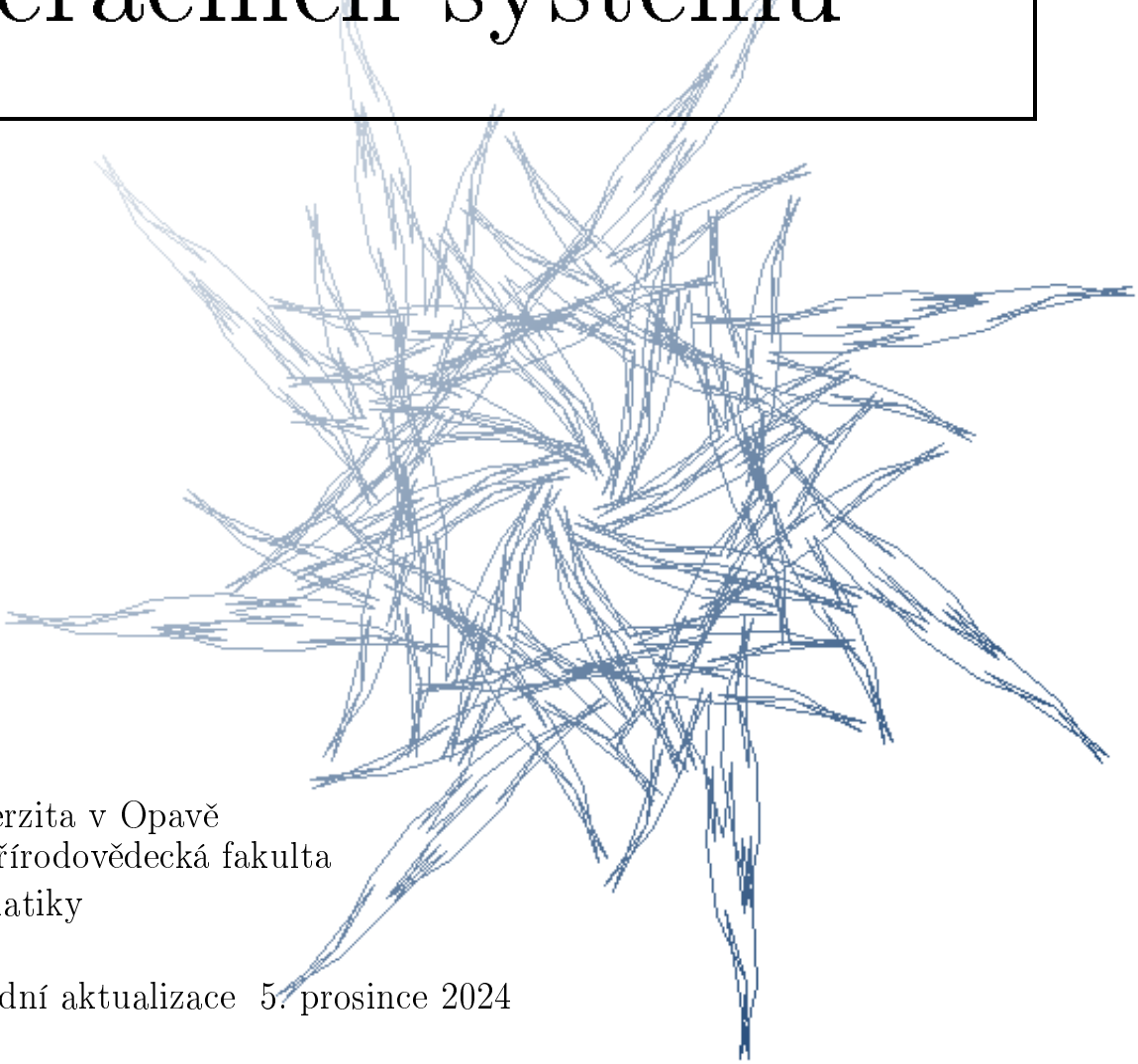
SLEZSKÁ UNIVERZITA
FILOZOFICKO-PŘÍRODOVĚDECKÁ FAKULTA V OPAVĚ

Šárka Vavrečková

Architektura operačních systémů



Slezská univerzita v Opavě
Filozoficko-přírodovědecká fakulta
Ústav informatiky

Opava, poslední aktualizace 5. prosince 2024

Anotace: Tento dokument je určen pro studenty druhého ročníku IVT na Ústavu informatiky Slezské univerzity v Opavě. Obsahuje látku probíranou na přednáškách předmětu *Architektura operačních systémů*. Zabýváme se zejména strukturou operačních systémů, správou paměti, procesů a zařízení – jak obecně, tak i konkrétně v operačních systémech Windows a Linux.

Architektura operačních systémů

RNDr. Šárka Vavrečková, Ph.D.

Dostupné na: http://vavreckova.zam.slu.cz/archos.html

Ústav informatiky
Filozoficko-přírodovědecká fakulta v Opavě
Slezská univerzita v Opavě
Bezručovo nám. 13, Opava

Sázeno v systému LaTeX

Předmluva

Co najdeme v těchto skriptech

 *Rychlý náhled:* Tato skripta jsou určena pro studenty informatických oborů na Ústavu informatiky Slezské univerzity v Opavě. Na přednáškách předmětu Architektura operačních systémů probíráme především teoretické koncepty související se strukturou operačních systémů, rolí jednotlivých částí jádra a mechanismy správy procesů, paměti a zařízení, ovšem každé téma je následně vztaženo na konkrétní operační systémy (obvykle Windows a Linux).

Některé oblasti jsou také „navíc“ (jsou označeny ikonami fialové barvy), ty nejsou probírány a ani se neobjeví na zkoušce – jejich úkolem je motivovat k dalšímu samostatnému studiu nebo pomáhat v budoucnu při získávání dalších informací dle potřeby v zaměstnání.

Značení

Ve skriptech se používají následující barevné ikony:

-  *Rychlý náhled* (skript, kapitoly), ve kterém se dozvíme, o čem to bude.
-  *Klíčová slova* kapitoly.
-  *Cíle studia* pro kapitolu nám řeknou, co nového se v dané kapitole naučíme.
-  Nové *pojmy*, značení apod. jsou označeny modrým symbolem, který vidíme zde vlevo. Tuto ikonu (stejně jako následující) najdeme na začátku odstavce, ve kterém je nový pojem zaváděn.
-  Konkrétní *postupy a nástroje* (příkazy, programy, soubory, skripty), způsoby řešení různých situací, atd. jsou značeny také modrou ikonou.
-  Některé části textu jsou označeny fialovou ikonou, což znamená, že jde o *nepovinné úseky*, které nejsou probírány (většinou; studenti si je mohou podle zájmu vyžádat nebo sami prostudovat). Jejich účelem je dobrovolné rozšíření znalostí studentů o pokročilá témata, na která obvykle při výuce nezbývá moc času.
-  Žlutou ikonou jsou označeny odkazy, na kterých lze získat *další informace* o tématu. Nejčastěji u této ikony najdeme webové odkazy na stránky, kde se dané tématice jejich autoři věnují podrobněji.

-  Červená je ikona pro *upozornění* a poznámky.
-  Toto značení znamená, že si u zkoušky můžete vybrat mezi několika alternativami. Typicky jde o rozhodnutí, zda daný postup, pojem, strukturu apod. vysvětlíte na Windows nebo na Linuxu.

Pokud je množství textu patřícího k určité ikoně větší, je celý blok ohraničen prostředím s ikonami na začátku i konci, například pro definování nového pojmu:

 **Definice**

V takovém prostředí definujeme pojem či vysvětlujeme sice relativně známý, ale komplexní pojem s více významy či vlastnostmi.

Podobně může vypadat prostředí pro delší postup nebo delší poznámku či více odkazů na další informace. Mohou být použita také jiná prostředí:

 **Příklad**

Takto vypadá prostředí s příkladem, obvykle nějakého postupu. Příklady jsou obvykle komentovány, aby byl jasný postup jejich řešení.

 **Úkol**

Otázky a úkoly, náměty na vyzkoušení, které se doporučuje při procvičování učiva provádět, jsou uzavřeny v tomto prostředí. Pokud je v prostředí více úkolů, jsou číslovány.

Pro úseky kódu, příkazy a jejich výstupy je používáno neproporcionální písmo.

Jak probíhá zkouška

Zkouška je písemná, může být ve formě online testu s ústní konzultací (ať máte zpětnou vazbu). Na stránkách předmětu je k dispozici orientační seznam otázek, které se mohou objevit na písemce, jsou přípustné i drobné modifikace. Tato skripta plně pokrývají odpovědi na všechny otázky ze seznamu a jejich modifikace.

Obsah

Kapitola 1

Úvod do operačních systémů

Rychlý náhled: Tato kapitola je úvodem do problematiky struktury operačních systémů. Seznámíme se zde se základními pojmy, definicí operačního systému, funkcemi a typy 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 (tj. také firmware).

Klíčová slova: operační systém, zdroje, instrukce, zakázka, úloha, proces, platforma, uživatelské a programové rozhraní, jedno/víceprocesorový systém, jedno/víceuživatelský systém, (a)symetrický multiprocessing, NUMA, jedno/víceprogramový systém, multitasking, realtimový operační systém, distribuovaný operační systém, distribuovaná aplikace

Cíle studia: Po prostudování této kapitoly porozumíte základním pojmům souvisejícím s operačními systémy a kategorizaci operačních systémů, seznámíte se také se speciálními typy operačních systémů – realtimovými a distribuovanými systémy.

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 (Contract) – pokyn, který má výpočetní systém provést.

Fyzické prostředky výpočetního systému jsou procesor, paměti, úložiště a další. Fyzické prostředky (především procesor) určují hardwarovou platformu, pro kterou je daný operační systém naprogramován, například Intel x86, amd64, PowerPC (ppc), MIPS, ARM, sparc64, Raspberry Pi, atd.

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.

Adresový prostor procesu je paměťový prostor ve vnitřní paměti, který je vyhrazen tomuto procesu a je na něm zavedena metrika (adresy, každý byte je očíslován).

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

Definice

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.

Zatímco u operačních systémů rozlišujeme různé varianty běžící na odlišných hardwarových platformách, u aplikací zvažujeme softwarovou platformu, pro kterou je programován. Ovšem pokud operační systém přechází na jinou hardwarovou platformu (jak se to stalo například u MacOS – přechod od Intel/amd64 na ARM), ovlivní to i aplikace, ty také musejí být přeprogramovány, protože různé hardwarové platformy mají různé instrukční sady, jejichž instrukce se vyskytují ve spustitelných souborech přeložených aplikací.

V případě *multiplatformního softwaru* nás však softwarová platforma nemusí moc zajímat. Obvykle jde o aplikace psané ve skriptovacích jazycích jako je Python, Lua, Perl apod. Do určité míry se to týká také aplikací programovaných v hybridních jazycích (Java, C#). Na cílovém zařízení ovšem musí být nainstalováno příslušné programové prostředí (překladač Pythonu či jiného skriptovacího jazyka včetně požadovaných knihoven, Java Runtime Environment pro Javu, pro C# .NET Framework nebo MONO).

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, operační systémy mohou zajišťovat i mnoho dalších jiných funkcí. Nejdůležitějším funkcím jsou vyhrazeny samostatné kapitoly.

Správa paměti představuje vedení evidence operační 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, patří sem správa ovladačů, 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 činnosti.
- *Uživatelské rozhraní operačního systému* (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 operačního systému* je rozhraní mezi 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

1.3.1 Základní rozdělení

Dále uvedeme dělení operačních systémů podle různých kritérií.

- **Podle počtu ovládaných procesorů** dělíme operační systémy na
  - *jednoprocesorové* (monoprocesorové) – Windows s DOS jádrem (verze 9x, ME),
  - *víceprocesorové* (multiprocesorové) – UNIXové systémy včetně Linuxu, Windows s NT jádrem (NT, 2000, XP, Vista a novější), dokážou rozplánovat alespoň některé úlohy tak, aby mohly být zpracovávány na více procesorech zároveň. UNIXové systémy obecně mohou běžet na clusterech s velkým množstvím procesorů, u Windows záleží na konkrétní edici a licenci (i běžné desktopové edice podporují dva procesory).

- Víceprocesorové dělíme na dvě podkategorie:
  - 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.

Prakticky všechny moderní systémy používají symetrický multiprocessing.

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 grafická API jako je OpenGL, DirectX apod.

Dá se vícejádrový počítač počítač brát jako víceprocesorový? Ne tak úplně. Jádra v rámci jednoho procesoru totiž některé prostředky sdílejí (záleží na konkrétní architektuře).

U víceprocesorových systémů (především serverů) se můžeme setkat s pojmem *NUMA* (Non-Uniform Memory Access). Paměť je rozdělena na samostatné části, uzly, a ke každému takovému uzlu je sběrnicí připojen jeden nebo více procesorů (každý uzel má svou paměťovou sběrnici). Procesor dokáže k paměti ve „svém“ uzlu přistupovat velmi rychle, k paměti v jiných uzlech má sice také přístup povolen, ale je pomalejší. Proto by procesor měl využívat především paměť lokální, ve svém uzlu.

Účelem architektury NUMA je co nejvíc zefektivnit a škálovat komunikaci procesorů s pamětí, protože u víceprocesorových systémů bez NUMA je paměťová sběrnice úzkým hrdlem, které zpomaluje celý systém. Navíc pro rozsah adres je limit daný počtem bitů určených pro uložení adresy, a NUMA tento limit umožňuje obejít (každý uzel má vlastní adresaci).

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 buď přímo na zařízení nebo vzdáleně přes terminály či jinak po síti. Tyto systémy především musí zajistit přísné oddělení prostředků (např. paměti) využívaných různými uživateli, aby jeden uživatel neměl přístup k datům a nastavení jiného uživatele.

Podle počtu spuštěných programů (podrobněji viz kap. 4.2) rozlišujeme operační systémy

- *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 míry specializace existují operační systémy

- *speciální* – jsou specializované na jeden typ (nebo několik málo typů) úloh, například v síťových zařízeních (pokud se nejedná o Linux), některých embedded zařízeních apod.,
- *univerzální* – běžné operační systémy na PC, řeší různé typy úloh.

Rozlišujeme také podskupiny operačních systémů – realtimové a distribuované.

1.3.2 Realtimové operační systémy

Realtimové operační systémy jsou operační systémy pracující téměř 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, pouze je pro každý typ realtimového požadavku určena „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 na rozdíl 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 kapitolu 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 (vyslovuj [kju:nix]) je realtimový systém postavený na hodně upraveném UNIXovém klonu, používá se především v automobilech a různých embedded zařízeních. 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 komerční systém, po určitou dobu byla dostupná volná verze, ale v současné době se jedná o plně uzavřený systém (pod taktovkou společnosti BlackBerry). 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, byl původně určen hlavně pro průmysl (řízení robotů ve výrobě apod.). Původně to byl komerční projekt, ale komerční podpora už není poskytována, doporučuje se přejít k RT-Preempt Patch (viz níže).

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.

Realtime Preemption patch (RT-Preempt Patch) je speciální aktualizace (patch) pro linuxové

[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).

jádro, která toto jádro upraví tak, aby pracovalo realtimově. Jedná se hlavně o úpravu synchronizačních mechanismů jádra (o synchronizaci je v těchto skriptech samostatná kapitola), časovačů a obsluhy přerušení. Jednou z oblastí využití je také průmysl.

RTX (RealTime eXtension) je modul, který rozšiřuje možnosti Windows s NT jádrem (NT/2000/XP/7, jen 32bitové) směrem k realtimovým systémům. Nejde tedy o realtimový systém, pouze o nástavbu pro operační systém klasického typu (vpodstatě takový patch jako u RT-Preempt).

K systému je přidáno zvláštní rozšíření vrstvy HAL[2] (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).

Realtime systémy pro Internet věcí. IoT zařízení vyžadují trochu jiný způsob zacházení. Některá tato zařízení potřebují realtimový systém, a to s určitými specifiky (co nejmenší, co nejjednodušší, s nízkou spotřebou, některé funkce tam vůbec nemusejí být, jiné naopak ano). Pro tyto účely dnes existují speciální realtimové systémy, například:

- Contiki-NG,
- TizenRT,
- RT Preempt Patch,
- FreeRTOS,...

