

8.4.4 Srovnání souborových systémů pro Windows

FAT16, FAT32 a NTFS mají své výhody i nevýhody, podle nich se rozhodujeme, který souborový systém zvolíme na určitý logický disk.

Obecně platí, že pokud volíme ten souborový systém, se kterým dokáže pracovat použitý operační systém, a pokud máme na počítači více operačních systémů, na prvním logickém disku (podle terminologie Windows disk C) volíme ten souborový systém, se kterým dokážou pracovat všechny operační systémy (obvykle to bývá FAT16 nebo FAT32, pokud máme Win98, Win2000 a Linux, použijeme na C FAT32 a tam taky nainstalujeme Win98, pokud máme Win95 starší nebo Win3x a cokoliv jiného, musíme mít na disku C FAT16).

Použitelnost jednotlivých souborových systémů v různých operačních systémech je v tabulce 8.4. Pro velikost logického disku a maximální možnou velikost souboru platí tabulka 8.5.

	Podporováno v OS
FAT16	DOS (samotný), všechny verze Windows včetně NT, Linux
FAT32	Windows 95 OSR2, 98, ME, 2000, XP, Linux
NTFS	Windows řady NT (včetně 2000 a XP), Linux někdy pouze pro čtení

Tabulka 8.4: Podpora souborových systémů v různých verzích Windows

	Max. velikost disku	**Počet clusterů**	**Max. objektů v rootu**	**Max. délka souboru**	**Max. počet souborů**
FAT16	2 (4 v NT) GB	max. 2^{16}	512	4 GB bez 1 B	2^{16}
FAT32	512 MB - 2 TB (XP: do 32 GB)	min. 2^{16}	65 534	2^{32} B bez 1 B	téměř 2^{32}
NTFS		$2^{64}-1$ (XP: $2^{32}-1$)		2^{64} B bez 1 kB (XP: 2^{44} B bez 64 kB)	$2^{32}-1$

Tabulka 8.5: Srovnání souborových systémů pro Windows

8.5 Souborové systémy pro Linux

8.5.1 VFS

Linux pracuje s virtuálním souborovým systémem *VFS* (Virtual File System), přes který jsou přístupné všechny „reálné“ souborové systémy na počítači. Jde o vrstvu jádra operačního systému, přes kterou jdou všechna volání diskových služeb, zastřešuje souborové systémy na všech svazcích a discích přítomných v systému (včetně disket a CD) a v případě potřeby předává řízení (lépe řečeno požadavky) vždy konkrétnímu souborovému systému, se kterým se pracuje. Přes VFS uživatel jednotně přistupuje také ke všem zařízením a vše je zahrnuto v jedné adresářové struktuře s jediným kořenem (root).

Pokud chceme disk (příp. svazek) používat, musíme ho připojit (mount) do VFS buď v grafickém rozhraní nebo v konzole příkazem `mount`. Systémový disk

je připojen už při startu systému, o ten se tedy nemusíme starat, ostatní svazky na pevných discích obvykle také bývají připojeny automaticky (záleží na distribuci). Připojit je třeba výměnná média (disketa, CD-ROM), v grafickém prostředí je to opět většinou řešeno automaticky třeba při poklepání na ikonu diskety. Disk, který nepoužíváme, musíme odpojit (viz cvičení).

V Linuxu se na pevných discích nejčastěji používají souborové systémy ext3fs a ReiserFS, můžeme používat také souborové systémy Windows a jiných operačních systémů, jsou mapovány pod těmito názvy:

`msdos` – kompatibilní s FAT12 nebo FAT16 bez VFAT,

`vfat` – obvykle se používá pro FAT32 nebo FAT16 s nástavbou VFAT,

`ntfs` – kompatibilní s NTFS Windows NT, implicitně je nastavena často pouze možnost čtení,

`iso9660` – CD-ROM,

`hpfs` – kompatibilní s HPFS v OS/2, a další.

Souborovým systémem je také linux swap, který je určen pro swap partition.

V Unixu a Linuxu platí, že „všechno je soubor“ (snad kromě uživatele :-), tedy i adresáře, se zařízeními se také pracuje jako se soubory. Při komunikaci mezi procesy používáme také speciální komunikační kanály nazývané roury (pipes), které se implementují pomocí souborů, ale pracují obvykle pouze v operační paměti.

8.5.2 Souborové systémy typu ext$x$fs

ext2fs: Partition se souborovým systémem je rozdělena na *bloky*, jejichž velikost je možné předem stanovit (obvykle 1024, 2048 nebo 4096 B). První tento blok, *bootblok*, na systémovém svazku obsahuje zaváděcí program, na jiných svazcích zůstává nepoužit.

Další bloky jsou rozděleny do *skupin bloků*. Každá skupina obsahuje speciální blok, tzv. *superblok*, s informacemi o souborovém systému jako celku (například velikost souborového systému, počet i-uzlů – viz dále, počet bloků, . . .), následuje blok s popisem této skupiny, bloky zaznamenávající obsazenost bloků a i-uzlů, tabulka i-uzlů a pak teprve bloky s daty.

To, že důležité informace o systému jsou přítomny v každé skupině, a tedy vlastně zálohovány, umožňuje nejen efektivnější práci v systému, ale také je to bezpečnější.

V tabulce 8.6 na str. 117 je zachyceno, jak může vypadat struktura partition s ext2, která je dlouhá 20 MB s délkou bloku 1024 B.

Začátek (č. bloku)	Počet bloků	Popis
0	1	boot blok
skupina bloků 0		
1	1	superblok
2	1	popis skupiny bloků
3	1	bitmapa použitých bloků ve skupině (pro každý blok 1 bit, pokud $= 0$, volný)
4	1	bitmapa použitých i-uzlů skupiny, bit určitého i-uzlu najdeme podle jeho indexu v tabulce i-uzlů
5	214	tabulka i-uzlů, obsahuje cesty k jednotlivým i-uzlům, tedy i-uzel je identifikován indexem v této tabulce
219	7974	bloky s daty
skupina bloků 1		
8193	1	superblok – záloha
8194	1	popis skupiny bloků
8195	1	bitmapa použitých bloků ve skupině
8196	1	bitmapa použitých i-uzlů skupiny
8197	214	tabulka i-uzlů
8408	7974	bloky s daty
skupina bloků 2		
16385	1	superblok – záloha
16386	1	popis skupiny bloků
16387	214	tabulka i-uzlů
16601	3879	bloky s daty

Tabulka 8.6: Struktura partition se souborovým systémem ext2fs

Jak je vidět, některé části skupiny bloků jsou nepovinné (například bitmapa použitých bloků). To, jestli je ve skupině přítomna určitá část, a také na kterém místě v paměti, se můžeme dovědět v popisu skupiny bloků (za superblokem), pozici celé skupiny a popisu skupiny najdeme v superbloku (samozřejmě kterémkoliv). Nejdůležitějším pojmem pro Unixové souborové systémy je *i-node* (i-uzel).

I-uzel je struktura obsahující důležité informace o souboru (ID vlastníka, délka souboru, čas posledního zápisu, čas posledního otevření, čas vytvoření, ...) a odkazy na 15 bloků. Z nich

- 12 bloků obsahuje data souboru (1. úroveň)
- 13. blok může obsahovat odkazy na další bloky, ve kterých jsou uložena data souboru (2. úroveň)

- 14. blok může obsahovat odkazy na bloky obsahující odkazy na bloky s daty (3. úroveň)
- 15. blok může obsahovat odkazy na bloky obsahující odkazy na bloky s odkazy na bloky s daty (4. úroveň).

Předpokládejme, že pro adresy se používá 32 bitů, tedy 4B, a délka bloku je 1024 B (1 kB). Soubor použije bloky jen po tu úroveň, která mu stačí.

- první úroveň stačí pro soubory s délkou do 12288 B (12*1024 B), tj. 12 kB, alokováno je 1 - 12 bloků podle potřeby,
- druhá úroveň stačí pro soubory s délkou do 12 kB + 256*1024 B = 268 kB,
- třetí úroveň stačí pro soubory s délkou do 268 kB + 256*256*1024 B = 65804 kB = 64 MB a 268 kB,
- čtvrtá úroveň stačí pro soubory s délkou do 65804 kB + 256*256*256*1024 B = 16 GB a 64 MB a 268 kB.

Tento příklad je pouze ilustrativní, ve skutečnosti je tato struktura ještě trochu složitější a samozřejmě může být zvolena jiná velikost bloků než 1024 B.

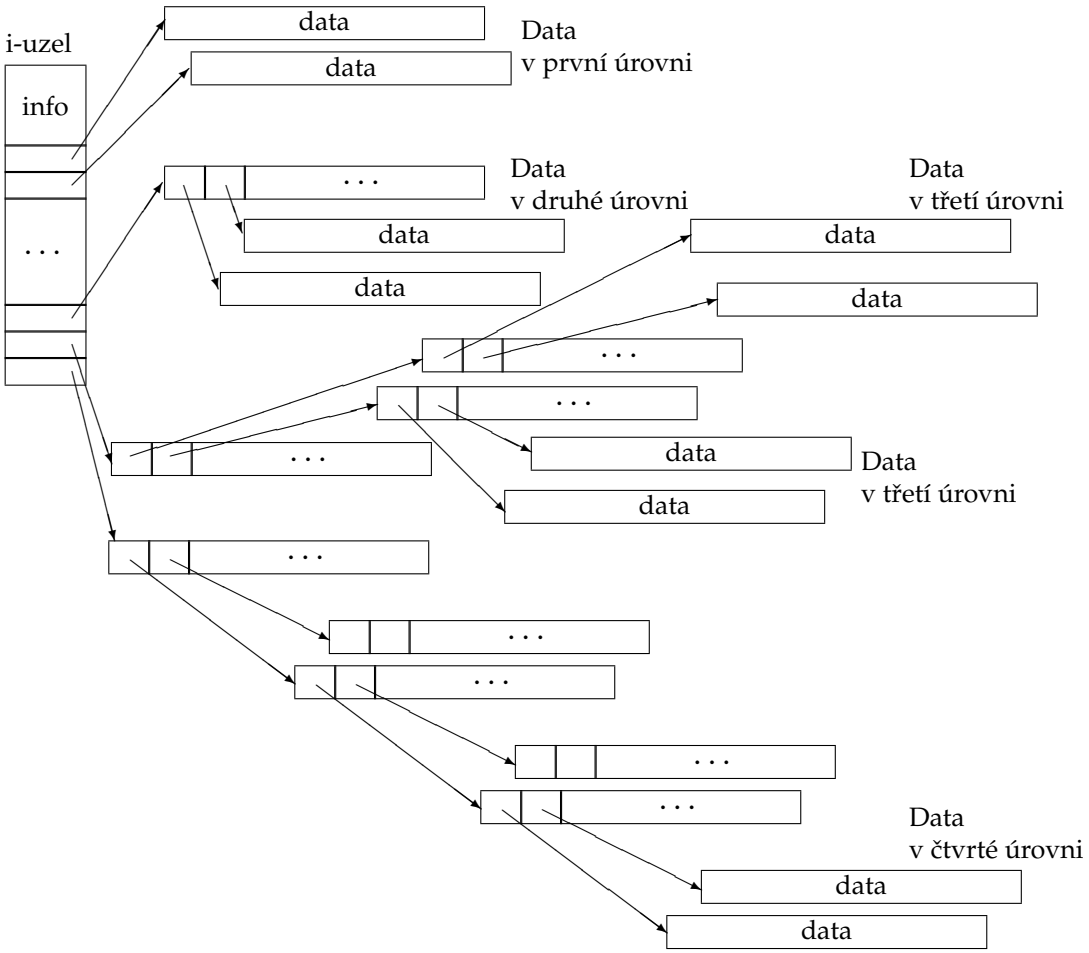
Obrázek 8.2 na straně 8.2 je zkrácenou ukázkou struktury souboru v souborovém systému ext2.