Další informace:

- https://blackberry.qnx.com/en
- http://www.tldp.org/HOWTO/RTLinux-HOWTO.html
- https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO
- https://www.contiki-ng.org/
- https://docs.tizen.org/application/tizen-studio/rt-ide/overview/
- https://www.freertos.org/
- https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

1.3.3 Distribuované operační systémy

Distribuovaný systém[3] (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 (obvykle síťovými protokoly nebo přes vzdálené volání procedur – RPC, je také možné využívat k tomuto účelu vytvořená objektová rozhraní – DCOM, CORBA apod.),
- každá taková část je (může být) zpracovávána na jiném procesoru se zajištěním co největší transparentnosti.

[2] O vrstvě HAL a jejích funkcích se více dovíme v kapitole 2.3.1 na straně 24.
[3] Můžeme také najít název *grid*.

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. S distribuovanými aplikacemi se setkáváme například v těchto případech:

- *distribuce dat* – databáze, systémy pro správu obsahu, informační systémy, atd. – je nutné sdílet data v mnoha pobočkách po celém světě,
- *distribuce výpočtů* – složité výpočty jsou distribuovány na více počítačů,
- *distribuce prostředků* – prostředky vlastněné jedním subjektem (například výpočetní výkon procesorů, paměťová úložiště, atd.) mohou být distribuována (nabízena) jiným subjektům s tím, že umístění a konkrétní způsob přístupu k nim jsou těmto subjektům skryty.

Typické a velmi běžné využití distribuovanosti je v databázích a (často na ně navazujících) informačních systémech. Velké databáze a informační systémy bývají distribuovány na mnoha uzlech i po celém světě, třebaže ne vždy jde opravdu o distribuovanou aplikaci splňující požadavek transparentnosti a flexibility (bude probíráno dále).

Jednou z nejznámějších distribuovaných aplikací je *BOINC* (zkratka pro Berkeley Open Infrastructure for Network Computing[4]) 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), MalariaControl.net (sledování a predikce šíření malárie), 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

[4] Informace o BOINC najdeme na http://boinc.berkeley.edu.

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.

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.

✎ **Existující distribuované operační systémy.** Svého času byl velmi oblíbeným distribuovaným OS systém LOCUS, který je kompatibilní s UNIXem. Dnes už pomalu ustupuje.

Distribuovaný operační systém se dá očekávat například v datových centrech, kdy potřebujeme, aby se celý cluster serverů choval jako jeden celek, aby působil jako jeden systém. V takovém případě

se typicky používá některá přizpůsobená distribuce Linuxu. Přímo pro tyto účely jsou přizpůsobeny například Red Hat Enterprise Cluster a SUSE Linux Enterprise with High Availability Extension.

1.4 Cloud Computing a operační systémy

V současné době se objevují možnosti provozování služeb, aplikací a datových úložišť (nebo jiných prostředků) na Internetu. Souhrnně se tento koncept nazývá Cloud Computing. Název je odvozen od obvyklého zobrazování principu konceptu, kdy poskytované prostředky nejrůznějšího druhu jsou umístěny „v oblaku“, jehož fyzické umístění ani vnitřní organizace nejsou zvenčí zřejmé. Pro přístup k takto nabízeným službám často stačí jen internetový prohlížeč a případně přídavný modul či dodatečný software nebo technologii (například AJAX, Javu, Flash Player nebo Silverlight).

V souvislosti s Cloud Computingem se setkáváme i s velkými firmami – Google, Microsoft, Amazon, Dell, atd. Také se tento trend osvědčuje v oblasti zabezpečení, některé firmy produkující bezpečnostní software nabízení své aplikace ve formě „Cloud“, tedy aplikace (včetně nejnovějších aktualizací) běží v cloudu, skenuje uživatelův počítač přes síť a téměř nezatěžuje jeho procesor (ovšem zatěžuje síťové připojení).

Z hlediska operačních systémů nás mohou zajímat operační systémy, které běží „v cloudu“, v oblaku – *cloud operační systémy*. Cloud operační systém se liší od běžného operačního systému především v tom, že kód jeho jádra (a také obvykle kód téměř všeho ostatního, co systém nabízí) běží na procesoru někde v cloudu, náš procesor není zatěžován, a s počítačem, u kterého sedíme, komunikuje obvykle přes síťové protokoly (náš počítač vlastně funguje jako terminál, vstupně/výstupní rozhraní).

Takový operační systém lze buď provozovat v internetovém prohlížeči podporujícím příslušnou technologii, pak se vlastně nejedná ani tak o operační systém, ale spíše o něco mezi webovou aplikací a operačním systémem. Jinou možností je provozování takového systému nad EFI (to je modernější náhrada BIOSu), kdy vlastně na počítači ani nemusíme operační systém instalovat (součástí EFI může být jednoduchý internetový prohlížeč). Některá řešení nabízení možnost provozu na „tenkých klientech“[5].

Mezi cloud operační systémy patří například

- *Chromium OS* (http://www.chromium.org/chromium-os) – produkt firmy Google, který umožňuje menším zařízením přistupujícím na Internet pracovat i bez nutnosti instalace, správy a také zdlouhavého načítání systému při startu zařízení, je možné využívat některé internetové aplikace od Googlu (Google Dokumenty, Picasa apod.), verze rozhodně ještě není finální.
- *Windows Azure* (http://www.windowsazure.com) – pokus Microsoftu o cloud operační systém, který zde slouží jako základ, na kterém klienti mohou vyvíjet a nabízet své aplikace. Jde o systém odvozený od Windows Server (používá se technologie serverové virtualizace), a sám o sobě je určen především programátorům aplikací.
- *ICloud OS* (http://www.icloud.com) – přes webový prohlížeč pracujeme v systému založeném na Ubuntu Linuxu, jehož grafické rozhraní je upraveno do vizáže Windows, máme k dispozici běžné aplikace a úložný prostor.

[5] Tenký klient je vlastně obdoba terminálu – počítač, na kterém není instalován operační systém, pracujeme vzdáleně, v operačním systému nainstalovaném jinde, obvykle na lokálním serveru. Tenké klienty jsou dobrým řešením pro skupiny počítačů na kvalitní síťové infrastruktuře s rychlým serverem, tenký klient obvykle ani nebývá vybaven pevným diskem (nejdůležitější je síťová karta umožňující vzdálené bootování – načítání systému).

- *SilveOS* (http://www.silveos.com) – vyžaduje technologii Silverlight a v internetovém prohlížeči běží v tzv. sandboxu (což znamená vyšší úroveň zabezpečení, ale také odříznutí od systému reálně běžícího na počítači). Je založen na Windows.
- *StartForce* (http://www.startforce.com) – systém, který svým vzhledem připomíná Windows, ale zřejmě jde o vlastní systém, máme k dispozici základní aplikace a úložný prostor. Využívá technologii AJAX.

To je jen výčet nejznámějších řešení, cloud operačních systémů je více. Odlišují se kromě jiného svou licenční politikou (některé jsou zdarma, většina poskytuje alespoň bezplatnou demonstrační verzi), mají různé jádro, poskytované prostředky (včetně úložného prostoru) a aplikace, některé umožňují množství aplikací rozšiřovat, jiné nikoliv, mohou být univerzální nebo určené k specifickému účelu.

Žádný z výše uvedených systémů, které jsou koncipovány jako univerzální, zatím nemůže nabídnout tytéž možnosti jako lokálně nainstalovaný systém nebo systém běžící na (jednom) serveru v lokální síti (přístupný na tenkých klientech). Ovšem výhodou je rychlost „bootování“ – zprovoznění systému po zapnutí zařízení a možnost sdílení prostředků včetně souborů v rámci téhož cloud systému, tedy tento typ systémů by mohl být určen například pro PDA, netbooky, smartphony a podobná zařízení s vhodně vybaveným firmwarem (třeba EFI). Ve většině případů by mohl být kámen úrazu v bezpečnosti systému.

Firmy, zejména střední velikosti, na cloudu obecně láká především dostupnost prakticky kdekoliv (kde je k dispozici síťová konektivita), dále to, že není nutné provozovat a financovat vlastní IT oddělení (servis obvykle zajišťuje provozovatel cloudu), a také jednoduchá možnost navyšování kapacity a rozšiřování služeb s cloudem souvisejících (připlatí si).

Kapitola 2

Struktura operačních systémů

Rychlý náhled: 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.

Klíčová slova: monolitická struktura, modulární struktura, vrstvená architektura, virtuální počítače, abstraktní počítače, klient-server, mikrokernel, mikrojádro, hybridní jádro, privilegovaný a uživatelský režim, HAL, USER, GDI, KERNEL, IFSM, exekutiva, IPC, VFS, udev, ring.

Cíle studia: Po prostudování této kapitoly porozumíte tomu, jaká je vnitřní struktura operačních systémů, a budete se rámcově orientovat v architektuře operačních systémů Windows a Linux včetně vazby na zabezpečení jádra rozdělením na privilegovaný a uživatelský režim.

2.1 Základní typy architektur

Následující pojmy (typy struktur) platí nejen pro výpočetní systémy jako celek, jsou obecně používány například také pro strukturu jádra systému nebo vrstev v jádře.

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

Jádra moderních operačních systémů jsou ponejvíce monolitická (týká se to prakticky všech UNIXových systémů a dále Windows rodiny NT). Znamená to, že při startu systému se jádro načítá z jediného souboru, funkcionalita jádra je rozšiřována moduly (knihovnami nebo za běhu linkovanými moduly). Ve Windows se jádro načítá ze souboru `ntoskrnl.exe`, v Linuxu jde o soubor `vmlinuz...` (v názvu souboru následuje označení verze).

✎ *Modulární struktura* – systém je členěn do modulů, které lze podle potřeby přidávat (nejlépe za běhu systému). U tohoto typu struktury se předpokládá unifikované rozhraní modulů, přes které může systém komunikovat i s takovým modulem, který v době vzniku systému ještě neexistoval.

Moduly, jak bylo dříve poznamenáno, se používají ve stále větším rozsahu v jádrech moderních operačních systémů. Jako moduly jsou implementovány ovladače (tedy ty, které se načítají do jádra), různé filtry (procesy pro zpracování dat), síťové protokoly (v UNIXových systémech a v novějších Windows od verze Vista), atd.

✎ *Vrstvená (hierarchická) struktura* – části systému jsou uspořádány do vrstev, každá vrstva využívá služeb nižších vrstev, ne naopak. Každá vrstva komunikuje právě jen s okolními vrstvami. Systém je budován od vnitřních vrstev k vnějším, proto vnitřní vrstvy, které jsou obvykle nejdůležitější z hlediska stability a bezpečnosti, bývají nejlépe otestovány. V informatice se s touto architekturou setkáváme také u počítačových sítí.

Tento typ struktury je u moderních operačních systémů nejběžnější při reprezentaci systému jako celku. Rozlišujeme především dvě hlavní vrstvy – vrstvu jádra (chráněný režim) a vrstvu uživatelskou.

✎ *Virtuální počítače* (virtuální stroje) – v systému existují samostatné moduly (virtuální počítače, virtuální zařízení), každý z nich je zhruba stejně vybaven prostředky (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 (v těchto podsystémech mohou například běžet starší aplikace, které by jinak nemohly na novějším systému fungovat). Tento princip na vyšší úrovni využíváme také v softwarové a hardwarové virtualizaci, které se budeme věnovat ke konci semestru.

✎ *Abstraktní počítače* – v systému také existují části, ale na rozdíl 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). Zatímco virtuální počítače mají „od každého prostředku něco“ (část paměti, část času procesoru apod.), abstraktní počítač má přidělen jeden prostředek, ale zato výhradně (neexistuje jiný vlastník tohoto prostředku).

Typické použití je v primárním rozhraní zařízení – ovladače. Například tiskárna je přidělena pouze jedinému procesu – obslužnému procesu tiskárny (ten může být totožný s ovladačem). Obslužný proces vede tiskovou frontu tiskárny a v ní eviduje požadavky procesů na tisk.

✎ *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 správa paměti, přepínání mezi procesy či 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.

✎ *Architektura mikrokernel* (mikrojádro, také multiserver) znamená použití speciálního typu jádra, ve kterém jsou přítomny pouze nejdůležitější části (správa procesů, IPC, správa paměti, plánování procesoru). Všechno ostatní je implementováno jako systémové procesy běžící v uživatelském režimu, tedy mimo jádro.

Hlavní výhodou mikrojádra je vyšší stabilita, odolnost proti chybám a odolnost proti útokům (čím méně toho běží v jádře, tím méně je míst k poškození a útoku). Nevýhodou je horší výkon, protože se častěji musí přepínat mezi režimem jádra a uživatelským režimem.

Mikrojádro se používá především v realtimových systémech, například QNX, dále PikeOS pro embedded systémy, ze starších také Symbian OS. Mikrojárem je také Mach, což je původní jádro pro projekt GNU (později byl nahrazen Linuxem).

Hybridní jádra mají vlastnosti někde mezi mikrojádry a monolitickými jádry. Zatímco GNU Mach je mikrojádro, Darwin (jeho odvozenina používaná v Apple MacOS) je hybridní jádro.

2.2 Systémy MS-DOS a Windows

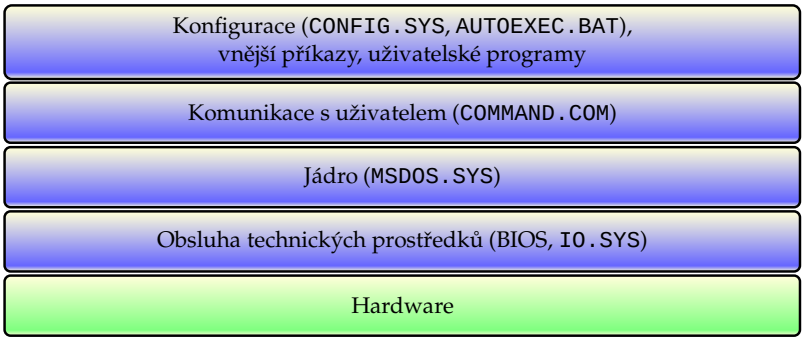
Řada Windows s DOS jádrem zahrnuje Windows 95, 95 OSR2, 98, 98 SE a ME. Tyto verze vznikly úpravou a včleněním původně samostatného operačního systému MS-DOS jako jádra do Windows, které se tímto staly samostatným operačním systémem[1].

Windows s NT jádrem (NT 3.x, NT 4.x, 2000, XP, Server 2003, Vista, Server 2008, 7, Server 2008 RC2, atd.) mají zcela přepracované jádro a přes značnou zpětnou kompatibilitu se vlastně jedná o jiný typ operačního systému.

2.2.1 MS-DOS a Windows do verze 3.x

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.



Obrázek 2.1: Struktura MS-DOSu 6.22

BIOS poskytuje programátorům základní ovládání hardwaru (např. klávesnice, monitoru) přes *hardwarová a softwarová přerušení*. 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

[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.

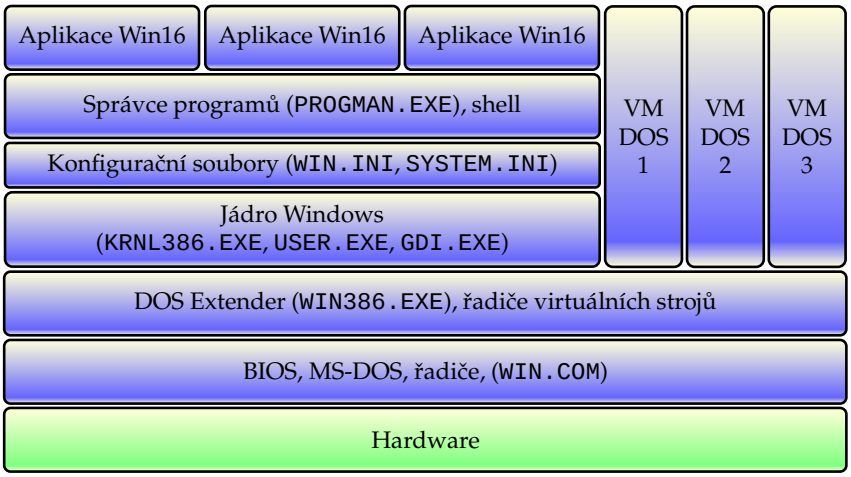
funkci ve chvíli, kdy je přerušení vyvoláno jinde než v programu (například takto hlídá stisknutí kláves nebo pohyb myši).

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 vrstvami, 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 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[2], 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, 16bitové knihovny a ve verzi 3.11 for Workgroups základní podporu sítě (pouze síť peer-to-peer).



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

Řadiče (ovladače, drivery) ovládají periferní zařízení pro Windows; řadiče přímo pro Windows jsou spouštěny v souboru `SYSTEM.INI` pomocí příkazu `DEVICE`.