Každý *adresář* může obsahovat soubory nebo další adresáře. Adresáře jsou speciální soubory obsahující seznam záznamů proměnné délky. Každý záznam obsahuje číslo i-uzlu, délku záznamu, název souboru a délku souboru. Záznamy jsou proměnné délky, aby bylo možné používat dlouhé názvy souborů - pokud bychom měli pevně danou délku záznamu, bylo by hodně místa v paměti nevyužitého. Adresáře, stejně jako každá jiná struktura na partition, je také chápán jako soubor, proto má svůj i-uzel a může být rozprostřen ve více blocích stejně jako jiné soubory.

Můžeme používat také *odkazy* (links).

U *pevného odkazu* (hard link) několik názvů souboru může být asociováno s jediným i-uzlem a tedy všechny ukazují na tentýž fyzický soubor. U každého i-uzlu je informace o počtu odkazů, při mazání souboru je soubor fyzicky smazán až tehdy, když tento počet klesne na 0, tedy když jsou už smazány všechny odkazy. Všechny pevné odkazy na jeden soubor mají stejnou důležitost, žádný z nich není hlavní.

Pevné odkazy mají některá omezení, která mají především zajistit, aby v grafu adresářové struktury nevznikl cyklus: pevný odkaz nesmí ukazovat na adresář kromě sebe sama a nadřízeného adresáře (to jsou odkazy `.` a `..`), a také nesmí ukazovat na objekty, které jsou v jiném souborovém systému (třeba na jiné partition).



Obrázek 8.2: Struktura souboru v souborovém systému ext2fs

Symbolické odkazy (soft link) jsou obdobou Zástupců u Windows systémů, obsahují (v textové podobě) údaj o umístění souboru, na který odkazují. Výhodou je odbourání omezení vynucených u pevných odkazů, symbolický odkaz může ukazovat na jakýkoliv uzel v adresářové struktuře včetně uzlů jiných souborových systémů.

Volný prostor je evidován v řetězovém seznamu, jehož struktura je podobná i-uzlům. V jednom z bloků skupiny bloků je pole, jehož prvky odkazují na volné bloky; pokud je těchto bloků více než je kapacita pole, potom jeden prvek tohoto pole ukazuje na blok, který obsahuje odkazy na volné bloky, ... Obdobně jsou evidovány také všechny i-uzly daného bloku.

Pro ext2fs se udává, že je použitelný pro disky až do 4 TB. Podporuje dlouhé názvy souborů (až do 255 znaků, ale tento limit je možné posunout ještě dále, pokud je potřeba). Tento souborový systém se však dnes už prakticky nepoužívá, jeho nástupcem je ext3fs. Má smysl pouze tam, kde je rychlost důležitější než zachování konzistence dat při jejich změnách, protože je o něco rychlejší než ext3fs (tj. pro ty adresáře, jejichž obsah se prakticky nemění, ale často nebo na dlouhý časový okamžik se k nim přistupuje, např. `/boot`).

ext3fs je vylepšením ext2fs. Je zpětně kompatibilní (přesněji kompatibilní v obou směrech), zachovává všechny struktury ext2, ale navíc jde o žurnálovací souborový systém (Journal File System). Pokud máme na partition souborový systém ext2, stačí vytvořit žurnálovací soubor a při nové inicializaci systému můžeme partition připojit jako ext3, a naopak, pokud máme partition nadefinovanou jako ext3, můžeme ji při dalším startu systému připojit jako ext2.

8.5.3 Další žurnálovací souborové systémy

ReiserFS je dalším z používaných Linuxových souborových systémů. Implicitně ho volí instalace některých distribucí, například Debian nebo SUSE (RedHat a Mandrake zase prosazují spíše ext3).

Je to žurnálovací souborový systém, tedy při výpadku je větší pravděpodobnost, že data na disku zůstanou konzistentní.

ReiserFS je založen na rychlém balancovaném stromu, což zrychluje práci s velkým množstvím souborů v adresáři. Další výbornou vlastností je, že je možné uložit několik malých souborů (nebo zbytků velkých souborů, které se nevešly do celých bloků) do jednoho bloku (jiné souborové systémy včetně ext2, ext3, FAT, NTFS každý blok vyhrazují pro určitý soubor, soubor může mít více bloků, ale ne naopak), takže na disku nevzniká zbytečně mnoho velkých „nedosažitelných“ děr. Nevýhodou je možnost snížení výkonu systému, který tento souborový systém částečně vylepšuje různými technikami používanými v databázových systémech. Pro systém, kde pracujeme především s velmi malými soubory, je to však dobrá volba.

Další zajímavou vlastností je možnost změny velikosti partition s tímto souborovým systémem, a to dokonce bez nutnosti odmontování systému (jistější je ale systém předem odpojit a po změně znovu připojit).

Práci systému lze zrychlit také volbou určitých parametrů při připojování disku (obvykle v souboru fstab), například volba `notail` zakáže ukládání konců více

souborů do jednoho bloku. Tím sice ztratíme část místa na disku (v době běžně používaných 80GB disků to není zase až taková hrůza), ale systém se zrychlí.

XFS je žurnálovací souborový systém, který se svými přístupovými algoritmy snížit zatížení systému způsobené používaním žurnálování. Navíc je to 64-bitový souborový systém (adresa je uložena v 64 bitech, narozdíl od jinde obvyklých 32 bitů), takže velikost souboru a velikost celého souborového systému může být úctyhodná.

Má mnoho zajímavých vlastností, jedna z nich je *realtime subvolume*, která dovoluje procesům rezervovat si k souboru přístupové pásmo v určité šíři (B/s). To je velmi praktické například při práci s multimédii, kdy k souboru (např. s videem) potřebujeme stálý a rychlý přístup.

Žurnálována jsou metadata (to zvyšuje rychlost při zjišťování a opravě chyb), ale ne samotná data, takže celkově je tento souborový systém považován za méně bezpečný než ext3fs nebo ReiserFS.

8.5.4 Virtuální souborové systémy

V Linuxu stejně jako v jiných Unixových systémech se používají i souborové systémy bez vazby na konkrétní datové médium (případně v sobě sdružují přístup k více různým datovým médiím). Jsou to především *procfs* a *devfs*.

procfs zpřístupňuje veškeré informace týkající se jádra, slouží ke komunikaci s jádrem systému. Neodpovídá žádnému fyzickému datovému médiu, je připojován do adresáře `/proc`. Z hlediska uživatele jsou zajímavé především jeho podadresáře, jejichž názvy jsou PID všech běžících procesů (v takovém adresáři jsou všechny důležité informace o procesu, jehož PID je názvem adresáře).

devfs je virtuální souborový systém typický právě pro Linux, který spravuje speciální soubory zařízení uložené v adresáři `/dev`.

Další: Nejdůležitější virtuální souborový systém už známe, je to VFS. Je to důležitá součást jádra systému, přes kterou procesy komunikují s konkrétními souborovými systémy. Existují však i další virtuální souborové systémy sloužící různým účelům, mají především zjednodušit přístup k různým virtuálním zařízením.

8.5.5 Srovnání Linuxových souborových systémů

Pod Linuxem můžeme používat samozřejmě i další souborové systémy, zde jsme mluvili pouze o nejpoužívanějších souborových systémech pro lokální disky. Nelze říci, který z uvedených souborových systémů je lepší nebo horší, každý má své výhody i nevýhody. Výhodou může být žurnálování, které ale může (nemusí) snižovat výkon systému, bohužel i u žurnálovacích souborových systémů se stává, že se při výpadku data ztratí, i když ne tak často jako u systémů bez žurnálu.

V následující tabulce je porovnání systémů podle kritérií, která přímo v kapitolách uváděna nebyla (údaje jsou pouze orientační, čísla jsou bohužel různá v různých zdrojích):

	ext2fs	**ext3fs**	**ReiserFS**	**XFS**
Max. velikost partition	4 TB	4 TB	16 TB *)	18∗210 PB
Velikost bloku	1 - 4 kB	1 - 4 kB	až 64 kB	512 B - 64 kB
Max. velikost souboru	2 GB	2 GB	až 210 PB *)	9∗210 PB

*) záleží na verzi souborového systému

Tabulka 8.7: Srovnání vlastností souborových systémů pro Linux

PB znamená PentaByte, 1 PB = 1024 TB (podle jiných zdrojů = 1000 TB). Udané hodnoty je však nutné brát s rezervou, na tom, jak velké soubory může souborový systém ukládat, záleží také na VFS.

KAPITOLA 9

Správa disků

V této kapitole se budeme podrobněji zabývat správou disků především z hlediska instalace a provozu operačních systémů. Ukážeme si, jakou strukturu má pevný disk, pod jakými názvy se k diskům přistupuje v různých operačních systémech, jaké nástroje používáme pro správu disků, které zavaděče operačních systémů je vhodné volit pro různé konfigurace hardwaru a instalovaných operačních systémů, a na závěr se podíváme na možnosti instalace více operačních systémů vedle sebe nebo jejich emulaci.

9.1 Problémy s BIOSem

Přístup k disku původně probíhal přes služby BIOSu. BIOS ve své standardní (starší) podobě však nedokáže zpřístupnit části disku nad 8 GB (1024 cylindrů), proto v tomto starším BIOS rozhraní nemohou být používány větší disky. To se týká disků s rozhraním IDE, disky s rozhraním SCSI tento problém nemají.

Novější rozhraní BIOSu již nabízí přístup nad tuto hranici (používá technologii LBA – Logical Block Addressing), není však zpětně kompatibilní a operační systémy vyvinuté bez podpory tohoto novějšího rozhraní nemohou tyto služby využívat. Týká se to například MS-DOSu a Windows s DOS jádrem (starších verzí).

Novější operační systémy problémy s BIOSem řeší především tak, že pro přístup na disk používají místo služeb BIOSu vlastní ovladače. BIOS je ale potřeba před vlastním zavedením takového operačního systému, proto (teoreticky) zaváděcí záznam operačního systému musí být na prvních 1024 cylindrech[1].

[1] Dá se to řešit volbou vhodného zavaděče systému, viz dále.

9.2 Základní pojmy

Jeden fyzický disk může být rozdělen na více *oddílů* (partitions, oblastí, svazků, ...). Oddíly mohou být *primární* (primary partition), jeden z nich může být označen jako *rozšířený* (extended partition) a dále rozdělen na prakticky jakékoliv množství oddílů – „pododdílů", které nazýváme logické disky. Na každém oddílu může být nainstalován operační systém (pak jde o bootovací oddíl) nebo to může být datový oddíl (bez operačního systému).

Dnes existují dva základní druhy disků: běžné disky MBR (Master Boot Record) a disky s dělením GPT (GUID Partition Table) používané na platformě Itanium (pro 64-bitové servery, na desktopu momentálně přístupné pouze z Windows XP 64-bit).

MBR disky mohou mít maximálně 4 primární oddíly, z nichž jeden může být označen jako rozšířený, v něm lze vytvořit jakýkoliv počet logických disků. GPT disky mají trochu jinou strukturu, mohou obsahovat až 128 oddílů (rozšířené oddíly a logické disky nejsou používány).

Dále se budeme věnovat pouze MBR diskům.

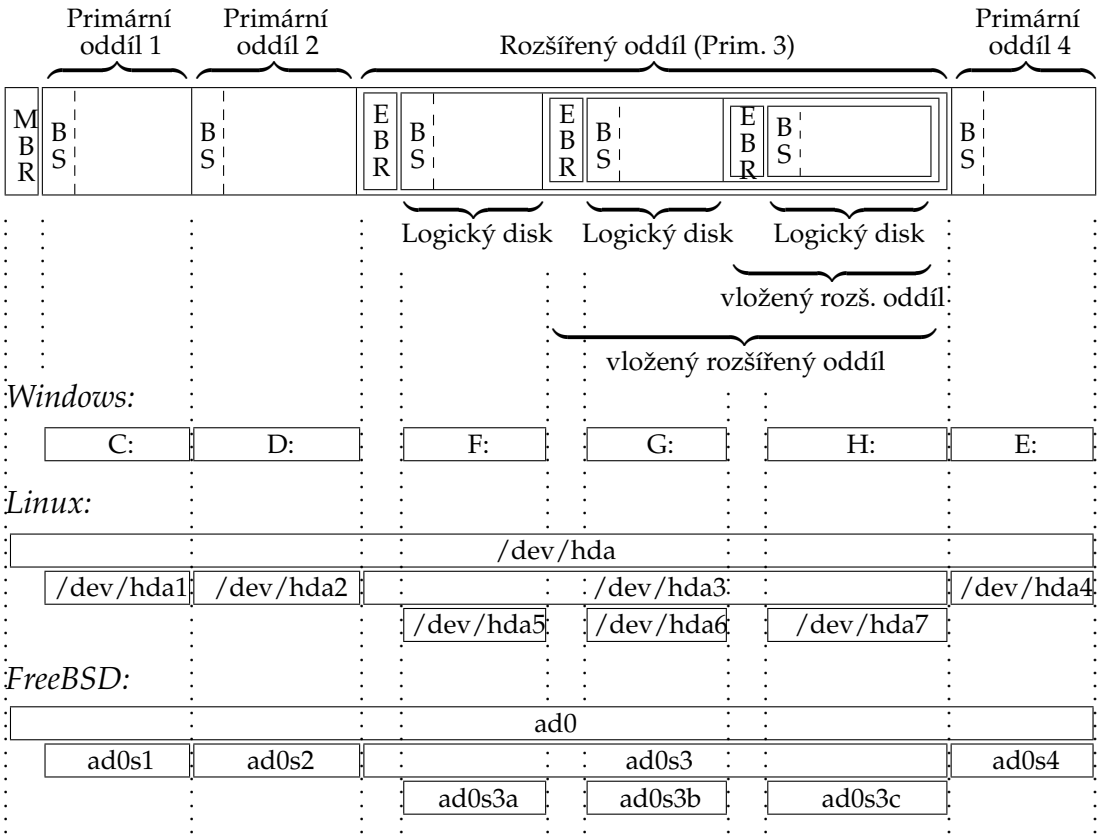
Na platformě Windows lze používat tzv. *dynamické svazky*. Tato technologie je však obecně přístupná pouze z Windows, nikoliv z jiných operačních systémů, proto pokud chceme mít takto upravené oddíly disku přístupné např. z Linuxu, nedoporučuje se použít pro ně tento formát[2]. Dynamický svazek nemusí být fyzicky souvislý, může se dokonce rozkládat na více pevných discích. Pro správu se používá skrytá databáze. Dynamické disky podporují prokládání, zrcadlení a další optimalizační a bezpečnostní funkce[3]. Pevný disk, na kterém jsou nějaké dynamické svazky, se nazývá *dynamický disk*.

9.3 Struktura disku

Na obrázku 9.1 je naznačena struktura disku, který byl rozdělen takto: nejdřív jsme vytvořili dva primární oddíly, pak rozšířený oddíl, a potom další primární oddíl. Pak jsme v rozšířeném oddílu vytvořili postupně tři logické oddíly. Postupně si popíšeme některé části této struktury.

[2]Ve skutečnosti se dynamické disky používají i v Unixových systémech, ale v trochu jiném významu – více pevných disků spojených do RAID pole se specifickým řízením přístupu.

[3]Tyto funkce lze však implementovat i bez dynamických svazků, takže dynamické svazky nejsou „životně důležité" pro servery, i když pro ně jsou podobné vlastnosti velice užitečné.



Obrázek 9.1: Struktura MBR disku a označení v různých operačních systémech

Zkratka MBR znamená *Master Boot Record* – hlavní zaváděcí záznam disku. Zde najdeme hlavní zaváděcí záznam (prostě instrukce pro BIOS, které říkají, co se má stát, když je počítač nastartován a má se zavést operační systém), a také tabulku rozdělení disku. Hlavní zaváděcí záznam zjistí, který oddíl je označen jako *aktivní*, a pak se pokusí z tohoto oddílu zavést operační systém (spustí zaváděcí program tohoto oddílu).

Tabulka rozdělení disku (Partition Table) zabírá na disku 64 B. Má čtyři záznamy (jeden pro každý primární oddíl – rozšířený oddíl také považujeme za primární), a v každém záznamu jsou o příslušném primárním oddílu tyto informace:

- zda je aktivní (aktivní oddíl má zde hexadecimální číslo 80, jinak 0),
- kde se nachází boot sektor oddílu (tedy adresa začátku oddílu),
- typ oddílu, způsob jeho organizace (zde se rozlišuje, zda jde o rozšířený oddíl nebo o oddíl s nějakým konkrétním souborovým systémem, každý souborový systém má své identifikační číslo),

- další metriky (adresa konce oddílu, počet sektorů od MBR k začátku oddílu, velikost oddílu v sektorech).

Zkratka BS znamená *Boot Sector* – zaváděcí sektor oddílu. Je to část oddílu, do které běžně uživatel nemá přístup. V případě, že je na tomto oddílu nainstalován některý operační systém, najdeme zde zaváděcí program tohoto operačního systému.

Zaváděcí program (boot loader, zavaděč) je program, jehož úkolem je zavést operační systém při startu počítače nebo v případě instalace více operačních systémů na disku umožnit výběr jednoho operačního systému ze seznamu a spustit zaváděcí program vybraného operačního systému (pak se nazývá *boot manažer*). Pokud máme instalováno více operačních systémů na více oddílech, při startu počítače se z MBR spustí zaváděcí program pouze jednoho z nich. Tento program by pak měl umožnit přístup i k zaváděcím programům ostatních systémů.

Zkratka EBR znamená *Extended Boot Record* – zaváděcí záznam rozšířeného oddílu. Je to obdoba MBR, obsahuje podobné informace. Rozšířený oddíl obsahuje jeden logický disk (s ním se pak ve většině případů zachází stejně jako s primárními oddíly), a pokud tento logický disk nezabírá celý rozšířený oddíl, ve volném místě může být vnořený rozšířený oddíl s vlastním EBR. Ten opět může obsahovat kromě logického disku další vnořený rozšířený oddíl, atd. Rozšířené oddíly lze vnořovat prakticky do jakékoliv úrovně a tak tvořit jakýkoliv počet logických disků a tím i oddílů. Každý logický disk také má svůj boot sektor.

Způsob označování fyzických disků v některých Unixových systémech je v tabulce 9.1.

IDE kanál	Linux	FreeBSD
1 Master	/dev/hda	ad0
1 Slave	/dev/hdb	ad1
2 Master	/dev/hdc	ad2
2 Slave	/dev/hdd	ad3

Tabulka 9.1: Označení fyzických disků v některých Unixových systémech

9.4 Nástroje pro správu disků

Pro MBR disky platí, že mohou mít nejvýše 4 primární oddíly (primary partition), z nichž jeden může být rozšířený (extended partition) a v něm jakýkoliv počet

logických disků (logical volume, logical disk, logical partition, ...). Oddíl může být systémový (s instalovaným operačním systémem) nebo datový (obsahuje pouze data) nebo odkládací (swap, obvyklé u Unixových systémů).

Nástroje pro správu disků by především měly umět vytvářet a rušit všechny tyto druhy oddílů. Často mívají ještě jiné funkce. Následují nejznámější programy pro takovou základní správu disků. Některé z nich jsou určeny pro práci v určitém operačním systému a jsou s ním dodávány, jiné jsou v tomto ohledu nezávislé.

fdisk firmy Microsoft: Název fdisk je u programů s tímto určením celkem obvyklý. fdisk od firmy Microsoft, který je dodáván s Windows, má jen velmi omezené vlastnosti. Pracujeme v textovém režimu, na pevném disku můžeme vytvořit pouze jediný primární oddíl a jediný rozšířený, v tom pak jakékoliv množství logických. Pokud chceme na některý oddíl instalovat Windows, pro vytvoření tohoto oddílu bychom měli použít fdisk od Microsoftu a ne například Linuxový, protože v některých případech dochází k nekompatibilitám a zde instalované Windows by se mohly stát nepoužitelnými (opravdu nepoužitelnými).