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

[2]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.

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, ...),
- `GDI.EXE` – rozhraní grafického zařízení (obsahuje funkce pro kreslení úseček, vytváření kurzoru, ikony, písmo, ...), cokoliv souvisí se základními funkcemi pro grafický výstup (na obrazovku, tiskárnu apod.),
- `USER.EXE` – uživatelské rozhraní, zdroje, které nepatří do GDI (dialogová okna, menu, okna, tlačítka, ...).

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.

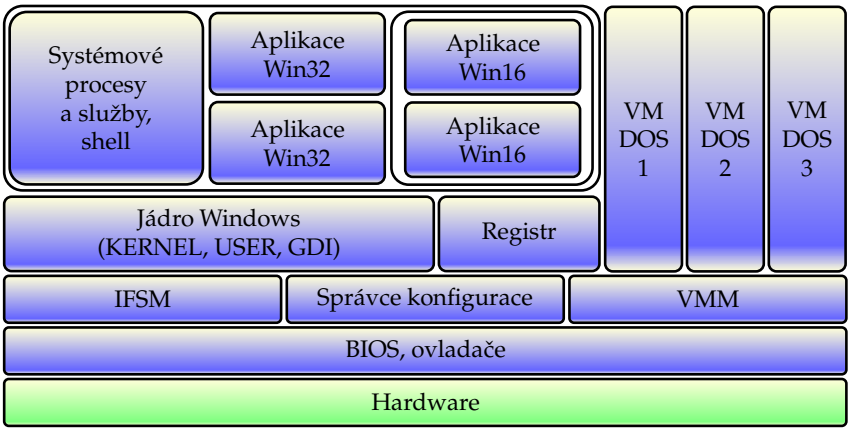
Všechny dosud uvedené vrstvy platí zejména pro aplikace psané pro Windows (16bitové aplikace pro Windows do verze 3.x), DOS programy nevědí o existenci jádra Windows a INI souborů, proto horní vrstvy nevyužívají. Protože však je samotný MS-DOS jednoprogramový systém (kromě ovladačů a rezidentních programů je spuštěn vždy jen jeden program), tyto programy jsou napsány bez jakýchkoliv ohledů na možnost sdílení prostředků 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.

2.2.2 Windows s DOS jádrem

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 32bitový systém (ale některé knihovny zůstávají 16bitové), ostatní vlastnosti zůstávají. Na obrázku 2.3 je naznačena zjednodušená struktura tohoto systému.

Spodní vrstva (BIOS a ovladače) 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 počítačů (viz stranu 12),
- *IFSM* je správce instalovatelných souborových systémů (Installable File Systems Manager), spravuje různé typy souborových systémů, např. FAT16, VFAT (FAT32 s rozšířeními), CDFS (pro



Obrázek 2.3: Struktura Windows 9x/ME

CD-ROM), UDF (pro DVD), atd., přes tuto komponentu se komunikuje s paměťovými zařízeními (vše, co odpovídá standardu *mass storage* s takovým souborovým systémem, kterému IFSM rozumí),

- *Správce konfigurace* spravuje ovladače hardware na vyšší úrovni, včetně funkce Plug&Play.

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

- *KERNEL* – multithreading, multitasking, správa paměti, synchronizace objektů, vstupu a výstupu u souborů, atd.,
- *GDI* (Graphics Device Interface) – rozhraní grafických zařízení (obrazovka, tiskárna, plotter, atd.), zde najdeme základní funkce pro výstup na obrazovku, tiskárnu apod., také správce tisku, spooler, zpracování grafiky, základní grafické objekty, ...,
- *USER* – také jako u předchozího jde o uživatelské rozhraní (pozor, nejen grafické), tedy vstupy z klávesnice, myši apod. (řízené přerušeními), výstupy do uživatelského grafického rozhraní (okna, menu, ikony, ...), práce s časovačem, atd.

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

Jak běží procesy:

- *Win32 aplikace* (psané pro Windows od verze 95 výše) běží všechny ve společném virtuálním stroji, ale každá má svůj vlastní paměťový prostor (pod toutéž číselnou adresou každá z těchto aplikací vidí různá fyzická umístění v paměti),
- *Win16 aplikace* (pro Windows 3.11 a nižší) běží všechny ve společném virtuálním stroji zároveň s Win32 aplikacemi, ale na rozdíl od Win32 aplikací sdílejí jeden společný paměťový prostor (společný pro Win16 aplikace; pod toutéž číselnou adresou vidí všechny Win16 aplikace tentýž objekt v paměti),[3]
- *DOS aplikace* mají každá svůj virtuální stroj, v něm svůj vlastní paměťový prostor.

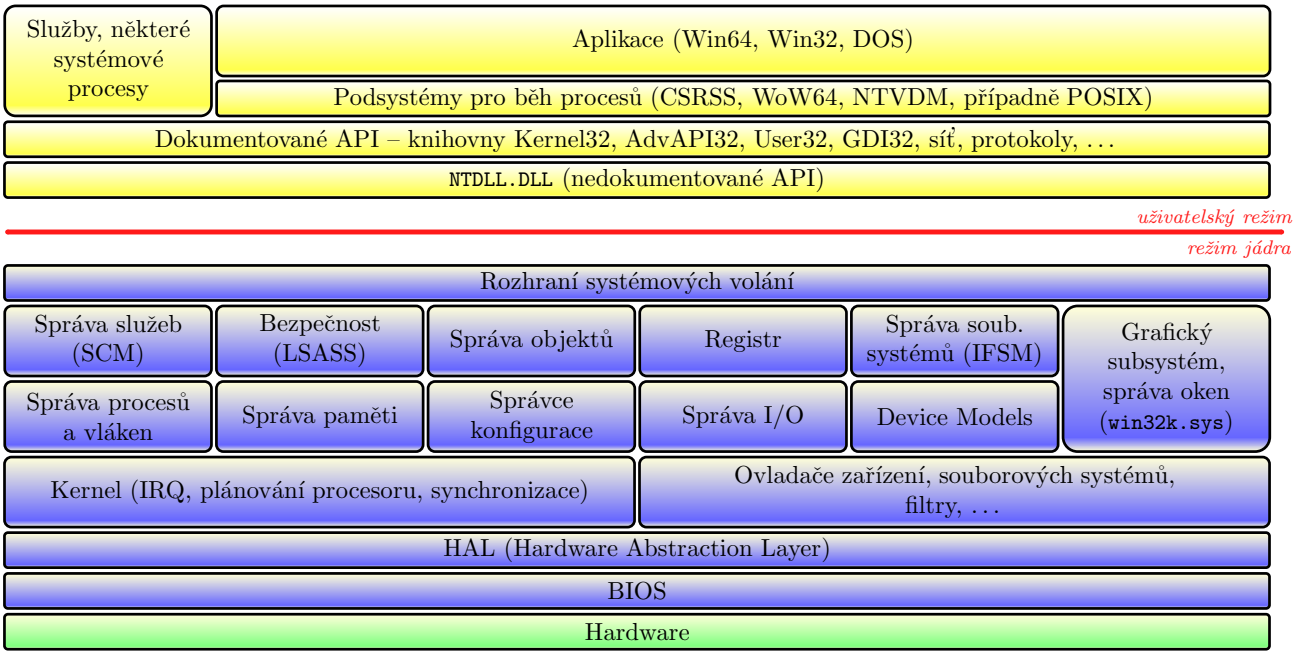
[3] To je kromě jiného proto, že Win16 aplikace byly psány pro systém, kde procesy mohou spolu komunikovat přes společnou paměť v DLL knihovnách, u Win32 aplikací a 32bitových knihoven to již není možné, byl upřednostněn model více vyhovující požadavkům stability a bezpečnosti systému.

2.2.3 Windows řady NT do verze XP

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. Na celém konceptu je vidět inspirace UNIXovými systémy, nejen co se týče vrstvy HAL.

Systém byl navržen jako víceprocesorový (SMP – symetrický multiprocessing) víceuživatelský multitaskový univerzální síťový systém.

Zjednodušená struktura systému je naznačena na obr. 2.4. Platí pro Windows NT 4.x, Windows 2000 a XP, ale v hlavních rysech platí i pro předchozí verzi.



Obrázek 2.4: Struktura Windows s NT jádrem do verze XP

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*.

HAL je vrstva abstrakce hardware (Hardware Abstraction Layer), rozhraní mezi hardwarem a zbytkem jádra systému. Při startu systému se načítá ze souboru `HAL.DLL`. Je oddělena od ostatních částí systému z důvodu snadnější přenositelnosti systému mezi (několika málo) hardwarovými platformami. Ovladače komunikují se zařízeními pouze zprostředkovaně přes tuto vrstvu.

Kernel (tzv. „tvrdé jádro“) a *exekutiva* jsou fyzicky uloženy v souboru `NTOSKRNL.EXE`.[4] Exekutiva není z obrázku přímo patrná, můžeme si představit, že stojí za vším v jádře (modrá oblast) nad HAL kromě kernelu.

[4] Ve skutečnosti není jenom `NTOSKRNL.EXE`. Ve víceprocesorovém (resp. vícejádrovém) systému je nahrazen souborem `NTKRNLMP.EXE`; v systému se zapnutou funkcí PAE (přístup k paměti nacházející se za adresovatelnou pamětí) se používá `NTKRNLPA.EXE` pro jednoprocesorový systém a `NTKRPAMP.EXE` ve víceprocesorovém systému. To platí o pojmenování souborů na instalačním CD, po instalaci nebo upgradu na víceprocesorový systém či systém s PAE najdeme v systému pouze názvy `NTOSKRNL.EXE` nebo `NTKRNLPA.EXE` (původně jinak pojmenovaná verze je během kopírování z CD přejmenována).

Kernel zachytává a obsluhuje přerušení, provádí správu procesorů (synchronizace přidělování procesorů), apod. Exekutiva je řídicí proces operačního systému, má na starosti řízení celého jádra běžícího v privilegovaném režimu a provoz modulů. Kernel i exekutiva se při startu systému načítají ze souboru `NTOSKRNL.EXE` (jediný soubor, tedy monolitické jádro), ostatní součásti jádra jsou k jádru linkovány z dynamických knihoven či souborů .sys (moduly, tedy modulární struktura běžícího jádra).

Hlavní systémový proces (v aplikacích pro správu procesů jej vidíme pod názvem „System“) je ve skutečnosti obrazem toho, co běží v jádře, exportovaným do uživatelského prostoru (reálně samozřejmě přímo do jádra nevidíme). Ve skutečnosti nejde o proces, ale o *kontejner* pro prováděcí vlákna jádra (o vláknech se více dovíme v kapitole o správě procesů).

Ovladače nesouvisejí jen se zařízeními, obecně jde o moduly jádra, které mohou sloužit jak k přístupu k zařízením, sběrnicím apod., ale také to mohou být filtry, přes které procházejí data (šifrování, komprimace, směrování mezi moduly, filtrování/zahazování/třídění, DRM, atd.).

Pro práci s ovladači existuje složitý systém, do kterého se zapojuje rovnou několik na obrázku vyznačených součástí, například *Device Models* (modely ovladačů) určují unifikovaný způsob zacházení s ovladači.

Nad ovladači souborových systémů je systém pro jejich správu – IFSM (stejně jako u dříve popisované verze Windows) – Installable File Systems Manager, který zajišťuje přístup k (předem určeným) souborovým systémům (NTFS, FAT32, UDF apod.). Je využíván modulem pro správu vstupů a výstupů a vyrovnávací paměti.

Správce konfigurace spolupracuje při správě ovladačů, například zajišťuje funkce Plug&Play a Hot-Plug, tedy neustále sleduje stav sběrnic a hlídá připojování nových či již dříve připojovaných zařízení, u nových se pokouší provést instalační a inicializační proceduru.

Moduly pro *správu oken a grafiky* běží ve Windows řady NT od verze 4 v režimu jádra z důvodu urychlení práce aplikací hodně využívajících grafická zařízení. Tato část jádra se načítá ze souboru `win32k.sys`.

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 paměti systémových procesů). Další nevýhodou je náročnější postup výměny uživatelského rozhraní za alternativní. Ve Windows Server od verze 2008 je možné instalovat systém bez GUI (a také bez dalších součástí, které jakkoliv GUI vyžadují) – instalace *Server Core*.

Grafický subsystém v jádře je modul GDI, ve Windows XP i jeho nástavba GDI+. Dynamické knihovny `gdi32.dll`, `gdiPlus.dll` a `gdi32Full.dll` jsou pouze přístupovými body k těmto modulům v uživatelském prostoru. GDI+ ve Windows XP je vylepšením původního GDI (například je možné používat jako souřadnice i racionální čísla, ne pouze celá, byla přidána podpora různých 2D operací, dalších formátů souborů včetně JPEG a PNG, atd.).

Bezpečnostní podsystém souvisí především s modulem LSASS (Local Security Authority SubSystem). Provádí autentizaci uživatelů, kteří se přihlašují lokálně, a podle databáze v klíči registru SAM určuje přístupová oprávnění.

Správce služeb SCM (Service Control Manager) se načítá ze souboru `services.exe` a zajišťuje běh *služeb* a komunikaci s nimi. Samotné služby sice běží v uživatelském prostoru, ale obvykle s vyššími oprávněními a bez vazby na konkrétního uživatele, a komunikuje se s nimi především přes modul SCM.

Od Windows NT verze 4 je jádro víceméně objektové. V jádře se udržuje databáze objektů a objekty exekutivy jsou exportovány do uživatelského prostoru. Databázi objektů spravuje *Správce objektů*.

Komunikace procesů s jádrem (například při žádosti o otevření souboru či žádosti o další oblasti operační paměti) probíhá obvykle tímto způsobem:

- proces spustí určitou funkci či proceduru z *dokumentovaného* nebo *nedokumentovaného API*, případně dokumentovaná funkce může zavolat nedokumentovanou funkci,
- provede se *systémové volání* v kontextu jádra,
- v rámci systémového volání je požadavek vyřešen.

Dokumentované rozhraní představují funkce a objekty ze systémových knihoven, jejichž názvy by nám měly být povědomé – `User32.dll`, `GDI32.dll` a další. Jejich určení je vpodstatě podobné tomu, co jsme se učili u starších systémů, rozdíly jsou ve způsobu naprogramování.

Nedokumentované API je soubor `NTDLL.DLL`. Funkce, které se zde nacházejí, již mohou přímo spouštět systémová volání – mohlo by se tedy zdát, že je lepší používat hned nedokumentované API (bylo by to rychlejší), ale problém je v tom, že funkce poskytované tímto rozhraním mohou být v každé verzi Windows jiné a mohou vyžadovat jiný způsob volání (spouštění). Nepříjemným důsledkem je, že aplikace používající nedokumentované API nemusí v některých verzích fungovat vůbec nebo se při jejím běhu můžeme setkávat s různými problémy souvisejícími s nekompatibilitou. Proto je lepší používat dokumentované API, které se vždy chová stejně (dokumentovaně). Soubor `NTDLL.DLL` je tedy jakýmsi rozhraním mezi jádrem a uživatelským prostorem, ale obvykle k tomuto rozhraní přistupujeme zprostředkovaně.

Podsystémy (subsystémy) prostředí jsou rozhraní zajišťující správný a bezpečný běh různých typů procesů. V těchto podsystémech běží aplikace, které ani 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“.

Je to především podsystém pro aplikace psané pro 32bitová Windows, MS-DOS a aplikace pro 16bitová Windows *Win32* (jediný podsystém pro všechny tyto typy aplikací, v něm jsou pak spouštěny případné virtuální počítače), podsystém pro OS/2, *POSIX*, atd.

Podsystém pro 32bitová Windows včetně NT (podsystém *Win32*, v 64bitovém systému se jmenuje prostě *Windows*) je představován souborem `CSRSS.EXE`, pro POSIX je to především soubor `PXSS.EXE` (to je server podsystému). Podsystém Win32/Windows je potřebný také pro běh mnoha systémových procesů, proto se jako jediný spouští hned po startu počítače, ostatní podsystémy jsou spouštěny až na žádost při spuštění aplikace patřící tomuto podsystému.

Každý podsystém potřebuje 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 podsystému Win32 patří také knihovny `KERNEL32.DLL`, `USER32.DLL` a `GDI32.DLL`. Tyto tři moduly jsou sice určeny 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.

Součástí podsystému Win32/Windows je mechanismus virtuálních počítačů. Aplikace, která fyzicky zajišťuje běh virtuálních počítačů pro starší aplikace (DOS a Win16), je spouštěna souborem `ntvdm.exe` (NT Virtual DOS Machine). Při pokusu o spuštění těchto aplikací je nejdřív spuštěna nová instance `ntvdm.exe` s parametrem – názvem spouštěné DOS či Win16 aplikace s cestou, která již „vnitřně spustí“ zadanou aplikaci.

Na 64bitovém systému je situace ještě trochu složitější. Podsystém Windows (`CSRSS.EXE`) slouží ke běhu 64bitových aplikací. Pro 32bitové aplikace máme ještě uvnitř podsystému Windows speciální podsystém *WoW64* (Windows-on-Windows), který slouží jako rozhraní pro 32bitové aplikace (32bitový kód se tímto podsystémem překládá na 64bitový, který lze již spouštět přes `CSRSS.EXE`).

Soubor `Win32k.sys` je sice technicky část podsystému prostředí Win32/Windows běžící v režimu jádra (všimněte si, má příponu typickou pro ovladače běžící v režimu jádra), ale ve skutečnosti je využíván všemi podsystémy prostředí včetně POSIXu. Obsahuje implementaci nízkoúrovňových funkcí pro uživatelské rozhraní, volá rutiny v ovladačích GDI zařízení. Je používán všemi podsystémy prostředí především z důvodu usnadnění funkčnosti těchto podsystémů (aby každý z nich nemusel mít vlastní část v jádře), tedy volání jednotlivých podsystémů jsou směrem do jádra překládána na volání generovaná podsystémem Win32.

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

1. Jádro je generováno z jediného souboru (`NTOSKRNL.EXE`), z toho pohledu jde o monolitické jádro.
2. Vrstvená architektura se uplatňuje především v rozdělení na uživatelský režim a režim jádra.
3. 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, ovladačů, atd. (modulů běžících v privilegovaném režimu).
4. 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`).

Jak běží procesy:

- Win32/Win64 aplikace běží ve společném virtuálním stroji, každá má svůj vlastní paměťový prostor (pod toutéž číselnou adresou každá z těchto aplikací vidí různá fyzická umístění v paměti),
- DOS a Win16 aplikace mají každá svůj virtuální stroj, v rámci virtuálního stroje svůj vlastní paměťový prostor.

2.2.4 Windows od verze Vista a Server 2008

Obrázek 2.5 platí především pro Windows 10, třebaže většina je platná i pro verze od Visty výše (ale například Universal Apps v nižších verzích nejsou). Správce oken DWM existuje už od Windows Vista, ale postupně byl hodně přeprogramován (největší změny jsou ve Windows 7 a pak ve Windows 10).

Vista. Jádro Windows Vista bylo oproti svým předchůdcům zcela přepracováno, i když základní principy zůstávají zachovány. Má vnitřní strukturu modulárního typu.

Důležitou změnou oproti jádru Windows XP je ve Vistě implementace IPv6. Dále prakticky celý síťový zásobník (podpora sítě) byl přesunut do režimu jádra, naproti tomu část implementace grafického rozhraní byla z jádra přesunuta do uživatelského prostoru.

Jádro Visty je monolitické stejně jako u XP, ale vnitřní struktura je modulárnější (další krok ke struktuře jádra modulárního typu), důsledky:

- snadnější rozšiřitelnost jádra,

- stejné instalační médium pro všechny varianty Visty (Home Premium, Ultimate, apod.), při instalaci se podle typu licence rozhodne, která varianta bude nainstalována (jsou instalovány jen vybrané moduly),
- nejsou rozlišeny jazykové varianty, existuje samostatný modul pro jazyk.

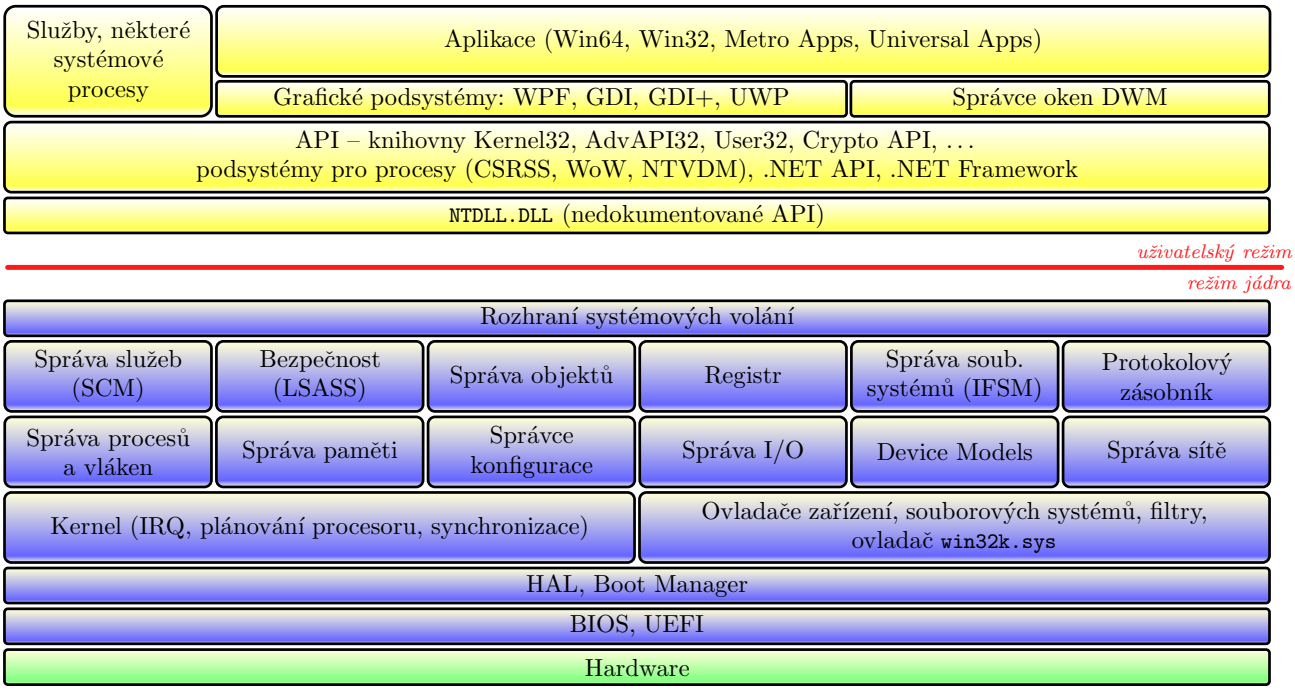
Takže instalační DVD je stejné pro všechny varianty Windows Vista určené pro danou architekturu (32bitovou nebo 64bitovou), na *plnohodnotném* DVD jsou tedy všechny moduly (konkrétně WIM soubory s obrazy modulů) pro jakoukoliv variantu. Existují také samostatné moduly, ve kterých je uloženo pouze jazykové nastavení, ostatní moduly jsou jazykově nezávislé. Důsledkem je, že také opravné balíčky mohou být *jazykově nezávislé* (tj. v neanglicky mluvících zemích není nutné čekat, až bude publikován opravný balíček pro danou jazykovou variantu Windows).

Během instalace je určeno, které moduly budou nainstalovány, a to především typem licence, která instalaci přísluší (například Vista Home Premium znamená jinou vybavenost moduly než Vista Ultimate).

Od Visty SP1 byla přidána podpora UEFI a systém startuje trochu jiným způsobem. To však neznamená, že by tyto systémy nemohly být instalovány na systémech bez UEFI (se starým BIOSem), instalační a bootovací procesy zvládají obojí.

Struktura jádra vypadá podobně jako u starších verzí (dělí se na kernel a exekutivu plus ovladače a další moduly), přičemž většina grafického subsystému se odstěhovala do uživatelského prostoru (k tomu se dostaneme o něco později) a naopak implementace síťových protokolů se objevila v jádře (předtím byla v uživatelském prostoru). Ostatní moduly celkem zůstaly, jen se trochu změnila jejich pracovní náplň, postupně přibyly další modely ovladačů a správa služeb také pracuje trochu jinak.

Naopak v uživatelském prostoru je změn hodně. Část je opět shodná (také máme podsystémy pro běh procesů jako CSRSS, WoW apod. a dokumentované a nedokumentované rozhraní), ovšem grafické prostředí už nestojí jen na klasickém WinAPI (modul GDI), ale staví na WPF.



Obrázek 2.5: Struktura Windows od verze Vista

Windows Presentation Foundation (WPF, také Avalon) je součástí rozhraní .NET Framework. Je to grafický podsystém, tedy především eviduje okna a jiné grafické komponenty vkládané do oken ve stromové struktuře zohledňující jejich vnořování, a zajišťuje správu oken (a dalších grafických komponent). Také plocha je považována za okno, podobně různé panely včetně hlavního panelu plochy. Inspiraci zde Microsoft zřejmě našel u systému X Window z UNIXového světa.

Starší aplikace mají grafické rozhraní naprogramováno v GDI nebo novějším GDI+, novější aplikace již používají WPF. Od Visty je také GUI systému naprogramováno s použitím WPF.

Ovšem modul WPF využívá služeb knihovny/modulu GDI, takže když se podíváme na seznam dynamických knihoven načtených kteroukoliv aplikací s GUI, najdeme tam `gdi32.dll` a případně další podobné soubory.

Srovnání: https://www.leadtools.com/help/leadtools/v19m/dh/to/differencesbetweengdiandwpf.html

Desktop Window Manager (DWM) je kompozitní správce oken (opět připomínka terminologie v X Window), který provádí vykreslování oken, jejichž strukturu spravuje WPF. Zatímco WPF se stará o data, DWM vykresluje, zajišťuje jakési primitivní 3D zobrazení (Flip3D, průhlednost apod.), náhledy, animace, reaguje při změně rozlišení monitoru, atd.

Ve Windows od verze Vista je uplatňována funkce *ASLR* (Address Space Load Randomization) – knihovny se při načítání do paměti (po vyžádání některým procesem) neukládají vždy na stejné místo v paměti jako v XP, ale náhodně na některou adresu ze seznamu. Tato funkce by měla být obranou proti zneužití přetečení paměti (hacker nedokáže odhadnout, na kterou adresu umístit kód, aby po přetečení paměti byl na „vhodném“ místě). Ovšem tato ochrana byla již dávno prolomena, jen její obcházení je časově náročné.

Windows 7. V případě Windows 7 nebylo v jádře provedeno moc změn co se týče funkčnosti, většina změn je ve vnitřní struktuře jádra, v řízení a provozu grafického rozhraní, a také ve způsobu využívání a nastavení systému.

Pro jádro byl použit koncept *MinWin* – co nejmenší základní jádro (téměř mikrojádro), ostatní části „širšího jádra“ (tj. toho, co běží v privilegovaném režimu) jsou moduly, tedy opět další krok k modulárnosti jádra. Do MinWin patří především kernel (tvrdé jádro). MinWin je samostatnější než původní část jádra, důsledkem je rychlejší start systému a celkově lepší odezva (po načtení MinWin je již možné používat více než jedno jádro procesoru, tedy další části systému mohou být načítány paralelně).

Dále zde můžeme vidět určité změny v používání API, využívání virtuálních DLL knihoven. Reálně běží výrazně méně služeb než ve Windows Vista (díky čemuž také běží systém svižněji) a nastavení jsou celkově přizpůsobena novým typům hardwaru (například Windows 7 dokáží při instalaci detekovat SSD disk a přizpůsobit tomu některá nastavení jako například vypnutí služby pro defragmentaci a úprava funkce SuperFetch). Služby spuštěné při běhu systému mohou být dočasně zastaveny, pokud SCM usoudí, že zrovna nejsou zapotřebí. Změny v grafickém rozhraní předpokládám není třeba rozvádět.

Ve Windows 7 přišel Microsoft se zcela novým řešením problémů s nekompatibilitou aplikací. U vybraných verzí lze použít funkci *XP Mode* pro provoz starších aplikací.

Windows 8, 10. Některé součásti jádra byly přepracovány, komunikační struktura se stala složitější (zejména z důvodu podpory DRM u multimédií), nicméně nic z toho na našem obrázku není přímo vidět.

Ve Windows 8 se objevily *Metro Apps*, ve Windows 10 pak *Universal Apps*. Také pro ně bylo třeba vytvořit vlastní grafický subsystém (nesouvisí s .NET, takže žádné WPF) – pro Metro Apps to bylo WinRT API, pro jejich nástupce Univesal Apps to je *Universal Windows Platform* (UWP). V obou případech jde vlastně o totéž (jen se to jinak jmenuje) – Metro Apps jsou určeny pro různé hardwarové platformy (desktopovou i mobilní RT), Univesal Apps taktéž pro různé hardwarové platformy (desktopovou, mobilní, XBox, atd.).

Do jádra Windows 10 se dokonce nastěhoval *Windows subsystem for Linux*, který má umožnit spouštění aplikací programovaných pro Linux.

Hlavní změny jsou opět v uživatelském prostoru a grafickém rozhraní, včetně vybavenosti různými nástroji (uživatelé zaznamenali především nový nástroj *Nastavení*, změny ve správě aktualizací, nové aplikace či naopak odstranění některých aplikací nebo nahrazení univerzálními, novou politiku ve sběru a správě osobních informací, větší tlak na využívání cloudu i včetně autentizace, atd.).

Serverové edice Windows. Serverové edice Windows používají totéž jádro jako desktopové edice, jen je jinak nakonfigurováno, v registru mají některé položky jinou hodnotu (zejména položky týkající se sítě) a máme k dispozici další nástroje. Přímo serverovými edicemi se v tomto předmětu nebudeme zabývat (jen okrajově).

Verze jádra	Serverový systém	Desktopový systém
NT 6.0	Windows Server 2008	Windows Vista
NT 6.1	Windows Server 2008 R2	Windows 7
NT 6.2	Windows Server 2012	Windows 8
NT 6.3	Windows Server 2012 R2	Windows 8.1
NT 10.0	Windows Server 2016	Windows 10

Tabulka 2.1: Serverové a desktopové systémy s odpovídajícími jádry

V tabulce 2.1 vidíme označení verze jádra a označení verzí serverových a desktopových Windows používajících příslušné jádro.

2.3 Systémy UNIXového typu

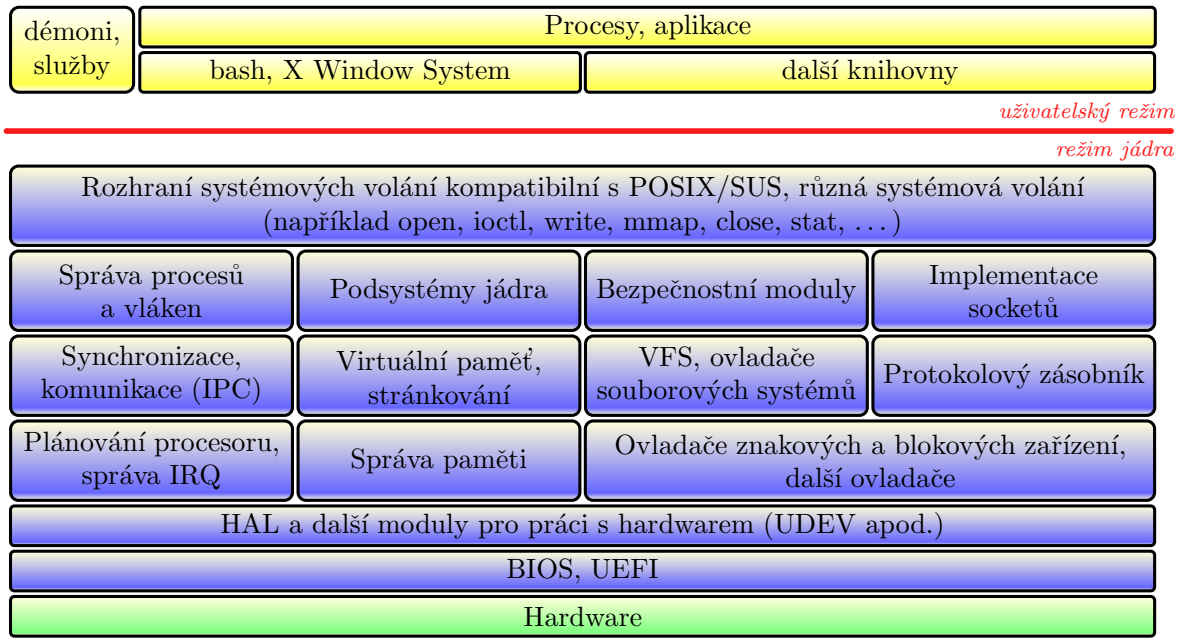
Většina UNIXových systémů má hodně podobnou strukturu (kromě těch, které byly upraveny pro real-time provoz). Jádro běží v privilegovaném režimu (režimu jádra), je často tvořeno jediným souborem a využívá jediný souvislý adresový prostor (proto je nazýváno monolitické, i když jeho vnitřní struktura je přesně rozvržena). Například u Linuxu je to soubor s názvem podobným `/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.