Neumožňuje nedestruktivní změnu hranic oddílu (na modifikovaných oddílech jsou data prakticky zničena), oddíly, se kterými pracujeme, nesmí být používány (tj. například z nich nesmíme spouštět fdisk a jím následně tento oddíl modifikovat). Obvyklé použití je ze startovací nebo záchranné diskety. Protože se programátoři firmy Microsoft moc nehrnou do „znovuvytvoření“ tohoto programu a změny provádějí pouze přidáváním kódu k existujícímu, důsledkem jsou některé drobné nesrovnalosti (například při práci s velkými disky se zobrazuje trochu jiná – menší kapacita oddílu než jaká je ve skutečnosti).

Ve Windows 2000/XP existuje nástroj s obdobnými schopnostmi, ale s grafickým rozhraním. Spustíme ho příkazem `diskmgmt.msc` nebo v grafickém rozhraní ho najdeme v konzole Správa počítače (kontextové menu ikony Tento počítač, volba Spravovat). Narozdíl od samotného fdisku zde navíc můžeme pro oddíl zvolit určitý souborový systém (FAT32, NTFS) – fdisk od Microsoftu je vlastně jedním z mála programů, které od souborových systémů dávají ruce pryč, k tomuto účelu v textovém režimu musíme zvolit příkaz `format` nebo v grafickém režimu kontextové menu daného disku.

fdisk dodávaný s Linuxem: Program fdisk dodávaný s Linuxem sice také přijímá příkazy v textovém režimu (vybíráme z textového menu tisknutím kláves na klávesnici), umí však mnohem více. Kromě vytváření a rušení oddílů můžeme určit souborový systém oddílu (jsou podporovány různé souborové

systémy včetně nelinuxových a také swap pro Linux). Existuje „uživatelsky přívětivější " varianta – cfdisk.

Program fdisk spouštíme obvykle s určením pevného disku, se kterým chceme pracovat, např. `fdisk /dev/hda`.

fips je prográmek, který umí zmenšit Windows oddíly, je to vhodný doplněk fdisku Microsoftu. Je dodáván také s některými Linuxovými distribucemi, což je užitečné, když potřebujeme na disku zmenšit místo, které do té doby uzurpovaly Windows, a nainstalovat tam jiný operační systém.

Partition Magic je komerční program pracující pod Windows. Vyznačuje se propracovaným grafickým prostředím, umožňuje kromě vytváření a rušení oddílů také změnu jejich velikosti (nedestruktivní, data zůstanou zachována).

GNU Parted, QtParted, GParted jsou programy pracující pod Linuxem. GNU Parted je nejjednodušší, kromě vytváření, rušení a změny velikosti oddílů umožňuje například také vytvoření obrazu disku (oddílu). GParted (Gnome Partition Editor) je program dodávaný s prostředím Gnome. Jeho grafické rozhraní a možnosti jsou podobné Partition Magicu. QtParted je obdobou GParted pro prostředí KDE.

Další: Ranish Partition Manager, DiskDrake, ..., a také některé boot manažery v sobě zahrnují možnost pracovat s oddíly disků.

K programům pro správu disku můžeme řadit také programy vytvářející obraz disku (diskového oddílu). Některé programy dokážou pracovat pouze s oddíly s určitými souborovými systémy, jiným je to celkem jedno. Tato funkce může být zahrnuta v univerzálnějším programu určeném obecně pro správu disků, nebo lze použít specializované programy. Z nejznámějších pro Windows: Norton Ghost, Drive Image, True Image, Power Quest, Drive Backup. V Linuxu je to především Partimage, ale najdeme zde mnoho dalších a tuto funkci mají prakticky všechny nástroje pro správu disků.

Existuje poměrně mnoho linuxových distribucí určených přímo pro správu (záchranu) disků, například System Rescue CD (zachraňuje nejen Linux, ale i Windows, jsou zde dokonce i nástroje pro práci s registrační databází).

9.5 Zaváděcí programy

Každý operační systém obvykle obsahuje alespoň jeden zaváděcí program. Úkolem zaváděcího programu je především tento operační systém zavést, tedy ve stanoveném pořadí spustit procesy potřebné k běhu systému včetně procesů jádra. To je

ovšem hodně zjednodušený popis činnosti zaváděcího programu, protože každý operační systém má pro své spouštění určitá specifika a tedy i každý zaváděcí program pracuje úplně jiným způsobem, navíc to, co se provádí před vlastním spuštěním jádra, často ještě nelze nazvat procesem (nemá své PID, nemá operačním systémem přidělené systémové zdroje, . . .).

Zaváděcí program (nebo jeho spouštěcí část) je v boot sektoru některého oddílu (oddílu, na kterém je nainstalován příslušný operační systém) nebo v MBR, podle toho, zda chceme či nechceme použít primárně jiný zaváděcí program, který by na něj dále mohl odkazovat.

Protože dnešní zavaděče jsou poněkud rozsáhlejší a kromě vlastního programu potřebují také prostor pro své konfigurační soubory, na uvedených místech je pouze základní část a odkaz na soubory se „zbytkem“. Například pro NTLoader (zavaděč Windows řady NT) najdeme na systémovém disku (na disku, kde je instalován příslušný operační systém) program ntldr a dále „pomocné soubory“ bootfont.bin a boot.ini.

Probereme si postupně některé zavaděče a jejich vlastnosti.

Zavaděč Windows 9x/ME je jednoduchý zavaděč, který kromě svého vlastního operačního systému dokáže zavést nanejvýš starší verzi Windows (MS-DOSu), např. MS-DOS + Windows 3.1. Konfigurace se provádí v souboru BOOT.INI na disku C:, zavaděč se vždy nainstaluje do MBR a boot sektoru disku C:. Vyžaduje, aby disk C: byl primární oddíl. Pracuje v textovém režimu, neumožňuje konfiguraci (pseudo)grafického prostředí.

V omezené míře lze ve starších Windows používat menu určující, co má být spuštěno (včetně Windows), a to v konfiguračních souborech CONFIG.SYS a AUTOEXEC.BAT (informace viz [32], [33]).

Zavaděč Windows NT/2000/XP (NTLoader) umí spouštět svůj operační systém i z jiného disku než C:. Je to jednoduchý boot manager, který však dokáže pracovat pouze s Windows oddíly („vidí“ pouze oddíly se souborovým systémem FAT nebo NTFS). Konfigurace se provádí v souboru BOOT.INI na disku C: (i v případě, že operační systém zavaděče je na jiném disku).

Časté použití je Windows 9x/ME na disku C: a Windows 2000/XP na disku D: s tím, že zavaděč umožní při svém startu vybrat si z těchto dvou systémů. Instaluje se vždy do MBR a boot sektoru příslušného disku se systémem (např. D:). O konfiguraci grafického/pseudografického prostředí platí totéž, co u zavaděče Windows 9x/ME.

Informace o využití tohoto zavaděče pro zavedení jiného než „Microsoftího“ systému jsou na [31].

Pokud je tento zavaděč (vlastně i předchozí) instalován až po instalaci jiného zavaděče (např. Linuxového), bez skrupulí přepíše starší odkaz v MBR, takže se často proti vůli uživatele stane primárním zavaděčem. Důsledkem je pak nemožnost spustit původní zavaděč a tím i původní operační systém. Tento problém je sice řešitelný, ale je lepší mu předejít vhodným členěním posloupnosti instalace systémů nebo alespoň včasným zálohováním původního zavaděče na externí médium (disketu).

LILO (LInux LOader) je zavaděč používaný pro Linux na HW platformě x86[4]. Je to univerzální zavaděč schopný spolupráce s prakticky všemi známějšími souborovými systémy, tedy nebývá problém s jeho používáním. Narozdíl od jiných známých zavaděčů dokáže zavést i takový operační systém, jehož boot sektor (jeho adresa) je za 8 GB (viz str. 123), protože používá LBA (Logical Block Addressing), pouze musí být splněna podmínka přítomnosti tohoto boot sektoru na některém z prvních dvou pevných disků (hda, hdb).

Konfigurace je uložena v souboru `/etc/lilo.conf`, konfiguruje se zde především obsah menu (které systémy se mají zavést a kde je hledat). Co se týče grafiky, máme na vybranou mezi LILO v textovém režimu a LILO v grafickém režimu. Konfiguraci všech vlastností zavaděče lze obvykle provádět pomocí grafických nástrojů dodávaných s grafickým prostředím operačního systému. O konfiguraci LILO jsou informace např. na [30].

GRUB (GRand Unified Boot loader) je zavaděč používaný pro Linux na HW platformě x86 a amd64. Je to univerzální zavaděč spolupracující se všemi běžnými souborovými systémy, stejně jako LILO.

Výhodou je výborně propracované skrývání oddílů, které může sloužit například při instalaci dalšího operačního systému, pokud tento systém chceme přesvědčit, že je instalován na první primární oddíl, i když ve skutečnosti tomu tak není (to je případ Windows 9x, které by bez tohoto mechanismu buď odmítly instalaci, nebo, což je horší, přepsaly by MBR, první primární oddíl i jeho boot sektor svými daty[5]).

[4] HW platforma s procesory Intel od i386 do Pentium IV a AMD procesory do 32-bitových včetně.

[5] Jinými slovy: když máme Linux a chceme navíc Windows 9x, vybereme pro Linux jako zavaděč GRUB, vybereme volnou primární partition na prvním pevném disku, vše, co je před ní, skryjeme, a instalujeme.

Konfigurace je obvykle uložena v souboru `/etc/grub.conf` nebo v souboru `/boot/grub.conf`, stejně jako LILO i GRUB lze v Linuxu obvykle konfigurovat v grafickém prostředí (např. program GrubConf). Při spuštění zavaděče (při zobrazení menu) se také můžeme dostat do speciálního konfiguračního módu GRUBU (jeho příkazového řádku) stisknutím klávesy $[c]$, díky tomu při změnách v konfiguraci nemusíme pokaždé restartovat, aby se změny projevily. Seznam příkazů se zobrazí po stisknutí klávesy $[TAB]$. Příkazový řádek GRUBu je možné také spustit za běhu operačního systému příkazem `grub` (pouze uživatel root). Další informace viz [29].

GRUB se standardně používá s textovým rozhraním, lze však s trochou námahy změnit. Lze také použít obrázek na pozadí – volba `splashimage` v konfiguračním souboru (nebo příslušný příkaz v příkazovém řádku GRUBu), tento obrázek však musí mít předem definované rozměry, barevnou hloubku a formát.

GRUB se vyznačuje vlastním názvoslovím týkajícím se identifikace disků. Pevné disky označuje postupně `hd0` (místo hda), `hd1` (místo hdb), . . . , oddíly na discích se značí čísly od 0. Vzniklé dvojice (pevný disk, oddíl) mohou být například

- `(hd0,0)` = první oddíl na prvním disku = hda1,
- `(hd0,1)` = druhý oddíl na prvním disku = hda2,
- `(hd1,0)` = první oddíl na druhém disku = hdb1, atd.

další linuxové zavaděče: aBoot, MILO (oba pro architekturu alpha), SILO (architektura sparc), yaBoot (architektura ppc – PowerPC), PALO (architekruta hppa), . . .