2.3.1 Klasický UNIX

Nákres architektury UNIXových systémů je na obrázku 2.6. Jádro systému se skládá ze dvou oddělených částí – závislé na konkrétní architektuře (HAL – Hardware Abstraction Layer) a kernelu nezávislého

na hardwaru. Jádro je typicky monolitické (načítá se z jednoho souboru a má souvislý adresní prostor), název příslušného souboru závisí na daném systému.



Obrázek 2.6: Struktura operačních systémů UNIXového typu

Vrstva HAL (Hardware Abstraction Layer) slouží jako základní rozhraní k zařízením, které je mimo jiné využíváno ovladači ke komunikaci se zařízeními. Hlavním úkolem HAL je skrýt technické detaily zařízení patřících do různých tříd (skupin s charakteristickými vlastnostmi). HAL zajišťuje načítání ovladačů, vytváření a odstraňování přípojných bodů pro bloková (paměťová) zařízení, a také provozování abstraktního modelu hardwaru.

Modul HAL existoval ve všech starších UNIXových systémech, v některých se však od jeho používání upouští – ve většině distribucí Linuxu a ve FreeBSD roli HAL převzaly další moduly jádra, především modul UDEV nebo obdobný.

Důležitými moduly jádra jsou *ovladače*. Existují ovladače blokových a znakových zařízení včetně síťových a virtuálních zařízení, a další specializované ovladače. Také souborové systémy jsou implementovány jako ovladače.

Souborový systém je v UNIXových systémech vlastně rozhraní. Často se jedná o rozhraní mezi ovladačem vnějšího paměťového média a vyššími vrstvami jádra, ale ve skutečnosti souborový systém vůbec nemusí s paměťovými médii souviset.

Protože v UNIXových systémech 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 Linuxu souborový systém *proc*) nebo sdružující jiné souborové systémy či představující část jiného souborového systému. Souborové systémy, které nenáležejí k žádnému konkrétnímu paměťovému médiu, ale přesto s nimi takovým způsobem zacházíme (pracujeme přes ně se soubory nebo tím, co jako soubory vypadá), nazýváme *virtuální souborové systémy*.

VFS (Virtual File System, virtuální souborový systém) je nejdůležitějším virtuálním souborovým systémem. 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 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.).

FUSE[5] (FileSystem in User Space) je mechanismus, který umožňuje běh souborových systémů v uživatelském prostoru (běžné souborové systémy musejí být součástí jádra). Spočívá v rozdělení souborového systému do dvou částí – spodní část, modul FUSE, je pro všechny souborové systémy tohoto typu společná a běží v režimu jádra. Horní část běží v uživatelském režimu a využívá služeb modulu FUSE. Tímto způsobem je v současné době implementováno mnoho souborových systémů (například ntfs-3g a ntfsmount pro provoz NTFS, souborové systémy pro kompresi a šifrování dat, multimediální rozhraní, sledování systému, správa verzí, atd.).

✎ Implementaci síťového protokolového zásobníku také najdeme v jádře (přesněji protokolových zásobníků, protože nemusí jít jen o TCP/IP), a také implementaci socketů.

✎ *Podsystémů (subsystémů) jádra* existuje poměrně hodně, každý má svou specifickou funkci, například *šifrovací podsystém*, *multimediální podsystém* (služby pro práci se zvukem a videem), *podsystém IPC* (mechanismy komunikace mezi procesy).

Bezpečnostní moduly jsou vlastně také podsystémy. Záleží na konkrétním systému, které bezpečnostní moduly jsou načteny, obvykle to bývá firewall (například v Linuxu NetFilter), dále modul pro zvýšení zabezpečení systému (v Linuxu často SELinux), AppArmor a další.

✎ *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). Hlavní úlohou je zajištění bezpečnosti, znemožnění zásahu uživatele do jádra. *Systémová volání* jsou vlastně funkce, kterými lze komunikovat s jádrem.

✎ V systému existuje velké množství knihoven, z nichž v Linuxu je nejdůležitější knihovna `glibc` (GNU C Library), v dalších UNIXových systémech je to `libc`. Tato knihovna zprostředkovává komunikaci procesů z uživatelského prostoru s rozhraním systémových volání, tedy procesy zasílají systémová volání právě této knihovně.

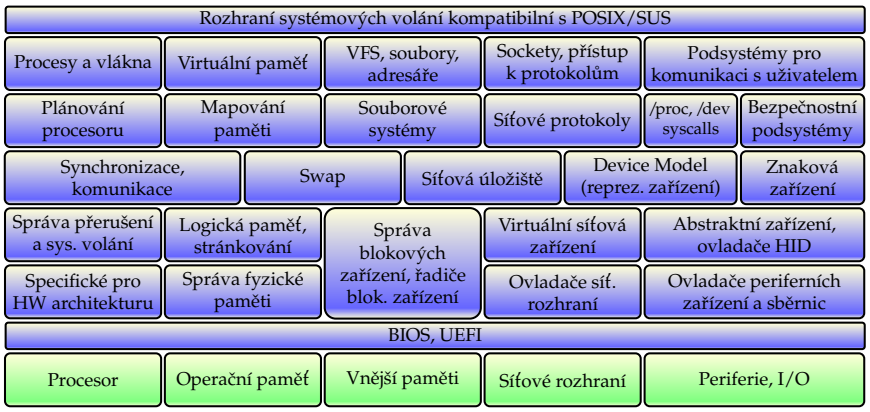
✎ *Shell* je rozhraní pro komunikaci s uživatelem. UNIXové systémy obvykle nabízejí více různých shellů, v Linuxu máme většinou *bash*. Komunikace probíhá v textové formě (uživatel zadává příkazy, systém reaguje textovými výpisy), ale současné UNIXové systémy mají také velmi propracované grafické rozhraní (obvykle založené na X Window).

2.3.2 Podrobněji k jádru Linuxu

Na obrázku 2.7 je podrobnější obrázek linuxového jádra. Všimněte si, že chybí vrstva HAL, v novějších verzích Linuxu ji už nenajdete. Její funkce převzaly jiné moduly jádra, především UDEV.

Tento obrázek je cíleně vytvořen tak, aby ve sloupcích byly nad sebou ty součásti, které spolu významově souvisejí, i včetně vazby na konkrétní kus hardwaru. Některé součásti souvisejí se dvěma hardwarovými komponentami, například swap souvisí s operační pamětí (protože stránky z ní se odkládají) a zároveň s paměťovými médii (protože na ta se odkládá). Síťová úložiště souvisejí se sítí a zároveň s paměťovými médii. Modely zařízení se vztahují jak k síťovým rozhraním, tak i k dalším I/O zařízením.

[5] http://fuse.sourceforge.net/



Obrázek 2.7: Jádro Linuxu v novějších verzích

Další informace:

Velmi zajímavou a podrobnou mapu jádra Linuxu najdeme na adrese
http://www.makelinux.net/kernel_map (mapu zvětšíme rolovacím kolečkem na myši)

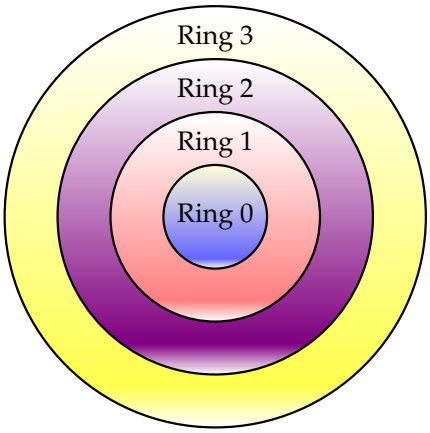
2.4 Hardwarové zabezpečení systému

Moderní operační systémy využívají hardwarovou ochranu prostředků. Na procesorech rodiny x86 je tato ochrana implementována ve formě čtyř okruhů – Ring 0, Ring 1, Ring 2 a Ring 3.

Každý proces běží v některém z těchto okruhů, což určuje jeho možnosti přístupu k chráněným prostředkům, což je především paměť, I/O porty, obecně přímý přístup k hardwaru a dále používání některých strojových instrukcí.

Většina operačních systémů používá pouze dva okruhy – Ring 0 pro jádro systému a Ring 3 pro ostatní procesy. Ring 0 představuje režim jádra (privilegovaný režim) a Ring 3 uživatelský režim. Na nákresech architektur operačních systémů v předchozích sekcích této kapitoly jsou oba režimy obvykle barevně odlišeny (v barvách podle obrázku 2.8).

Z výše uvedeného vyplývá, že Ring 1 a Ring 2 obvykle nejsou používány. Přesto je lze využít pro další rozškálování přístupových oprávnění a například s Ring 1 se setkáme u některých virtualizačních technik, zejména na serverech.



Obrázek 2.8: Hardwarová ochrana prostředků

Kapitola 3

Správa paměti

Rychlý náhled: 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.

Klíčová slova: správa paměti, fyzická adresa, logická adresa, virtuální paměť, stránkování, segmentace, copy-on-write, mapování paměti

Cíle studia: Po prostudování této kapitoly porozumíte tomu, jak operační systémy zacházejí s pamětí, jak ji evidují, přidělují procesům a chrání před neoprávněnými zásahy, také jak se řeší problémy s nedostatkem operační paměti.

3.1 Modul správce paměti

Modul správce paměti je v operačních systémech většinou součástí jádra. Jeho implementace může být různá, ale funkce jsou obvykle podobné:

1. Udržuje informace o paměti (která část je volná, která část je přidělena, kterému procesu je přidělena, atd.).
2. Novému procesu přiděluje paměťový a adresový prostor, do adresového prostoru mapuje všechny potřebné soubory.
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.
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.

Ochrana paměti je v současných běžných operačních systémech zajišťována hardwarově tak, jak je popsáno v kapitole 2.4 na straně 26.

Fyzicky je operační paměť umístěna na základní desce, ale také na rozšiřujících kartách (deskách, adaptérech). Například část operační paměti, kterou nazýváme *videopaměť*, 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, tedy zavést metriku – *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. Pokud tato posloupnost začíná hned na začátku paměti, pak 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 přímo v dané paměti (třeba v čipech operační paměti),
- *logický adresový prostor* – adresový prostor, který je k dispozici (a vypadá, že je v čipech operační paměti, ale může být fyzicky jinde), je vždy souvislý.

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“ paměťovými oblastmi na vnějším paměťovém médiu (obvykle pevném disku), pak je logický adresový prostor větší než fyzický, logický adresový prostor se pak rozprostírá přes několik fyzických paměťových komponent (např. operační paměť a oblast na disku).

Pokud je možné, aby logický adresový prostor byl větší než fyzický, pak hovoříme o *virtuálních metodách přidělování paměti*.

✎ *Virtuální adresový prostor* jádra/procesu – tyto adresy již používá přímo jádro či konkrétní proces, překládají se na fyzické adresy, ale virtuální adresa dvou různých procesů obvykle vede na různé fyzické adresy. Jde vlastně o logický adresový prostor pro konkrétní proces (či jádro). Takže rozlišujeme tři druhy adres:

- fyzické adresy – přímo ve fyzické paměti (pokud se používá odkládací prostor na disku, pak adresy v něm jsou také fyzické),
- logické adresy – v souvislém adresovém prostoru, při použití se mapují na fyzické adresy v odpovídající paměti,
- virtuální adresy – typ logických adres pro konkrétní proces či jádro.

Překlad mezi virtuální (příp. logickou) adresou provádí modul, který bývá součástí procesoru – *MMU* (Memory Management Unit).

3.2 Základní 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.

Evidence přidělené paměti je prakticky na všech architekturách hardwarově závislá.

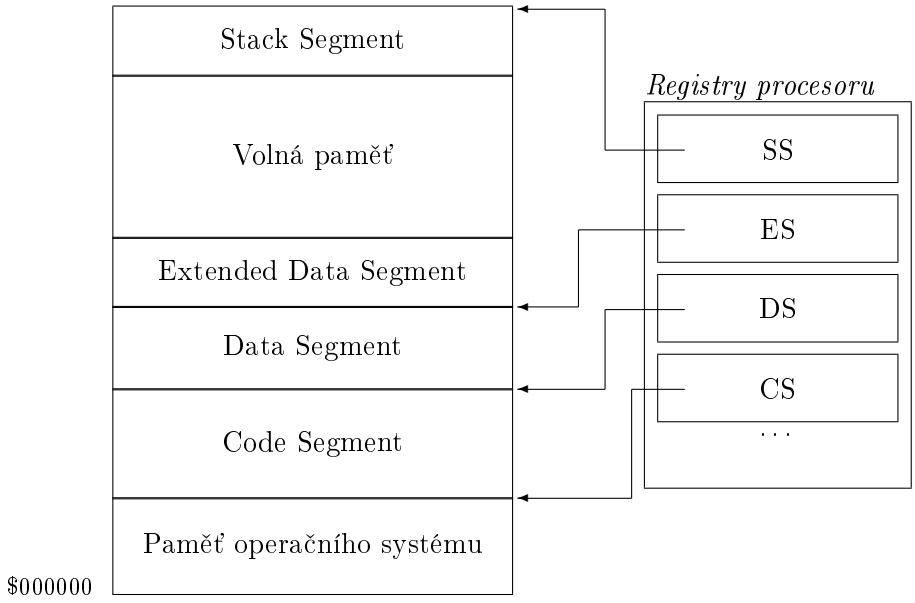
3.2.1 Segmentace

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

Každý segment obvykle mívá určitý účel, například

- *code segment* – po startu procesu je utvořen tento segment a ze spustitelného souboru je sem načten kód procesu, tento segment může být (nemusí – záleží na systému) z bezpečnostních důvodů připojen pouze pro čtení, protože se do něj typicky nezasahuje, jeho velikost je konstantní po celou dobu běhu procesu,
- *data segment* – pro globální konstanty a proměnné, se jeho velikost nemění, ale musí být připojen i pro zápis (hodnoty globálních proměnných se mění),
- *stack segment* (zásobníkový segment) – pro lokální data funkcí (kdykoliv je v procesu zavolána funkce, vytvoří se její *aktivační záznam* obsahující reálné parametry funkce, lokální proměnné a prostor pro uložení návratové hodnoty funkce, tento aktivační záznam se uloží na vrchol zásobníku v tomto segmentu); velikost segmentu se průběžně mění podle toho, kolik funkcí je momentálně rozpracovaných.

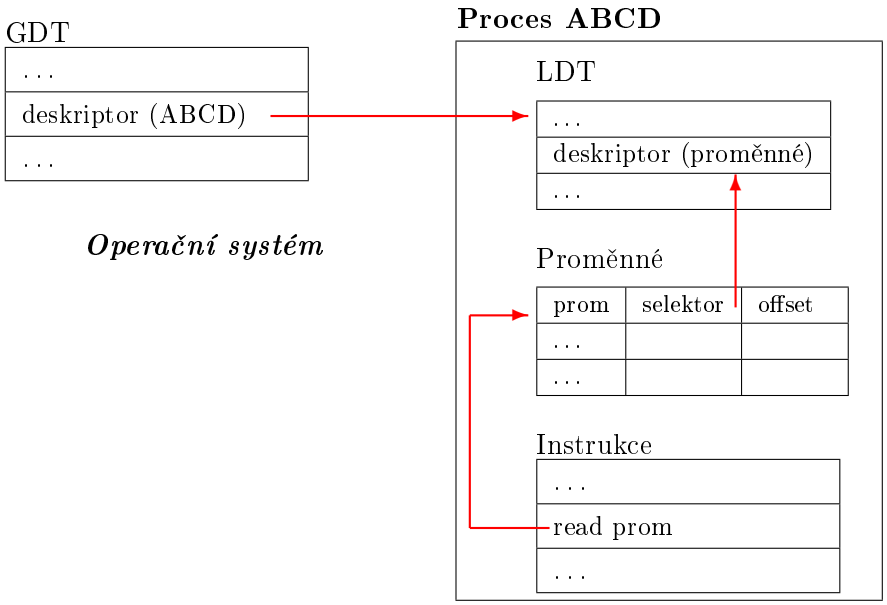
Zatímco segment s kódem a datový segment se obsazují od nejnižších adres k nejvyšším, zásobníkový to má naopak – roste směrem od nejvyšších adres k nejnižším, aby v případě přetečení zásobníku (což je běžná hackerská taktika) byl postižen spíše datový prostor téhož procesu než datový prostor toho „vedlejšího“.



Obrázek 3.1: Segmentace paměti v starším systému (segmenty jediného procesu)

Procesy, které jsou instancemi téhož programu, mohou sdílet plně konstantní segmenty (pokud to systém umožňuje), to se týká především segmentu s kódem.

V současných operačních systémech má typicky každý proces dva zásobníkové segmenty. Jeden z nich používá při volání vlastních funkcí (v uživatelském režimu), druhý se používá v privilegovaném



Obrázek 3.2: Tabulky deskriptorů

režimu při systémových voláních: pokud proces žádá službu od jádra, pro její splnění mu poskytne právě tento zásobník.

Segment je tedy paměťový blok určený pro jeden konkrétní účel, jehož délka je danému účelu přizpůsobena.

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 adresové/segmentové 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.

Metoda segmentace paměti se běžně používá v moderních operačních systémech (jak ve Windows, tak i v UNIXových systémech), ale vždy v kombinaci s jinou metodou. Se zjednodušenou variantou jsme se mohli setkat u MS-DOSu a dalších operačních systémů pro první minipočítače.

Absolutně jsou adresovány pouze začátky segmentů, uvnitř segmentu se používá relativní adresa. Například první prvek (například proměnná) uvnitř datového segmentu by měl adresu 0.

32bitová architektura x86. Následující platí pro 32bitové procesory Intelu a AMD. *Deskriptor segmentu* je datová struktura v operační paměti jádra, která obsahuje informace o konkrétním segmentu. Najdeme zde

- adresu začátku segmentu,
- velikost segmentu (každý segment může být jinak dlouhý), pro segmenty s proměnlivou délkou je to maximální velikost,
- typ segmentu (code, data, stack),

- příznak úrovně oprávnění (systémový nebo uživatelský),
- příznak určující, zda byl od poslední kontroly použit (k tomu se vrátíme v tématu o virtuální paměti), atd.

 Operační systém si vede pro každý proces *Lokální tabulku deskriptorů* (LDT, Local Descriptor Table), v této tabulce jsou právě deskriptory segmentů používaných daným procesem. Dále existuje (jedna) *Globální tabulka deskriptorů* (GDT, Global Descriptor Table), ve které jsou deskriptory globálně platných segmentů (včetně segmentů jádra) a také deskriptory lokálních tabulek deskriptorů všech procesů.

Lokální tabulky jsou dohledatelné pomocí globální tabulky, globální tabulku systém najde díky tomu, že její adresa je uložena ve speciálním registru procesoru, který označujeme *GDTR register*.

 Pro adresování segmentů se používají *selektory*. Selektor je obdoba pointeru, ale kromě adresy položky v tabulce LDT nebo GDT obsahuje i další informace (řídicí bity): informaci, zda jde o položku z LDT (hodnota 1) nebo GDT (hodnota 0), a pak dva bity pro úroveň oprávnění podle hardwarových okruhů (používá se 00 pro režim jádra a 11 pro uživatelský režim, viz konec předchozí kapitoly). Adresa konkrétního místa v paměti v daném segmentu je pak vektor se dvěma hodnotami:
`(selektor,offset)`

V procesoru se používají segmentové registry CS (Code Segment), DS (Data Segment), SS (Stack Segment), ve kterých je vždy aktuální hodnota selektoru pro ten proces, jehož kód je právě zpracováván (případně pro jádro, pokud se zpracovává kód jádra). Existují i další segmentové registry – FS a GS, jejichž využití závisí na operačním systému.

 Z programování si možná pamatujete *heap segment*, tedy segment pro dynamicky alokované proměnné. Heap bývá většinou softwarově implementovaný a nachází se uvnitř datového segmentu, zatímco dříve začínal na konci datového segmentu a rostl proti zásobníkovému segmentu (tedy tyto segmenty šly „proti sobě“).

 **64bitová architektura amd64.** Pro architekturu amd64 (která se používá především v současných procesorech od Intelu a AMD) platí do určité míry vše, co bylo uvedeno pro x86 s určitými rozdíly. Význam segmentů je potlačen, jsou zde především kvůli zpětné kompatibilitě adresování. Registry CS, DS a SS obsahují v adresní části číslo 0, všechny tedy ukazují na začátek virtuálního adresového prostoru daného procesu, což vypadá, jako by se adresový prostor těchto segmentů překrýval. Ve skutečnosti se díky mapování stránek (na což se podíváme později v této kapitole) navzájem nepřekrývají a nepřepisují. Registry FS a GS jsou používány pro správu vláken procesů.

Výše uvedené platí pro code a data segment (jak pro jádro, tak i pro procesy. Zásobníkový segment je buď implementován jako na architektuře x86, nebo softwarově stejně jako heap.

 **Procesory ARM.** Architektura ARM nepoužívá segmenty, ale Linux běžící na architektuře ARM používá softwarově implementované segmenty (všechny).

3.2.2 Jednoduché stránkování

 Metoda stránkování stojí na využití logických adres. Rozlišuje *fyzickou adresu* objektu v paměti (nebo v odkládacím prostoru) určující skutečné umístění tohoto objektu a *logickou adresu* tohoto objektu (s tou pracují procesy). Paměťový prostor je rozdělen na stejně dlouhé úseky – *stránky*, pokud možno spíše menší velikosti, procesu je přiděleno tolik úseků, kolik potřebuje. Velikost stránky by měla

být *mocninou čísla 2* (typicky 1024 B, 2048 B, nejčastěji 4096 B = 4 KiB, ale mohou být i v jednotkách MB), aby bylo možné provádět rychlé operace s adresami pomocí bitových posunů.

Postup

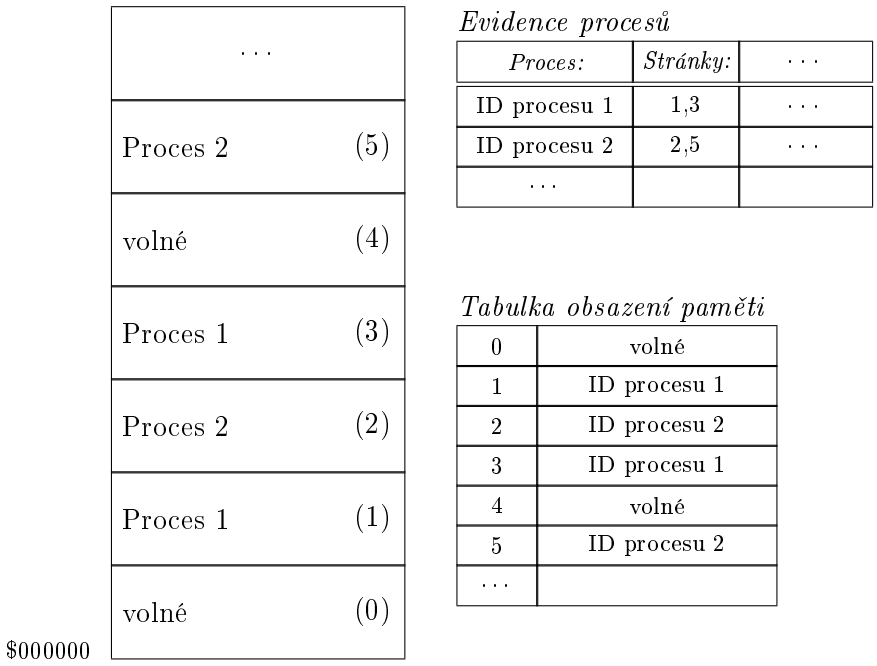
Používané velikosti stránek závisejí na hardwarové platformě, která pro ně určuje povolený interval. Ovšem instalovaný operační systém si z tohoto intervalu vybere vhodnou velikost, přičemž obvykle bere v úvahu také celkové množství dostupné paměti.

Na architektuře x86 se používají stránky o velikosti 4 KiB až 4 MiB. Pokud se však používá PAE (Physical Address Extension) rozšiřující fyzický adresový prostor nad 4 GiB, může být rozložení velikostí stránek komplikovanější.

Architektura amd64 používá stránky velikosti od 4 KiB do 2 MiB nebo 1 GiB (to je možné, pokud má procesor nastaven příznak `PDPE1GB`).

Architektury procesorů ARM používají stránky o velikosti mezi 4 KiB a 1 MiB nebo 16 MiB (to záleží na konkrétním výrobci).

Procesu se jeho adresový prostor jeví jako spojitý, třebaže fyzicky spojitý nemusí být. Proces používá ze svého hlediska „absolutní adresy“, které jsou ve skutečnosti pouze logickými adresami (od nuly) nebo se jako v případě segmentace paměti skládají ze dvou částí – adresy segmentu a offsetu, které je nutno překládat.



Obrázek 3.3: Stránkování paměti

Překlad adres. 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.

Předpokládejme, že proces nahlíží na svůj adresový prostor jako na spojitý. Velikost jeho adresového prostoru je

$$velikost\ prostoru = počet\ stránek\ procesu \times velikost\ stránky$$

Pak má k dispozici adresy v rozmezí $0 \ldots (velikost\ prostoru - 1)$.

Při jakémkoliv přístupu do paměti správce paměti provádí *překlad adres*:

- $index\ stránky\ procesu = logická\ adresa\ /\ velikost\ stránky$ (celočíselné dělení)
- $offset = logická\ adresa\ \%\ velikost\ stránky$ (zbytek po celočíselném dělení)
- $fyzická\ adresa = \Big(mapuj\ stránku(index\ stránky\ procesu) \times velikost\ stránky\Big) + offset$

Mapování stránek se provádí podle seznamu stránek přidělených procesu.

✎ Proces používá pro adresaci objektů ve své paměti pouze jediné číslo – logickou adresu. Podle předchozího výpočtu to vypadá, že se tato jedna adresa překládá pomocí celočíselného dělení na dvě čísla určující fyzickou adresu (index stránky procesu a offset). Aby se překlad co nejvíc zjednodušil, už samotná logická adresa je rozdělena do dvou částí (určitý počet bitů je v jedné části, zbytek v druhé):

- Virtual Page Number (VPN) – z této části se počítá index stránky procesu,
- offset – tato část tvoří offset i pro fyzickou adresu.

Takže jen první část logické adresy musí být příslušně zpracována, druhá část je jen bitově přičtena k výsledku. Tento postup se používá prakticky na všech dnes běžných hardwarových architekturách.

✎ **Tabulky stránek.** Informace o stránkách jsou v tabulce stránek (Page Map Table, jednoduše Page Table). Tabulka stránek se nachází v paměti jádra operačního systému, její adresa je v jednom z registrů procesoru (v procesorech od Intelu a AMD je to registr CR3).

Jeden řádek takové tabulky se označuje jako Page Table Entry (PTE) a obsahuje údaje o jedné konkrétní stránce. Najdeme tam například určení fyzického umístění stránky (číslo rámce, ve kterém je stránka momentálně načtena, viz dále u virtuální paměti), bity ochrany paměti (například R/W bit určující, zda je stránka jen pro čtení nebo i pro zápis, User/Supervisor bit pro úroveň oprávnění), a pak další provozní bity. Konkrétní položky v PTE závisejí na hardwarové architektuře.

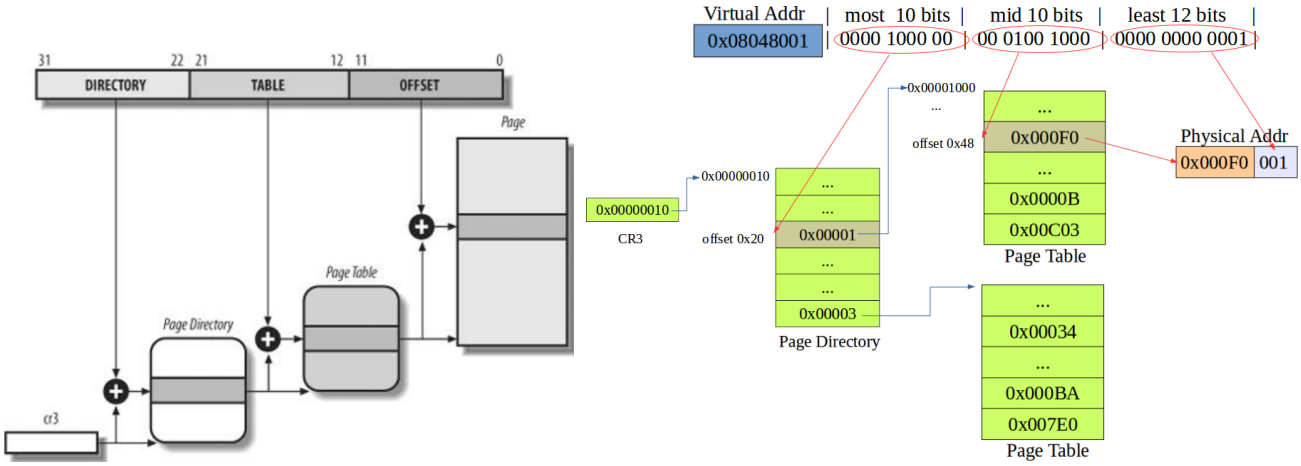
✎ Většina architektur používá víceúrovňové stránkování. Výše uvedený identifikátor stránky VPN se pak skládá ze dvou až čtyř částí (pokud tedy budeme brát v úvahu i offset, pak se celá logická adresa skládá ze tří až pěti částí). Tabulky jsou uspořádány do stromové struktury (se dvěma až čtyřmi úrovněmi), přičemž poslední úroveň (listy stromu) tvoří skutečné tabulky stránek, předchozí úrovně jsou tzv. Page-Directory Tables (adresáře stránek), jakési kontejnery. Ke konkrétní stránce se dostáváme přes nadřízené kontejnerové tabulky.

Logická adresa vypadá nějak takto (zde pro tříúrovňové adresování):

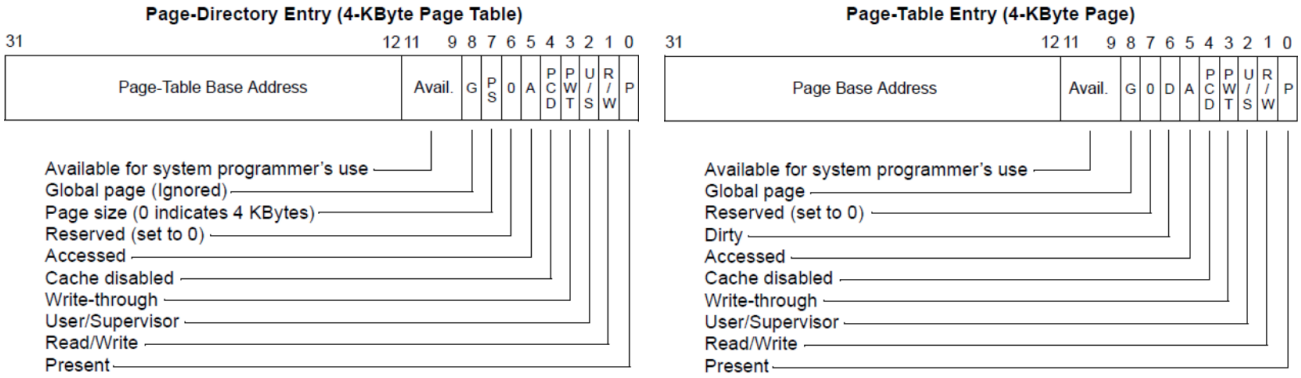
Dir level 1 index	Dir level 2 index	Page table index	offset

Jak se tato adresa používá:

1. první část logické adresy vede do (kontejnerové) tabulky v první úrovni stromu, v příslušném záznamu PTA najdeme odkaz na kontejnerovou tabulku následující úrovně,
2. v nalezené kontejnerové tabulce použijeme druhou část adresy, v příslušném záznamu PTA je odkaz na tabulku třetí úrovně, což už je skutečná tabulka stránek,
3. v tabulce stránek podle třetí části adresy najdeme umístění stránky v paměti (nebo v odkládacím prostoru), tedy zjistíme fyzickou adresu,



Obrázek 3.4: Dvouúrovňové stránkování na architektuře x86[1]



Obrázek 3.5: Struktura záznamu v kontejnerové tabulce (PDE) a v tabulce stránek (PTE) na architektuře x86 při použití dvouúrovňového stránkování[1]

4. přičteme poslední část logické adresy (offset) a máme výslednou fyzickou adresu hledaného objektu.