XOSL, OS Selector, EasyBoot, Smart Boot Manager, . . . jsou univerzální boot manažery. Některé komerční (např. OS Selector), jiné volně šiřitelné (např. XOSL). Každý má své specifické vlastnosti. Obvykle dovolují vybrat ze seznamu operačních systémů, po výběru spustí příslušný zavaděč. Většinou dokážou také skrývat oddíly stejně jako GRUB. Omezení se týkají většinou souborového systému nebo oddílu, na který jsou tyto programy instalovány (mnohé vyžadují instalaci na Windows oddílu s FAT, i když dokážou spustit zavaděč systému instalovaný na úplně jiném souborovém systému a oddílu).

Pokud například XOSL nainstalujeme (na Windows oddíl) a vhodně nakonfigurujeme, pak se při startu počítače zobrazí nabídka s možnostmi spuštění instalovaných operačních systémů. Po vybrání se pak spustí zaváděcí záznam

vybraného systému. Při konfiguraci určujeme, jaké systémy máme a kde se nachází jejich zaváděcí záznam (ve kterém boot sektoru), původní primární zavaděč z MBR je obvykle detekován automaticky.

9.6 Možnosti instalace operačních systémů

Dnes je obvyklé a většinou vyžadované instalovat každý operační systém na samostatný oddíl. Tento požadavek nebyl potřebný v případě, že vedle MS-DOSu s Windows 3.x jsme chtěli instalovat Windows 95 (oba systémy s DOS jádrem a vpodstatě stejným zavaděčem). Pak mohly být oba operační systémy na jednom oddílu (disky byly ostatně tak malé, že bylo škoda je nějak dělit). Pokud se o totéž pokusíme třeba v případě Windows 98 + Windows XP nebo Windows + Linux, druhý systém při instalaci přepíše ten první (nebo se instalace vůbec nespustí, když odmítneme rozdělit disk).

Pokud chceme mít instalováno více operačních systémů na jednom počítači, musíme brát ohled na požadavky těchto systémů. Například

- Windows s DOS jádrem (včetně Windows 9x/ME) vůbec nepočítají s tím, že na disku budou ještě nějaké další operační systémy, bez ptaní přepíšou MBR a boot sektor prvního primárního oddílu.
- Windows s NT jádrem (Windows NT/2000/XP) sice umožňují instalaci na jiný než první oddíl, ale měl by být primární. Navíc tento systém dokáže detekovat pouze Microsoftí operační systémy, takže pokud je v MBR záznam jiného než Windows zavaděče, odmítnou ho vzít na vědomí a tento zavaděč je prostě přepsán.
- Unixové a Linuxové operační systémy mohou být instalovány téměř na kterémkoliv oddílu včetně logického na rozšířeném oddílu, respektují zavaděče jiného systému, nebývají s nimi problémy (s určitými „černými“ vyjímkami, jako je třeba Solaris pro určitou skupinu předem nainstalovaných systémů). Konkrétní chování závisí na volbě zavděče (LILO, GRUB, . . .).

Takže pokud nechceme používat skrývání oddílů, volíme tuto posloupnost instalací (samozřejmě kterýkoliv člen posloupnosti může být vynechán):

1. Windows systémy s DOS jádrem
2. Windows systémy s NT jádrem
3. Linuxy, Unixy

Pokud máme instalováno více operačních systémů, každý z nich má svůj zavaděč. Jeden z nich je primární, jeho záznam je v MBR, a z něho jsou spouštěny ostatní zavaděče. Tak vzniká „stromová struktura“ zavaděčů, například když máme Windows 98, Windows XP a některý Linux, primární je obvykle linuxový zavaděč (např. LILO). Pokud v něm vybereme spuštění Linuxu, tato akce se provede hned, pokud ale vybereme spuštění Windows, spustí se zavaděč Windows XP, ve kterém si můžeme vybrat mezi Windows 98 a Windows XP.

Jestliže je na počítači provozováno více operačních systémů, je vhodné myslet i na to, abychom z nich měli *přístup k* našim *datům*. Také z důvodu bezpečnosti dat má být alespoň jeden diskový oddíl vyhrazen pouze pro data, souborový systém volíme takový, se kterým dokážou pracovat všechny instalované operační systémy. Linux dokáže pracovat se všemi běžnými souborovými systémy (donedávna byly problémy s NTFS, ty jsou v nejnovějších jádrech odstraněny – nebylo možné měnit délku souborů), Windows řady NT bez vhodných berliček pouze s FAT a NTFS, Windows 95 OSR2/98/ME si rozumí jen s FAT včetně FAT32, Windows 95 a starší pouze FAT16.

Windows a Linux mohou sdílet data po síti například pomocí protokolu *smb*. Obvyklé je mít Linux instalován na serveru a Windows na pracovní stanici, na Linuxu běží služba (démon) `samba`.

Co se *sdílení instalace aplikací* týče, je situace trochu horší. Ve Windows způsobují problémy především údaje v registru (registr nelze mezi různými instalacemi sdílet) a někdy také formát dynamických knihoven (může být jiný například pro Windows 98 a XP), proto je obvykle nutné aplikaci v každých Windows instalovat zvlášť (někdy je možné zvolit stejný adresář/složku pro umístění souborů aplikace, jen údaje v registrech jsou pro každou instalaci zvlášť). Různé verze Windows mohou sdílet tentýž odkládací soubor.

Více linuxových distribucí může sdílet totéž jádro (to většinou lze určit při instalaci), odkládací (swap) oddíl či soubor, případně další oddíly (třeba `\home`), za určitých okolností lze sdílet i jiné aplikace. Sdílení aplikací mezi Windows a Linuxem obvykle není možné, vyjímkou jsou multiplatformní aplikace psané v Javě nebo pomocí technologie .NET (případně v Pythonu, Lispu či v jiném interpretačním jazyku).

9.7 Emulace jiného operačního systému

Předchozí stránky se týkaly především případu, kdy máme na disku instalováno více operačních systémů a počítáme s tím, že v jednom okamžiku používáme jen

jeden z nich a při potřebě změny restartujeme systém. Někdy však potřebujeme pracovat s více operačními systémy najednou. Pak využijeme možnost emulace operačního systému, tedy použijeme program, který běží jako proces (procesy) v jednom operačním systému a simuluje běh jiného operačního systému (ten běží „v okně“).

Programy, které můžeme pro emulaci (simulaci) využít, můžeme rozdělit do několika skupin:

- virtuální počítač – provádí simulaci počítače (může jít o úplně jinou HW platformu než na které systém běží „v reálu“), na tomto počítači můžeme mít instalován jakýkoliv počet jiných operačních systémů,
- emulátor operačního systému – simuluje konkrétní operační systém,
- podsystém pro spouštění aplikací jiného operačního systému.

9.7.1 Virtuální počítače

Tento typ emulátorů je nejsložitější a často i nejpomalejší. Po instalaci obvykle můžeme nakonfigurovat simulovaný hardware (kterou HW platformu chceme používat, který hardware bude k dispozici a jak se jeho používání projeví na „skutečném“ hardwaru, například napojíme tiskárnu), dále nastavíme BIOS, a pak můžeme instalovat operační systémy. Některé programy vyžadují jisté „předupravení “ zdroje operačního systému do tzv. *image* (obraz, něco jako obraz CD).

Každý z těchto programů má své typické vlastnosti, například existují emulátory sloužící čistě k emulaci konkrétní hardwarové platformy (pro Amigu, ZX Spectrum, PowerPC, kapesní počítače – využívají především programátoři těchto zařízení, herních konzolí, apod.) nebo umožňující volit mezi několika platformami (instalace nové platformy se pak provádí instalováním příslušného modulu), případně s volbou HW platformy volíme i operační systém (na některé platformě „není z čeho vybírat“, například u Amigy).

Z nejznámějších programů:

VMWare běží pod Windows i Linuxem. Je to jeden z nejlepších a nejoblíbenějších univerzálních simulátorů, komerční.

MS Virtual PC je distribuován Microsoftem (původně byl vyvíjen jinou firmou, Microsoft tuto firmu odkoupil), běží pouze pod Windows, komerční.

Bochs běží pod Windows, je to freeware. Je to velmi dobrý program s mnoha volbami, ale poměrně složitý.

Qemu je o něco rychlejší a ovladatelnější než Bochs, běží pod Linuxem, Windows i MacOSX, je volně dostupný na Internetu.

Xen je obdoba Bochs, ale na rozdíl od něho přejímá hardwarovou platformu od počítače, na kterém běží, jinak můžeme instalovat jakékoliv operační systémy (starší verze vyžadují vytvoření obrazu tohoto systému). Oproti Bochs má výhodu také v rychlosti (nemusí simulovat veškerý hardware). Volně šiřitelný, pro Linuxy a některé Unixy.

9.7.2 Emulátory operačního systému a podsystémy

Tyto programy simulují běh konkrétního operačního systému, tedy nový operační systém samotný nemusíme již instalovat.

Pokud je emulován operační systém se vším všudy (téměř), můžeme v tomto operačním systému pracovat se vším všudy včetně konfigurace (systém běží v okně celý, v rámci tohoto okna pak jeho aplikace). Jestliže se však jedná o podsystém, účelem je především možnost spouštět aplikace určené pro „cizí“ operační systém (každá aplikace mívá obvykle vlastní okno/okna).

Z nejznámějších:

Wine je ve skutečnosti rekurzivní zkratka slov „Wine Is Not Emulator“. Autoři tímto názvem chtěli zdůraznit, že nezamýšlejí emulovat Windows, ale pouze umožnit spouštění programů psaných pro Windows v Unixových systémech (je to vlastně podsystém, zavaděč programů). Jde o vlastní implementaci Win API (rozhraní, překladové vrstvy mezi aplikací a jádrem skutečného operačního systému).

Vyznačuje se výrazně vyšší rychlostí než emulátory, udává se, že běh některých aplikací je dokonce rychlejší než ve Windows. Na stránkách tvůrců Wine je rozsáhlý seznam programů, případných problémů při jejich provozování přes Wine a jejich řešení. Některé programy bohužel takto nelze zprovoznit nebo dochází k neodstranitelným problémům (ale jde o vyjímky).

Wine najdeme prakticky ve všech distribucích Linuxu a také v mnoha dalších Unixech. Je volně šiřitelný. Pokud jde ale o aplikace, které do Wine instalujeme, musíme respektovat jejich licenci.

Cedega je komerční projekt vycházející z Wine. Oproti Wine obsahuje navíc některé komerční technologie, dokonce i přímo od firmy Microsoft. Je určen ke spouštění různých, i náročnějších programů pro Windows, je využíván především pro spouštění her.

CrossOver je také komerční varianta pro Wine, ale je určen především do kanceláří. Využívá se především pro provoz účetních a jiných ekonomických aplikací psaných pro Windows.

DosEmu, DosBox jsou programy emulující DOS pod Linuxem. Neobsahují instalaci MS-DOSu, který je zatížen licencí EULA a tedy pro tyto účely nepoužitelný, ale DosEmu využívá instalaci systému freeDOS a DosBox má vlastní implementaci DOSu (pouze nejzákladnější příkazy). Využívají se například ke zprovoznění DOSovských účetních programů (DosEmu) a her (DosBox).

KAPITOLA 10

Grafický subsystém

V této kapitole se budeme zabývat grafickým subsystémem – nástavbou, která sice není „životně důležitá“, přesto však pro mnoho uživatelů téměř nepostradatelná. Svůj grafický subsystém má většina rozsáhlejších systémů, my zde pod tímto pojmem budeme chápat grafický subsystém operačního systému.

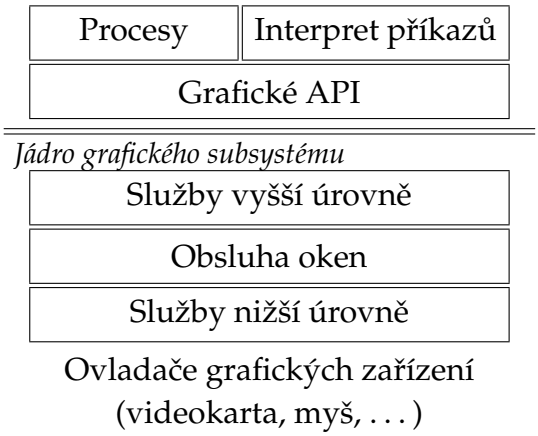
Nejdřív probereme základní pojmy související s tímto tématem včetně obvyklé struktury subsystému, a pak se podíváme na jednu z nejpropracovanějších multiplatformních implementací, X Window System.

10.1 Základní pojmy

Grafický subsystém je skupina procesů tvořící vhodné grafické rozhraní mezi uživatelem a jinými procesy včetně systémových. Jeho úkolem je zobrazovat momentální stav vybraných částí systému, zařízení a procesů a poskytovat možnosti jejich ovládání. Struktura grafického subsystému je naznačena na obrázku 10.1 na straně 138.

Služby nižší úrovně umožňují vykreslování základních grafických objektů přímo na obrazovku. Tato vrstva komunikuje přímo s ovladači grafických zařízení, především monitoru a grafické karty, nepracuje s okny.

Obsluha oken se zabývá správou oken a veškerými operacemi s nimi – vykreslování jednotlivých prvků okna, vyhrazování části okna procesu, přesouvání, změna velikosti, určování viditelnosti částí okna, obsluha základních ovládacích prvků okna, ale také udržování informací o oknech, struktury oken (bývá to hierarchická struktura), apod. Pokud systém podporuje používání virtuálních obrazovek, jejich správa je také v této vrstvě.



Obrázek 10.1: Struktura grafického subsystému

Služby vyšší úrovně dovolují pracovat s grafickými objekty na abstraktnější úrovni – kromě základních 2D objektů, jako je úsečka, čtverec, kružnice, se zde definují také některé 3D objekty a operace nad nimi, to záleží na konkrétním operačním systému.

Grafické API je sada služeb vykreslujících a umožňujících ovládat „složitější" grafické prvky – menu, tlačítka, dialogová okna, atd.), je obvykle realizováno sadou knihoven.

Interpret příkazů je vlastní grafické uživatelské rozhraní (GUI), se kterým pracuje přímo uživatel (pracovní plocha, tlačítko pro přístup k nástrojům systému a instalovaným aplikacím, ikony pro přístup k nástrojům systému, kontextová menu jednotlivých prvků, atd.)

Okno je část obrazovky, obvykle ve tvaru pravoúhelníku (nemusí to být pravidlem). Každé okno má svého vlastníka, což je proces toto okno využívající (a obvykle zároveň jeho tvůrce), v případě hierarchické struktury oken určujeme kromě vlastníka také rodičovské okno. Důležitými vlastnostmi okna jsou jeho *umístění* (souřadnice levého horního rohu okna) a *rozměry* (šířka a výška), a dále *plátno*, tedy vnitřní obsah okna. Pracovní plocha okna je oblast okna, kterou může využívat vlastník, je to tedy plátno okna. Okno se může pohybovat po pracovní ploše svého rodičovského okna.

Pracovní plocha obrazovky je oblast obrazovky, na které mohou být umístěny prvky grafického rozhraní včetně oken. Může být větší než obrazovka, zobrazuje se tedy pouze ta část pracovní plochy obrazovky, která se kryje se zobrazitelnou částí obrazovky (totéž platí o pracovní ploše okna). Obvykle jsou její rozměry několikanásobkem rozměrů obrazovky. Obrazovka (resp. její pracovní plocha) může být

chápána jako druh okna, které je rodičovským oknem pro všechna okna umístěná na pracovní ploše okna.

10.2 X Window System

X Window System (pozor, ne X Windows!!!) je multiplatformní síťový grafický systém používaný jako nástavba operačního systému. Vznikl v MIT (Massachussets Institute of Technology) v 80. letech 20. století, u tohoto systému se inspirovali tvůrci systému Apple Macintosh i Windows. Účelem bylo vyvinout grafické prostředí snadno přenositelné na různé platformy a pracující po síti – zpracování (výpočet) probíhá na serveru, zobrazování probíhá na klientském počítači (i když server a klient může být jeden a tentýž počítač).

Vlastnosti:

- Je postaven na modelu klient-server. Server je v tomto případě stroj zobrazující data, klient je kterýkoliv program, který posílá serveru příkazy související s grafikou (co kam se má zobrazit apod.). Paradoxně takovýmto serverem bývá běžný klientský počítač a klientem může být třeba aplikace běžící na serveru v síti, takže pojmy server a klient nelze zaměňovat s pojmy používanými v počítačových sítích. Často se z důvodu odlišení používají pojmy *X server* a *X klient*.
- Okna jsou uspořádána do stromové hierarchie. Okno je potomkem toho okna, z něhož bylo spuštěno. Kořenem stromu je *kořenové okno*, které odpovídá ploše. Kontextové menu kořenového okna se někdy také nazývá kořenové menu (kořenová nabídka), většinou slouží k rychlejšímu spouštění nejpoužívanějších programů a pro snadný přístup ke konfiguračním nástrojům.
- X Window System má vrstvenou strukturu. *Nejnižší úroveň* komunikuje s vrstvou přistupující přímo k hardwaru (grafická karta a monitor, myš, klávesnice, zvuková karta apod.), je závislá na hardwaru, jsou zde funkce komunikující s těmito zařízeními. *Vyšší vrstva* již obsahuje základní grafické funkce jako je vykreslení bodu, kružnice apod. *Třetí vrstva* je již naprosto nezávislá na hardwaru a dalších nastaveních včetně rozlišení obrazovky, obsahuje funkce pro ovládaní GUI na abstraktní úrovni, jako například práci s fonty a funkce pro základní práci s okny a jejich evidenci ve stromové struktuře.
- Přehledná struktura X Window umožňuje tento systém jednoduše rozšiřovat, proto pokud operační systém používá tento grafický subsystém, jeho uživatel

má na výběr z mnoha variant prostředí, která navíc obvykle lze rozsáhle konfigurovat.

X Window je dnes ve verzi X11R6 a nepředpokládá se další převratnější vývoj. Číslo 11 je verze protokolu používaného ke komunikaci mezi X serverem a X klientem, 6 je verze samotného X Window (ve skutečnosti 6.x). Systém X Window v této verzi má několik variant, na Linuxech se používá některá ze dvou uvolněných verzí – *XFree86* nebo *X.org*. Postupně se přechází na X.org, a to z důvodu pružnějšího vývoje a vhodnější licenční politiky.

X Window neurčuje přesný vzhled jednotlivých grafických komponent, nabízí pouze postup, jak nějakého vzhledu dosáhnout. Proto kromě samotného systému X Window používáme vždy některý z programů nazývaných *správce oken*. Správce oken již přímo určuje vzhled jednotlivých grafických prvků (horní lišta okna, vzhled tlačítek, apod.), vykresluje a spravuje menu aplikací, spravuje okna (od toho se ostatně jmenuje), tj. změnu velikosti, přesouvání, minimalizaci, atd., spravuje ikony, plochu, virtuální plochy, atd.

Veškeré grafické prvky používané správci oken jsou uloženy v tzv. *widget knihovnách*, sadách prvků. Widgety jsou drobné prvky, ze kterých se skládá grafické prostředí – tlačítka, posuvníky na okraji oken, menu, kontextová menu, textová pole, atd. Každý správce oken má svou widget knihovnu, kterou používá, a tím je dán víceméně jednotný vzhled oken u aplikací psaných pro určitého správce oken.

Každý programátor píšící aplikaci pro X Window si může vybrat widget knihovnu, kterou jeho program bude používat. Protože programátoři píšou programy obvykle pro určitého správce oken, vybírají si jeho widget knihovnu. Taková aplikace obvykle bez problémů funguje i pod jiným správcem oken, ale její vzhled už tak „nezapadá“ (nemusí fungovat tehdy, když není příslušná widget knihovna nainstalovaná, ale to se tak často nestává).

Tento problém se částečně vyřešil vznikem *desktopových prostředí*. Desktopové prostředí je správce oken plus spousta doprovodných aplikací naprogramovaných s použitím widget knihovny tohoto správce oken. Doprovodné aplikace pokrývají softwarovou potřebu (nejen) běžného uživatele, ale ten samozřejmě může instalovat jakýkoliv další software.

Desktopové prostředí se dodává se spoustou aplikací, ale to neznamená, že správci oken, kteří nejsou „obklopeni“ žádným desktopovým prostředím, nejsou dodáváni s žádnými doprovodnými aplikacemi. Obsahují obvykle nějaké konfigurační nástroje a několik jednoduchých aplikací. Ostatně samotný systém X Window je dodáván zároveň s jednoduchými aplikacemi demonstrujícími některé jeho vlastnosti, například xclock (grafické hodiny), xcalc (kalkulačka), xload (zobra-

zuje zatížení systému), xterm (terminál), xkill („odstřelení“ některé aplikace), atd. V některých osekanějších Unixech či Linuxech některé tyto aplikace nemusí být zahrnuty.