Může se to zdát zbytečně složité, ale pro používání víceúrovňových stránek existují důvody:

- některé části adresového prostoru nejsou používány (nejsou přiřazeny žádnému prostoru ani jádru) a není nutné tyto části zohledňovat v tabulkách stránek (takže pro ně neexistuje tabulka stránek poslední úrovně),
- pokud se používají malé stránky (4KiB stránky), pak určitě existuje velké množství stránek, při jednoúrovňovém adresování bychom proto museli používat velmi dlouhé adresy; jestliže však máme více úrovní, pak si (i díky prvnímu bodu tohoto seznamu) vystačíme s méně bity na adresu (i přes to, že je více částí logické adresy),
- typická délky adresy dnes bývá 64 bitů (obvykle používáme 64bitové systémy), zde je víceúrovňové stránkování nezbytné, opět v návaznosti na předchozí bod,
- MMU při víceúrovňovém stránkování pracuje s menšími čísly, což zvyšuje propustnost systému (i přes to, že při zpracování jedné logické adresy postupně pracuje se dvěma až čtyřmi čísly).

[1] Zdroj: http://www.renyujie.net/articles/article_ca_x86_5.php

3.3 Virtuální paměť

3.3.1 Odkládací prostor a stránky

Virtuální paměť je koncept, kdy oblast vnitřní paměti rozšíříme o oblast na vnějším paměťovém médiu, obvykle pevném disku. Pro procesy je tento koncept naprosto transparentní (stejným způsobem se z pohledu procesu zachází se všemi adresami v paměti, ať už se fyzicky nacházejí kdekoliv), proto hovoříme o virtualizaci paměti. Adresní prostor, který „vidí“ procesy, je *logická paměť*. Protože každý proces (včetně jádra) má svůj vlastní logický adresový prostor, hovoříme o *virtuálním adresovém prostoru* daného procesu.

Důvodem virtualizace paměti je odstranění základní nevýhody všech reálných metod přidělování paměti, tedy 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 metody stránkování v kombinaci s jinou metodou.

Odkládací prostor sloužící k nastavení vnitřní paměti může být buď celý diskový oddíl nebo jeden či více souborů (záleží, co umožňuje daný operační systém). Není vhodné vytvořit odkládací oblast na SSD (zbytečně se rychleji snižuje jeho životnost, navíc SSD mívají menší kapacitu), lepší volbou je klasický disk, třebaže je pomalejší, případně mít tolik operační paměti, aby odkládací prostor nebyl nutný. Typicky se odkládací oblast označuje jako swap, odkládací soubor/oddíl, stránkovací soubor apod.

Fyzická vnitřní paměť a také odkládací oblast jsou rozděleny na *rámce* a logická paměť je rozdělena na *stránky*. Všechny rámce a stránky mají stejnou velikost. Rámec slouží k umístění stránky, je to jakýsi slot, do kterého ukládáme stránku. Každá stránka má tedy přiřazen právě jeden rámec, a to buď v operační paměti nebo v odkládacím prostoru.

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). Taková nepoužívaná stránka tedy klidně může být umístěna do rámce v odkládacím prostoru a místo v operační paměti bude přenecháno stránce, se kterou se více pracuje.

Jak jsme se dozvěděli v sekci o stránkování, evidence paměti je vedena v *tabulce stránek*, tam je kromě jiných informací uvedeno, kde se momentálně nachází (bit „Valid“ nebo „Present“ určující, zda je v operační paměti nebo v odkládacím prostoru). Ať už se jedná o operační paměť nebo odkládací prostor, v obou těchto umístěních jsou rámce organizovány naprosto stejně – všechny jsou stejně velké (přesně tak, jak velké jsou stránky) a očíslovány, aby bylo možné v daném umístění jednoznačně identifikovat rámec. Číslo rámce s umístěnou stránkou je pak také uvedeno v záznamu stránky v tabulce stránek.

Takže celkový postup nalezení fyzického umístění stránky je následující:

1. pro stránku najdeme její záznam v tabulce stránek PTE (podle postupů probíraných výše, včetně víceúrovňového stránkování),
2. v záznamu PTE ověříme v příznaku „Valid“, zda je v operační paměti nebo odložená,
3. v záznamu PTE najdeme číslo rámce, podle kterého již dohledáme (v operační paměti nebo v odkládacím prostoru) danou stránku.

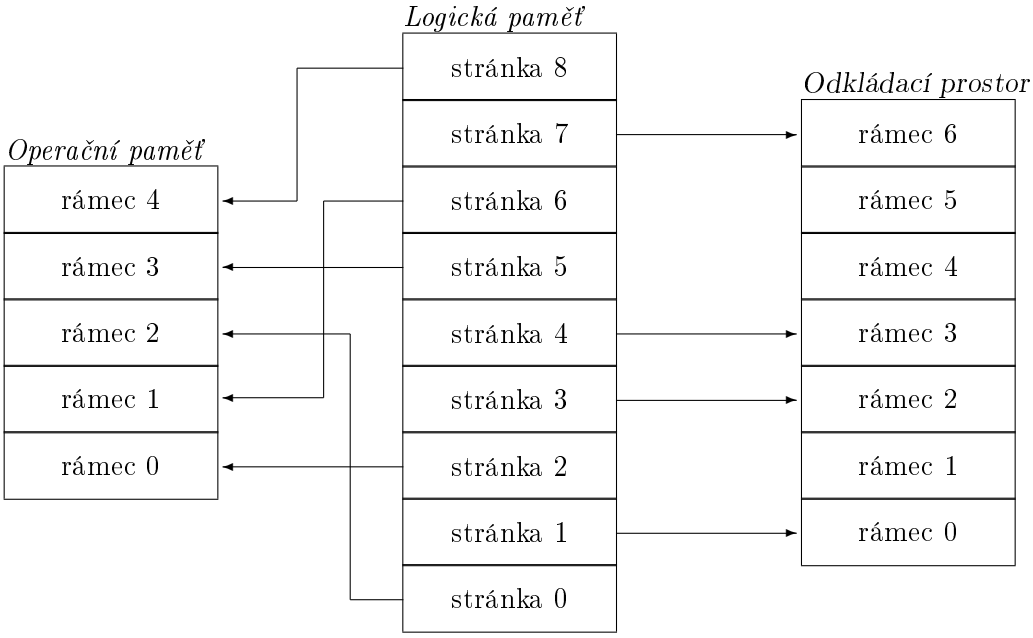
3.3.2 Stránkování na žádost

Metoda stránkování na žádost využívá právě výše popsaný způsob práce s pamětí a odkládacím prostorem a přidává postupy pro transfer požadované stránky z odkládacího prostoru do operační paměti. Ta žádost v názvu tedy značí, že určitý postup se provede právě při požadavku procesu na přístup k obsahu konkrétní stránky (například některé proměnné v datové oblasti paměti).

Proces ke svým stránkám přistupuje takto:

a) *Stránka nachází ve fyzické paměti*: pak této stránce odpovídá některý rámec v paměti. Zpracování žádosti probíhá přesně podle výše popsaného postupu, tedy podle logické adresy se v tabulce stránek najde záznam PTE a podle něj se vypočte fyzická adresa objektu, k němuž chce proces přistupovat.

b) *Stránka je odložena na disku*: tento fakt se zjistí z nalezeného záznamu PTE dané stránky (podle bitu „Valid“). Správce paměti vyvolá přerušení nazvané *výpadek stránky* (page fault), čímž se přeruší zpracovávání kódu žádajícího procesu, správce paměti tím získá čas na provedení následných kroků. Požadovanou stránku je totiž nutné nejdřív dostat z odkládacího prostoru do paměti. Po tomto transferu je požadavek na přístup ke stránce zopakován a nyní už probíhá podle bodu a).



Obrázek 3.6: Stránkování na žádost

Při určování, který rámec v paměti má být uvolněn v případě, že je nutné některou stránku přesunout z odkládacího prostoru do paměti, používáme některou z *metod výběru oběti*. Dále jsou popsány nejznámější metody výběru oběti (obětí je stránka, která původně byla umístěna v rámci v paměti, ale bude odsunuta do odkládacího prostoru, aby uvolnila místo požadované stránce).

Jaká kritéria by měl splňovat ideální algoritmus výběru oběti?

- chceme, aby se tento algoritmus nemusel spouštět moc často (tj. aby co nejméně probíhaly page faults), tedy aby byly přednostně odkládány ty stránky, které pravděpodobně v nejbližší době nebudou používány,

- samotný algoritmus by neměl být moc výpočetně náročný, tedy aby související evidence a hledání oběti byly co nejoptimálnější.

 **Optimal (OPT).** K odsunutí je vybrána ta stránka v paměti, která nebude v nejbližší budoucnosti svým vlastníkem používána. Tento algoritmus je sice bezkonkurenčně nejoptimálnější, protože takto odsunutá stránka určitě v nejbližší budoucnosti nebude požadována a tedy nebude nutné brzy znovu spouštět algoritmus hledání oběti, ale žádný operační systém nedokáže spolehlivě předpovědět chování procesů. Tedy sice optimální, ale nepoužitelné.

 **First-in-First-out (FIFO).** Odsunuta je ta stránka, která byla v paměti nejdéle. Implementace této metody je celkem jednoduchá: správce paměti si vede frontu stránek s přiděleným rámcem v paměti. Pokud je stránka umístěna do rámce v paměti (například při přesunu z rámce v odkládacím prostoru), pak je zařazena na konec fronty. Když je nutné najít oběť k odsunu do odkládacího prostoru, je vybrána stránka na začátku fronty, protože ta je v operační paměti přítomna nejdéle.

Výhodou je tedy jednoduchost, ale nevýhodou je to, že stránky, na které je velmi často přistupováno (a které by primárně měly v paměti co nejvíce zůstávat), jsou zbytečně často odsouvány do odkládacího prostoru. Stránka, na kterou je přistupováno jen občas, je v paměti prakticky stejně dlouho jako stránka, na kterou je přistupováno často. Tato metoda není ani zdaleka optimální, nezohledňuje frekvenci používání stránek.

 **Least Recently Used (LRU).** Přednostně jsou odsouvány ty stránky, které se v poslední době nejméně používaly. Tento algoritmus je založen na heuristice: pokud stránka v blízké minulosti nebyla používána, tak asi nebude používána ani v blízké budoucnosti. Je několik možných implementací:

- V záznamu PTE o dané stránce v tabulce stránek je čítač, který se vždy při přístupu na stránku zvýší o 1. Při nutnosti uvolnit rámec v paměti hledáme jako oběť stránku, jejíž čítač je nejmenší číslo.

  V zájmu spravedlnosti je třeba čítače stránek čas od času nulovat, aby například stránka právě vzniklého procesu nebyla hned odsunuta do odkládacího prostoru (protože na ni ještě nebylo přistupováno), a také abychom se vyhnuli práci s příliš velkými čísly. Navíc tato metoda znamená poměrně hodně přístupů k záznamům PTE, protože procesy své stránky používají poměrně intenzívně. Nevýhodou je i to, že při hledání oběti porovnáváme velká čísla.
- Můžeme používat časové razítko v PTE při každém přístupu na stránku. Tím sice systém zpřehledníme a nemusíme se bát přetečení čítače, ale opět pořád pracujeme s velkými čísly (časová razítka zabírají několik oktetů).
- Další možnost je vést si frontu stránek, ale na rozdíl od metody FIFO se stránka dostane na konec fronty tehdy, když je na ni přistupováno. Na začátku fronty tedy máme tu stránku, na kterou se nejdéle nepřistupovalo. To ovšem znamená, že s frontou stránek se velmi intenzívně pracuje – změna nastává při každém přístupu do paměti.

  Nemusíme řešit problémy s velkými čísly, ale výpočetní náročnost spočívá ve velmi častém provádění operací s frontou.

Tento algoritmus je sice poměrně optimální (spravedlivý), ale všechny tři popsané implementace jsou příliš výpočetně náročné a metoda bohužel postihuje nejvíc čerstvě spuštěné procesy (jejich stránky dosud nebyly hodně používány).

Porovnání s předchozí metodou: zatímco FIFO hledá oběť podle toho, kdy se stránka dostala do paměti, LRU hledá oběť podle toho, kdy se stránka naposledy používala.

Least Frequently Used (LFU). Na obrázku 3.5 na straně 35 jste si možná všimli, že součástí záznamu PTE pro stránku je také bit označený „Accessed“. Tento bit je nastaven na 1 při každém přístupu na stránku (bez ohledu na to, jaká byla jeho původní hodnota).

Dále se pro každou stránku vede čítač, který se v pravidelných intervalech zvyšuje o hodnotu bitu „Accessed“ a zároveň je tento bit vynulován. Pokud tedy v proběhlém intervalu proces tuto stránku použil, zvedne se hodnota čítače o 1. Pokud ne, pak se hodnota čítače nezmění. Pokud je třeba najít oběť pro odložení stránky do odkládacího prostoru, hledá se stránka s nejnižší hodnotou čítače.

Oproti předchozí metodě je přesnost mírně menší (odkládá se nejméně používaná stránka), ale zato je výpočetně mnohem méně náročná. Ostatní výhody a nevýhody zůstávají.

Not Used Recently (NUR). Tento algoritmus je upřesněním předchozího. Místo jediného bitu se pro každou stránku používají dva bity:

- „Accessed“ bit (také „Referenced“) – nastaven na 1 při každém přístupu na stránku,
- „Modified“ bit – nastaven na 1 při každé změně obsahu stránky, tedy při přístupu pro zápis.

Při přístupu pro zápis se tedy nastavují oba bity, při přístupu pro čtení pouze ten druhý. Taktéž se pracuje v intervalech, přičemž na začátku každého intervalu jsou všechny bity vynulovány.

Pokud je třeba najít oběť pro odsun do odkládacího prostoru, nejdřív se hledá stránka, jejíž „Accessed“ bit je nulový, tedy v daném intervalu na ni nebylo vůbec přistupováno. Jestliže žádná taková stránka není, hledá se stránka, která má první bit nastavený, ale druhý ne (tj. přistupovalo se k ní pouze pro čtení).

Tento algoritmus je poměrně blízko optimálnímu, ale je použitelný jen na hardwarových architekturách, které v PTE záznamech pro stránky používají oba uvedené bity. Architektury procesorů Intel a AMD používají pouze „Accessed“ bit, tedy tento algoritmus na nich nelze použít.

Na architekturách procesorů Intelu a AMD máme tzv. „Dirty“ bit, který by zdánlivě mohl sloužit k podobnému účelu, ale není to pravda (navzdory tomu, že na některých webech výrobců hardwaru to je psáno). Dirty bit je nastaven tehdy, když se stránka nachází v cache paměti, ale tam na ní byla provedena změna, která zatím neproběhla přímo v operační paměti. Dirty bit tedy značí, že je nutné synchronizovat cache s operační pamětí.

Hodinový algoritmus (Clock Algorithm). Tento algoritmus je hybrid mezi FIFO a LRU a optimalitou se blíží algoritmu NUR. Používá se buď přímo „Accessed“ bit (kterému se v tomto algoritmu říká „Used“ bit), nebo se nahrazuje vícebitovou hodnotou ukládanou v evidenci stránek.

Podobně jako u FIFO, i zde máme frontu stránek s přiřazeným rámcem v paměti, ale zde jde o kruhovou frontu. Správce paměti používá ukazatel do této fronty (vždy jedna stránka je „vybrána“). Zároveň pro každou stránku v této kruhové frontě se používá „Used“ bit stejně jak bylo výše popsáno pro „Accessed“ bit, tedy při každém přístupu na stránku se nastaví na 1.

Když je třeba najít oběť, je zkontrolován Used bit pro stránku právě označenou ukazatelem.

- Pokud je bit =0, pak je tato stránka odsunuta a ukazatel se posune na další stránku v kruhové frontě.
- Pokud je bit =1, pak je tento bit vynulován, ukazatel se posune na další stránku a algoritmus přechází k předchozímu bodu tohoto postupu.

Při hledání oběti se tedy postupuje v kruhu jako u hodin, zmíněný ukazatel si můžeme představit jako hodinovou ručičku. Hodinová ručička se zastaví až u stránky, jejíž Used bit je =0, tuto stránku odsune a uvolní tak místo.

V současných operačních systémech se používá obdoba poslední popsané metody (hodinový algoritmus), ovšem upravená, aby fungovala poněkud optimálněji. Nepoužívají se časová razítka, ani jeden bit, ale pro každou stránku několik bitů.