Někteří správci oken, desktopová prostředí a widget knihovny:

- Nejstarší widget knihovna pro X Window je *Athena*, používá ji nejstarší správce oken *twm* (Tab Window Manager). Není esteticky moc dokonalá a má trochu zvláštní způsob ovládání některých prvků (například okno získává zaměření, pokud nad ně najede myš, není třeba klepnout na jeho plochu, trochu jinak se zachází s posuvníky, změnou velikosti oken, atd.).
- Desktopové prostředí *CDE* (Common Desktop Environment, je komerční, používané na komerčních Unixech) je postaveno na správci oken *Motif* se stejnojmennou widget knihovnou.
- Desktopové prostředí *KDE* (K Desktop Environment) má správce oken nazvaného *K Window Manager* používajícího widget knihovnu *Qt* (*[kjůt]*).
- Desktopové prostředí *Gnome* používá správce oken *Metacity* používá widget knihovnu *GTK+*.

Pokud po startu operačního systému není hned spuštěno grafické prostředí (máme spuštěnu pouze textovou konzoli), pak X Window spustíme příkazem `xinit` nebo `startx`. Pak se podle údajů v konfiguračních souborech spustí také příslušný správce oken a případně desktopové prostředí.

Další informace o nejpoužívanějších správcích oken a desktopových prostředích jsou ve skriptech pro cvičení.

10.3 Technologie rozšiřující možnosti grafiky

Vývoj v oblasti hardware, a tedy i v oblasti grafických karet, jde neustále kupředu. Grafické subsystémy samotné tomuto tempu nestačí a proto vznikají a vyvíjejí se softwarové technologie, které rozšiřují jejich možnosti. Jde obvykle o podsystémy či sady knihoven, které zpřístupňují nebo usnadňují přístup programátorům graficky (či multimediálně) náročnějších aplikací k pokročilejším funkcím různých zařízení.

Nejznámějšími technologiemi rozšiřujícími možnosti grafických systémů jsou OpenGL a DirectX, ale existují i další, specifičtější, například DirectFB. OpenGL a DirectX si navzájem konkurují v oblasti 3D grafiky. Tyto technologie se používají zejména při programování her, aplikací virtuální reality, CAD programů, vědecko-technické vizualizace, 3D modelování a dalších náročných multimediálních aplikací.

Pokud je aplikace psána s pomocí knihoven některé z těchto technologií, pak musí být tyto knihovny přítomny i v systému uživatele. Protože však jde obvykle o technologie volně dostupné na Internetu, není to až takový problém.

Účelem je především akcelerace (urychlení) přístupu ke grafickému (obecně multimediálnímu) hardwaru. Protože existuje mnoho druhů grafických zařízení (týká se to především grafických karet), a každé má trochu jiné vlastnosti, musí být to, co není přímo podporováno dotyčným hardwarem, softwarově emulováno. Například pokud grafická karta nemá podporu 3D, musí tuto funkci „dodat" OpenGL nebo DirectX (to rozhraní, které programátor aplikace vyžadující 3D používá), místo aby byly přímo využívány funkce karty.

OpenGL (Open Graphics Library) je standard pro API pro tvorbu aplikací počítačové grafiky (prostě sada knihoven s API funkcemi). Projekt je od počátku vyvíjen jako nezávislý na hardware, operačním systému a programovacím jazyce, a také volně šiřitelný. Samotný projekt se soustřeďuje pouze na grafiku (včetně 3D).

Nezávislost na hardware znamená, že to funguje, ať už používáme jakýkoliv hardware, a nové vlastnosti hardwaru se v OpenGL implementují jako extensions (rozšíření) – rozšiřující moduly, není třeba přepisovat celý systém.

Nezávislost na operačním systému znamená, že OpenGL je snadno přenositelný na různé operační systémy (tj. nevyužívá žádné konkrétní funkce jádra operačních systémů), což se také využívá, existují porty – varianty pro snad všechny používanější operační systémy včetně Unixů a Windows.

OpenGL je také nezávislý na zvoleném programovacím jazyce, protože knihovny tohoto systému jsou psány nikoliv objektově, ale strukturovaně a rozhraní přístupových funkcí je dostupné ze všech běžných programovacích jazyků (datové typy návratových hodnot, parametrů, apod.).

Toto rozhraní používá mnoho náročnějších her pro Linux, ale také grafické knihovny (Mesa) a grafické aplikace (například Blender, Maya). Je součástí většiny dnes používaných operačních systémů včetně BSD, Linuxu a Windows (zde ovšem ve starší verzi), je dostupná z mnoha programovacích jazyků (nebo existuje možnost přidání jako modulu) – GLT pro C++, PHP OpenGL, AdaOpenGL, GTK+ OpenGL Toolkit, GtkGLExt, OpenGL.NET (pro C#) , dokonce takto mohou být vytvářeny i objekty ActiveX (Graphcontrols[1]).

DirectX je knihovna pro tvorbu multimediálních aplikací vytvořená Microsoftem, dnes se používá verze 9. Je určena pouze pro Windows, na jiném operačním systému

[1] viz `http://www.freshmeat.net`

nefunguje. Tato technologie je částečně implementována ve Wine a jeho klonech, proto s určitými omezeními lze provozovat aplikace napsané s použitím DirectX také na Unixových systémech.

Narozdíl od OpenGL se DirectX nesoustřeďuje jen na grafiku, ale celkově na multimédia včetně zvuku. Má několik částí, například DirectDraw (2D grafika), Direct3D (3D grafika), DirectSound (přehrávání zvuků wave), atd.

DirectX je považován za objektový (i když ne plně) systém, používají se COM objekty. Výhodou tohoto přístupu je samozřejmá kompatibilita s COM objekty ve Windows a s technologií ActiveX, nevýhodou je ne právě snadná manipulace s COM objekty.

Ve Windows je možné provádět základní konfiguraci DirectX v nástroji *Nástroj pro diagnostiku DirectX*, který spustíme příkazem `dxdiag.exe` nebo se k němu dostaneme v nástroji *Systémové informace* (Start - Programy - Příslušenství - Systémové nástroje - Systémové informace nebo spustit program `msinfo32.exe`), v menu Nástroje - Nástroj pro diagnostiku DirectX.

DirectFB je knihovna obdobná OpenGL (akcelerace grafického hardwaru), ale pouze pro 2D grafiku. Používá se především na kapesních počítačích s Linuxem (ale je podporována také na operačních systémech BSD a MacOSX), obchází knihovny X Window a přistupuje přímo ke grafickému hardwaru (ovladačům).

Používá se například při vykreslování fontů, akceleraci grafických karet a operací jimi podporovaných (vykreslování jednoduchých objektů, stínování, barvy, alfa-kanál, apod.), práci s formáty obrázků (PNG, GIF, JPG, atd.), videoformáty včetně podpory Flashe, ke správě vstupních zařízení (klávesnice, myši s různým rozhraním, joystick).

PŘÍLOHA A

Výstupy některých diskových nástrojů pro Windows

```
Microsoft Windows XP [Verze 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>chkdsk
Systém souborů je typu NTFS.

UPOZORNĚNÍ!  Nebyl zadán parametr F.
Nástroj CHKDSK proběhne v režimu jen pro čtení.

Program CHKDSK ověřuje soubory (fáze 1 z 3)...
Ověření souboru dokončeno.
Program CHKDSK ověřuje rejstříky (fáze 2 z 3)...
Ověření rejstříku dokončeno.
Program CHKDSK obnovuje ztracené soubory.
Program CHKDSK ověřuje popisovače zabezpečení (fáze 3 z 3)...
Ověření popisovače zabezpečení bylo dokončeno.
Program CHKDSK nalezl volné místo označené jako přidělené v
bitové mapě tabulky MFT.
Oprava chyb v rastru svazku.
Systém Windows nalezl chyby v systému souborů.
Spusťte nástroj CHKDSK s přepínačem /F (opravit) a pokuste se je opravit.

  15358108 kB místa na disku celkem.
   9601772 kB v 102634 souborech uživatele.
     30344 kB v 9969 rejstřících.
         0 kB v chybných sektorech.
    179832 kB používá systém.
     65536 kB zabírá soubor s protokolem.
   5546160 kB na disku je volných.
```

```
      4096 bajtů v každé alokační jednotce
   3839527 alokačních jednotek na disku celkem.
   1386540 volných alokačních jednotek


C:\>fsutil fsinfo
---- Podporované příkazy FSINFO ----

drives          Vytvoří seznam všech jednotek.
drivetype       Odešle dotaz na typ dané jednotky.
volumeinfo      Odešle dotaz na informace o svazku.
ntfsinfo        Odešle dotaz na informace o svazku NTFS.
statistics      Odešle dotaz na statistiku systému souborů.

C:\>fsutil fsinfo ntfsinfo c:
Sériové číslo svazku NTFS:       0xe2ecd978ecd94785
Verze:                          3.1
Počet sektorů:                   0x0000000001d4b138
Celkový počet clusterů                  0x00000000003a9627
Volné clustery:                  0x0000000000152892
Celkem vyhrazeno:                  0x0000000000000000
Počet bajtů na sektor:               512
Počet bajtů na cluster:               4096
Počet bajtů na segment záznamu souboru: 1024
Počet clusterů na segment záznamu souboru: 0
Platná délka dat hlavní tabulky souborů (MFT):
  0x0000000006e3c000
Počáteční číslo logického clusteru hlavní tabulky souborů (MFT):
  0x00000000000c0000
Počáteční číslo logického clusteru hlavní tabulky souborů (MFT2):
   0x00000000001d4b13
Začátek zóny hlavní tabulky souborů (MFT):
  0x000000000022d020
Konec zóny hlavní tabulky souborů (MFT):
  0x000000000024c780

C:\>
```

LITERATURA

[1] ČADA, O.: *Operační systémy*. Praha, Grada, 1993.

[2] PLÁŠIL, F.: *Operační systémy*. Praha, ČVUT, 1983.

[3] PLÁŠIL, F. - STAUDEK, J.: *Operační systémy*. Praha, SNTL, 1991.

[4] Horák, J.: *BIOS a Setup*. Brno, Computer Press, 2004.

[5] Kadlec, Z.: *Průvodce nitrem BIOSu*. Praha, Grada Publishing, 1996.

[6] Lee, J. - Ware, B.: *OpenSource – vývoj webových aplikací*. Brno, Computer Press, 2000.

[7] Lasser, J.: *Rozumíme Unixu*. Praha, Computer Press, 2002.

[8] Kolektiv autorů: *Linux Dokumentační projekt*. 3. aktualizované vydání. Brno, Computer Press, 2003. Soubory ke stažení na adrese http://knihy.cpress.cz/DataFiles/Book/00000675/Download/K0819.pdf

[9] Raymond, E. S.: *Umění programování v Unixu*. Brno, Computer Press, 2004.

[10] Lucas, M.: *FreeBSD*. Brno, Computer Press, 2003.

[11] Toxen, B.: *Bezpečnost v Linuxu*. Brno, Computer Press, 2003.

[12] Čada, O.: *Mac OS X Shell krok za krokem*. Praha, Grafika Publishing.

[13] Solomon, D. A.: *Windows NT pro administrátory a vývojáře*. Brno, Computer Press, 1999.

[14] Microsoft Corporation: *Microsoft Windows XP Professional Resource Kit*. Brno, Computer Press, 2004.

[15] Moskowitz, J.: *Zásady skupiny, profily a IntelliMirror ve Windows 2003, 2000 a XP*. Brno, Computer Press, 2006.

[16] Stanek, W. R.: *Příkazový řádek Microsoft Windows*. Brno, Computer Press, 2005.

[17] Price, B.: *Active Directory*. Brno, Computer Press, 2005.

[18] Allen, R. - Lowe-Norris, A. G.: *Active Directory*. Praha, Grada Publishing, 2005.

[19] Kokoreva, O.: *Registr Microsoft Windows XP*. Brno, Computer Press, 2002.

[20] Walnum, C.: *Programujeme grafiku v Microsoft Direct3D*. Brno, Computer Press, 2004.

[21] Kraval, I. - Ivachiv, P.: *Základy komponentní technologie COM*. Brno, Computer Press, 2001.

[22] Kačmář, D.: *Programujeme v COM a COM+*. Brno, Computer Press, 2000.

[23] Pokorný, J.: *Úvod do .NET Framework*. Brno, Computer Press, 2000. Soubory ke stažení na knihy.cpress.cz/DataFiles/Book/00000896/Download/frameworknet.zip

[24] Born, G.: *Skriptujeme operace na PC pomocí Microsoft Windows Script Host 2.0*. Brno, Computer Press, 2001.

[25] Aitken, P. G.: *Windows Script Host 2.0*. Praha, Grada Publishing, 2001.

[26] Walnum, C.: *VMWare*. Brno, Computer Press, 2004.

[27] Caletka, O.: *Partition Magic, Symantec Ghost a další utility pro práci s pevným diskem*. Brno, Computer Press, 2002.

[28] Kolektiv autorů: *The Linux Documentation Project*. Poslední aktualizace 2006. Dostupné z http://www.tldp.org

[29] Machej, P.: *GRUB – zavaděč systému*. Časopis Linux+ 2/05, str. 52-57

[30] Vondra, T.: *Gentoo Handbook, konfigurace zavaděče*. Dostupné z http://www.fuzzy.cz/gentoo/files/html/ch09.html

[31] Řehák, M.: *Operační systémy*. Dostupné z http://www.volny.cz/rayer/os/os.htm

[32] Mráz, O.: *Autoexec.bat a Config.sys*. Dostupné z http://www.volny.cz/otakarmraz/swPomoc/autocnfg.html

[33] Maturita.cz: *Konfigurace počítače*.
Dostupné z http://www.maturita.cz/prv/konfigurace_pocitace.htm

[34] Dvořák, V.: *WIN95 + RH6.2 = 10GB HDD*. Linuxové noviny, 2001.
Dostupné z http://www.linux.cz/noviny/2001-08/clanek05.html

[35] Park, E. B. - Galíček, R.: *Dual-Boot Linux a Windows 2000/XP s Grub HOWTO.*
Dostupné z
http://www.volny.cz/galicek/linux%20ver%20XP/grub-w2k-HOWTO-cz.html

REJSTŘÍK

Symbols
`cacls` 124

A
ACL 123
adresa absolutní 31, 38
adresa relativní 31
adresář 112, 131
adresář kořenový 112
algoritmus hodinový 44
algoritmus pekařův 83
API 9, 21, 29, 90
API grafické 151
aplikace distribuovaná 14
architektura hardwarová 144

B
Bakery algorithm 83
best fit 39
bestfit 126
BIOS 19, 136
blitter 40
boot loader 139
boot manažer 139, 144
boot sektor 118, 138, 139
bootblock 129
broadcast 66
brána 68
buffer neomezený 73, 74
buffer omezený 73, 74, 87
běh procesu 71

C
cluster 111, 118
cylindr 111

D
delta-list 60
deskriptor 47
DirectFB 156
DirectX 156
disk dynamický 137
disk logický 111, 137, 140
dispetcher 61
DOS Extender 20
dostaveníčko 89

E
EBR 139
emulace operačního systému 147
exekutiva 24

F
FAT tabulka 118
fdisk 140
first fit 39
flexibilita distribuovaného OS 15
fork 52
fragmentace 36, 39, 117

funkce operačního systému 9

G
GDT 47
GPT 137
granularita distribuovanosti 13
grid 13

H
HAL 24, 28
halda 36

I
i-uzel 129
I/O rozhraní 104
I/O zařízení 104
image 147
instrukce 7, 107
instrukce swap 84
instrukce TSL 84
instrukce XCHG 84
interpret příkazů 151
IPC 66
IRQ 107

J
job 8
jádro 20, 23, 24, 27

K
kernel mode 27
kernel-thread 59
klient 18
klient-server 12, 18
knihovna dynamicky linkovaná 21
komunikace asymetrická 67
komunikace asynchronní 67, 73
komunikace meziprocesová 66
komunikace nepřímá 88
komunikace symetrická 67, 75
komunikace synchronní . . 67, 74, 75, 80, 89, 90
kontext 54
konzistence dat 69, 76
konzument 73, 76, 87
krok úlohy 8
kvantum 61, 62, 64, 65

L
last fit 40
LDT 47

M
maskování přerušení 81
Master Boot Record 138
MBR 137, 138
mechanismus zpráv 88
MFT 125
mikrojádro 12
mikrokernel 18
minikernel 18
monitor 89
multicast 66
multiprocessing asymetrický 10
multiprocessing symetrický 10
multitasking 11, 53
multitasking kooperativní 55
multitasking preemptivní 56
multithreading 58
mutex 81
místo 70

O
obsluha oken 150
oddíl 137
oddíl aktivní 138
oddíl datový 140
oddíl primární 137, 138
oddíl rozšířený 137–139
oddíl systémový 140
odkaz pevný 125, 131
odkaz symbolický 131
offset 36, 38, 45, 47, 109
okno 151
okno kořenové 152

OpenGL 155
ovladač 106

P

page fault 42
pager 50
paměť operační 8
paměť virtuální 41
paměť vnitřní 8
paměť vnější 8
partition 137
PCB 51, 52
periferie 104
Petriho síť 70, 72, 74, 75, 77, 79, 80
PID 52, 53
platforma hardwarová 7
platforma softwarová 8
plocha pracovní 151
plánovač procesoru 61
plánování nepreemptivní 62, 63, 65
plánování preemptivní 62, 63, 65
podmínka 89
podmínky Bernsteinovy 69
podsystém prostředí 24
popisovač 47
port 68
POSIX 25
povrch 111
počítač abstraktní 18
počítač holý 8
počítač virtuální 17, 20, 147
PPID 53
priorita dynamická 65
priorita realtimová 12
priorita statická 65
proces 8, 51
proces systémový 26
procesor 7
procesor vícejádrový 7
producent 73, 76, 87
producent - konzument 107
program rezidentní 46
program zaváděcí 139, 142
prostor adresový 31
prostor procesu adresový 8
prostor procesu paměťový 8
prostor systému paměťový 8
prostředky fyzické 7
prostředky logické 8
prostředí desktopové 153
proud dat 124
průměrování exponenciální 64
pseudoLRU 43
pseudomultitasking 54
pseudoparalelismus 54
přechod 70
přerušení 107
přerušení hardwarové 107
přerušení maskované 81, 108
příkazy vnitřní 20
příkazy vnější 20

R

registr 23
registr příznaků 54
registr segmentový 36, 54
registr zásobníkový 54
rozhraní dokumentované 26
rozhraní systémových volání 29
rozšiřitelnost distribuovaného OS 16
RPC 67, 80, 90
rámec 41

S

schránka 68
segment 36, 44, 109
sekce kritická 72, 76
sektor 110
selektor 47
semafor 84
semafor binární 85
semafor obecný 86
server 18
shell 28

služba 26
socket 68
soubor 112, 115
soubor odkládací 48
soubor řídký 125
souborový systém žurnálovací 123
správa oken a grafiky 26
správce IFS 22
správce konfigurace 22
správce oken 153
správce periferií 105
správce programů 21
správce virtuálních zařízení 20, 22
stav konzistentní 69
stav procesu 71
stopa 110
stream 124
stroj virtuální 17
struktura abstraktní počítač 18
struktura hierarchická 17
struktura klient-server 18, 27
struktura modulární 26
struktura monolitická 17
struktura stavebnicová 18
struktura virtuální počítač 17
struktura vrstvená 17, 26
stránka 37, 41
stránkování 37
střídání procesů 82
subsystém grafický 150
subsystém prostředí 24
superblok 129
svazek 125, 137
svazek dyniamický 137
synchronizace procesů 72, 80
systém distribuovaný 13
systém I/O 8
systém jednoprocesorový 10, 19
systém jednoprogramový 11, 19
systém jednouživatelský 10, 19
systém jednoúlohový 11
systém lokální 11, 19
systém multitaskový 11, 24, 27
systém operační 8
systém operační distribuovaný 15
systém realtimový 11, 76
systém souborový . . 22, 28, 115, 117, 118, 121, 123, 128, 129
systém souborový virtuální 117, 133
systém souborový žurnálovací . 116, 131, 132
systém speciální 11
systém síťový 11, 24, 27
systém univerzální 11, 19, 24, 27
systém V/V 8
systém víceprocesorový . . . 10, 24, 27, 88
systém víceprogramový 11
systém víceuživatelský 10, 24, 27
systém víceúlohový 11
systém výpočetní 7

T

tabulka deskriptorů 47
tabulka rozšíření disku 138
tabulka stránek 41
transparentnost distribuovaného OS . 15
TSR 46, 55

U

unicast 66
used bit 43
user-thread 59
uživatel 8

V

V/V zařízení 104
vektor přerušení 47, 109
VFAT 121
VFS 29, 128
videopaměť 31
virtualizace zařízení 105
vlákno 58, 75
volání systémové 88

výpadek stránky 42

W
widget knihovna 153

X
X Window System 28, 152

Z
zakázka 7
zasílání zpráv 66, 67, 73, 75, 88
zavaděč 139, 142
zařízení blokové 106
zařízení periferní 8
zařízení sdílené 105
zařízení speciální 106
zařízení společné 105
zařízení vyhrazené 105
zařízení znakové 106
zákaz přerušení 81

Č
čekání aktivní 80, 81
čekání pasivní 80, 81
čítač programový 52, 54

Ú
úloha 8
úloha Kritická sekce 72
úloha Model - obraz 75
úloha Producent - konzument 73, 75, 87, 89
úloha Pět hladových filozofů 78, 87
úloha Souběh procesů 79
úloha synchronizační 72
úloha Čtenáři - písaři 77
úložiště 110