3.3.3 Segmentace se stránkováním na žádost

 Současné operační systémy používají virtuální paměť a kombinují segmentaci se stránkováním. Proces má přiděleno několik segmentů, každý segment se může rozkládat na několika stránkách. Oproti předchozí metodě tedy není na stránky dělen adresový prostor procesu jako celek, ale zvlášť jeho jednotlivé segmenty.

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 je zachyceno, zda má konkrétní stránka přidělen rámec ve vnitřní paměti nebo je odložena na disku.

Také zde je možné sdílet segmenty, jejichž délka ani obsah se nemění (obvykle se však spíše mluví o sdílení stránek, takové sdílení je jednodušší a obecnější). 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.

3.3.4 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 svůj adresový prostor 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.

Tato metoda se původně používala u starých UNIXových systémů na hardwarových architekturách bez podpory stránkování.

3.4 Technologie

3.4.1 Copy-on-Write

 *Mechanismus copy-on-write* se používá ve všech současných operačních systémech (do Windows se dostal z UNIXu) a jde o metodu umísťování požadovaného sdíleného souboru (často dynamicky linkované knihovny) nebo obecně sdíleného místa do paměti. Pro zjednodušení budeme dále psát o

objektu v paměti.

V případě, že tentýž objekt používá více procesů, ale pouze pro čtení, je tento objekt fyzicky v paměti pouze jednou, třebaže se do adresového prostoru jednotlivých procesů mapuje pod různými logickými adresami. Ovšem v okamžiku, kdy některý proces požádá o přístup pro zápis, je ve fyzické paměti vytvořena nová kopie objektu a je přidělena procesu s těmito „speciálními“ požadavky. Nová kopie (Copy) se vytvoří až v okamžiku přístupu pro zápis (on Write).

3.4.2 Adresový prostor a virtuální paměť

Část adresového prostoru obvykle zabírá samotný operační systém, jsou to většinou adresy *na konci adresového prostoru* a jsou systému napevno přiděleny. 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 sem mohou pouze v režimu čtení (podle nastavení přístupových oprávnění).

Ve Windows i v Linuxu se adresový prostor dělí na část, do které může proces jakkoliv zasahovat (nižší adresy), a na část, do které nemůže zasahovat, ale pouze z ní číst (vyšší adresy – zde jsou systémové soubory a knihovny, které proces potřebuje ke svému běhu, pouze pro čtení). U 32bitového systému má proces celkem k dispozici 4 GB paměti (ve skutečnosti o něco méně), z toho spodní 2 nebo 3 GB jsou procesu plně k dispozici.

Ve Windows i v mnoha UNIXových systémech včetně Linuxu se používá virtuální paměť, a to metoda *segmentace se stránkováním na žádost*. Z důvodu snadnější údržby virtuální paměti, odstranění nutnosti mít celou tabulku stránek v paměti a také z důvodu urychlení přístupu k ní se používají *víceúrovňové struktury stránek* (obvykle 2–4úrovňové stránkování), jak bylo popsáno výše.

Víme, že množství adresovatelného prostoru je omezeno délkou adresy. Například 32bitový systém používá pro uložení adresy pouze 32 bitů (4 B) a horní hranice je proto přibližně 4 GB. UNIXové systémy (a dokonce i některé verze Windows) však dovolují používat také prostor nad touto hranicí (u 64bitového systému to nemá moc význam, tam je standardně adresovaný prostor dostatečně rozsáhlý, ale u 32bitového systému to stojí za úvahu). Jde o *mechanismus PAE* (Physical Address Extensions). Zpřístupnění funguje dočasným mapováním jedné z takovýchto oblastí za horní hranicí (obvykle je jich více) do adresovatelné části rozsahu. Do celkem malého adresovatelného prostoru si namapujeme vždy tu část „neadresovatelného“ prostoru, se kterou právě chceme pracovat.

U současných procesorů se používá ochrana proti spouštění kódu na paměťových stránkách s daty. U procesorů AMD jde o *NX bit* (Non-Executive), u Intelu *XD bit* (Execute Disable). Operační systém tento bit eviduje v běžné evidenci stránek, vždy při přístupu na stránku se obsah tohoto bitu uloží do registru procesoru. Pokud má stránka nastaven tento bit na 1, je považována za datovou a při pokusu o zacházení s obsahem stránky jako s kódem (spuštění instrukcí zde uložených) je vyvolána výjimka, která většinou končí ukončením procesu, který se o to pokusil. Účelem je zabránit útokům typu přetečení paměti.

3.4.3 Garbage Collection

Garbage Collection je forma dynamické správy paměti používaná zejména v běhových prostředích či podsystémech. Nejznámější garbage collectory jsou v běhových prostředích pro Javu (Java RunTime

Environment, JRE), dále .NET (tedy .NET Framework), podobnou komponentu používáme při interpretaci kódu v SmallTalku, Haskellu, Ruby, ale také v prostředích pro C++.

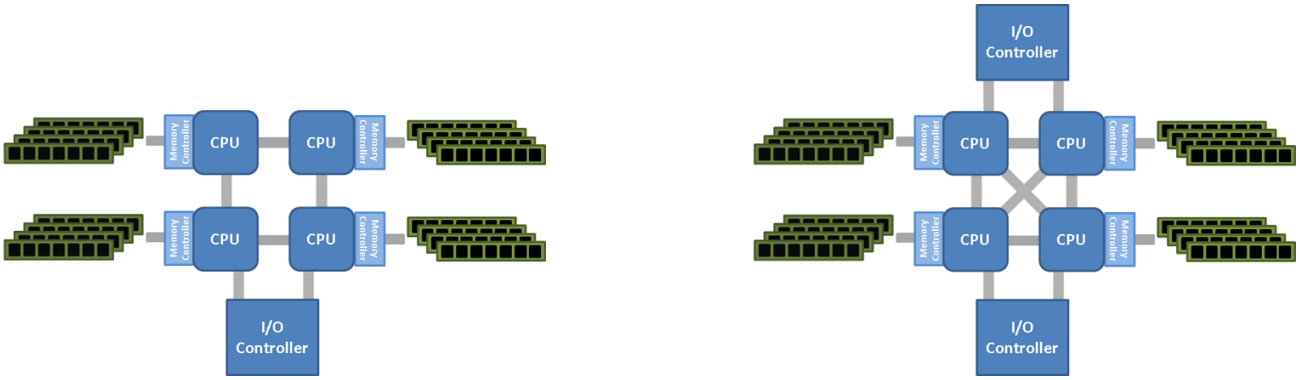
Hlavním úkolem garbage collectoru je dynamická správa objektů v paměti – v okamžiku, kdy je objekt vytvořen, je pro něj automaticky vyhrazena paměť, a když už tento objekt používán nebude, je tato paměť automaticky uvolněna. Programátor se o nic z toho nemusí osobně starat.

Pro garbage collection je tedy důležité hlavně to, jak vyhledávat nepoužívané sekce (objekty) v paměti. Pro to existuje celá řada metod, nejpoužívanější jsou tyto:

- *Reference Counting* (počítání odkazů): každý objekt má čítač (counter), ve kterém se eviduje počet odkazů vedoucích na tento objekt. Pokud hodnota čítače klesne na 0, je objekt odstraněn.
- *Mark-and-Sweep algorithm* (označ a zameť): každý objekt má ve svých vlastnostech položku „visited“. Správce paměti v pravidelných intervalech provádí tento algoritmus:
  1. U všech objektů nastaví hodnotu „visited“ na false.
  2. Prochází celou paměť a hledá odkazy na objekty. Pokud najde takový odkaz, pak u cílového objektu nastaví hodnotu „visited“ na true.
  3. Nakonec projde všechny objekty a zkontroluje hodnotu „visited“. Pokud je tato hodnota false, pak to znamená, že na daný objekt už nevede žádný odkaz, tento objekt je odstraněn.

3.4.4 NUMA architektura

Jak Windows, tak i prakticky všechny známější UNIXové systémy využívají na víceprocesorovém systému symetrický multiprocessing (SMP). To je z většiny hledisek výhoda, ale jednu nevýhodu SMP přece jen má: sdílené fyzické adresy v paměti. To, že všechny procesory sdílejí stejnou paměť a přenášejí data po společné paměťové sběrnici, může být úzkým hrdlem systému.



Obrázek 3.7: NUMA architektura pro procesory AMD (sběrnice HT) a Intel (sběrnice (QPI)[2]

Řešením je, opět v různých operačních systémech, podpora NUMA (podpora musí být samozřejmě i na straně hardwaru). *NUMA* (Non-Uniform Memory Access) je architektura, která dělí celý systém na uzly (nodes, také zones), což jsou relativně samostatné části. Každý node má přidělen jeden nebo více procesorů, část operační paměti a vlastní paměťovou sběrnici. Každý procesor primárně používá to, co je k dispozici v jeho nodu, včetně paměťové sběrnice.

[2]Zdroj: https://www.sqlskills.com/blogs/jonathan/understanding-non-uniform-memory-accessarchitectures-numa/

Vedle toho existuje také hlavní paměťová sběrnice (superior memory bus, interconnection bus) pro komunikaci mezi uzly, tato komunikace je však pomalejší. Účelem je tedy vymezení paměti pro velmi rychlý přístup za cenu rychlostního omezení přístupu ke zbytku paměti.

Vedlejším příjemným efektem je, že paměťové prostory různých uzlů jsou adresovány nezávisle, tedy nám vyjdou adresy pro mnohem víc paměti než bez NUMA.

3.4.5 Little a Big Endian

Předpokládejme, že ukládáme velké číslo (více oktetů) do paměti. V jakém pořadí se jednotlivé oktety celého datového typu ukládají?

- Little Endian – nejnižší oktet se zapisuje na nižší adresu – Intel, DEC Alpha
- Big Endian – nejvyšší oktet se zapisuje na nižší adresu – Motorola, SPARC
- Mixed Endian – pomíchaně – PDP

PowerPC zvládá obojí. Rozlišuje dva módy: big a little.

Příklad

Rozdíl mezi Little a Big Endian si ukážeme na příkladu. Na adresu 400 (hexadecimálně) uložíme číslo o délce 4 oktetů (postupujeme přes registr EAX, protože instrukce mov obvykle vyžaduje, aby alespoň jeden parametr byl registr) a podíváme se, v jakém pořadí jsou jednotlivé oktety uloženy.

```
mov EAX, 1234ABCDh
mov [400h], EAX         ; EAX je 32bitový datový registr
```

Little Endian:

- adresa 400h = CDh
- adresa 401h = ABh
- adresa 402h = 34h
- adresa 403h = 12h

	400	401	402	403	
...	CD	AB	34	12	...

Big Endian:

- adresa 400h = 12h
- adresa 401h = 34h
- adresa 402h = ABh
- adresa 403h = CDh

	400	401	402	403	
...	12	34	AB	CD	...

Všechna uvedená čísla jsou v šestnáctkové soustavě.

Problémy s Endians mohou nastat například při komunikaci s jiným počítačem v síti, který používá jiné pořadí, při komunikaci se zařízením, které používá jiné pořadí (ovladače).

Řešení v UNIXových systémech:

- „ruční“ konverze,
- většina zařízení je Little Endian, I/O funkce operačních systémů s tím počítají a Big Endian systém automaticky provádí konverze,
- v UNIXových systémech existují Big Endian varianty I/O funkcí pro případ komunikace s takovými zařízeními.

3.5 Správa paměti v některých operačních systémech

3.5.1 MS-DOS

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[3]) 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), 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.

3.5.2 Windows

Windows v chráněném režimu používají segmenty, stránky (segmentaci se stránkováním na žádost), deskriptory, selektory tak, jak bylo výše popsáno.

Většina paměti procesu je *stránkovaná* (paged), tj. může být odkládána, ale procesy (a především jádro) mají také *nestránkovanou paměť* (nonpaged) používanou časově kritickými funkcemi (například obsluha IRQ nebo volání DPC, podrobněji v kapitolách o komunikaci procesů a správě zařízení).

Virtualizace paměti. Windows používají virtuální metodu *segmentace se stránkováním na žádost*. Každý proces má přiděleny své segmenty a ty jsou rozděleny na stránky. Při potřebě uvolnění rámců v operační paměti se pro výběr „oběti“ používá na jednoprocesorových systémech hodinový algoritmus, na víceprocesorových systémech v kombinaci s algoritmem FIFO.

[3] 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.

Stránkovací soubor se obvykle jmenuje `pagefile.sys` a je na systémovém disku, ale ve skutečnosti můžeme mít více stránkovacích souborů (v tom případě by však každý měl být na jiném pevném disku). Názvy stránkovacích souborů najdeme v klíči *HKLM/SYSTEM/CurrentControlSet/Control/Session Manager/Memory Management*, a to v položce typu pole řetězců *PagingFiles*.

Pokud máme více stránkovacích souborů, měl by být každý na jiném *fyzickém* disku (pokud je jich víc na stejném fyzickém disku, třeba i na jiných oddílech, může to systém dokonce zpomalit).

Umístění stránkovacího souboru do samostatného oddílu na disku (odděleně od oddílů se systémem a daty) může být užitečné například proto, že takto nehrozí fragmentace stránkovacího souboru (ale na druhou stranu tady máme horní ohraničení paměťové oblasti, která může být pro stránkování využita).

V tomtéž klíči je ještě jedna zajímavá položka – *ClearPageFileAtShutdown*. Pokud existuje a je nastavená na 1, před každým (regulérním) vypnutím systému se smaže stránkovací soubor. To je důležité z hlediska bezpečnosti, protože v stránkovacím souboru se mohou nacházet důležité informace, které by se neměly dostat do nepovolaných rukou (jistá verze Windows takto „zveřejnila“ nezašifrované heslo administrátora).

Možnost nastavit umístění odkládacího souboru v systémech s NT jádrem je výhodná v případě, že máme nainstalováno více systémů rodiny Windows a chceme, aby používaly tentýž odkládací soubor, nebo v případě, že chceme rozprostřít stránkování mezi více disků.

Sdílení paměti. V 16bitových 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 32bitových Windows byly již tyto aktivity omezeny. Pokud proces požádá o přístup k dynamicky linkované knihovně (nebo jakémukoliv jinému souboru), je obsah souboru nejdřív zpřístupněn pro čtení, ale při pokusu o zápis je *namapován* do adresového prostoru tohoto procesu (metodou copy-on-write), tedy proces zapisuje do své vlastní soukromé kopie. Tak je soubor v paměti přítomen i vícekrát.

Jako sdílenou paměť lze však použít objekty exekutivy k tomu určené (brali jsme na cvičeních předmětu Operační systémy II), které na rozdíl od knihoven jsou transparentnější a poskytují lepší možnosti nastavení přístupových oprávnění.

3.5.3 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 segmentací i ochrany paměti, ale způsob odkládání zůstal dlouho podobný – odložen byl vždy paměťový prostor celého odkládaného procesu, ne pouze jedna stránka, tedy byla používána virtuální metoda swapování procesů. Přesto byly používány i segmenty, mohou být sdíleny.

Virtualizace paměti. Téměř všechny dnešní větší UNIXové systémy včetně Linuxu zvolily jinou

formu virtualizace paměti – stránkování na žádost nebo *stránkování se segmentací na žádost* (to je také případ většiny linuxových distribucí). Používá se víceúrovňové stránkování, na 64bitovém systému jde o 3–4 úrovně.

Každý proces má čtyři *segmenty* – segment kódu, segment pro globální data, zásobník pro uživatelský režim, zásobník pro režim jádra. Globálně alokovaná data a zásobník jsou na adresách „proti sobě“ (zásobník roste směrem k datovému segmentu).

Přestože již nejde o swapování, i nadále se používá pojem swapovací soubor nebo swapovací oblast na disku.

Výběr oběti je prováděn pomocí *modifikace hodinového algoritmu*, s ohledem na víceúrovňové stránkování. Indikace „starých“ stránek není pouze dvouhodnotová, ale pohybuje se v intervalu 0...20, kdy 0 znamená „starou“ stránku velmi vhodnou k odložení. Při každém přístupu na stránku je tato hodnota zvýšena (ve výchozím nastavení o 3 až k maximu), správce paměti neustále (hodinový algoritmus) prochází všechny tyto hodnoty a snižuje (ve výchozím nastavení o 1).

Postup

Můžeme mít víc odkládacích souborů (plus odkládací oddíl). Další odkládací soubor vytvoříme takto:

- vytvoříme souvislý soubor na disku naplněný symboly `#0` (ne nulami!),
- označíme ho jako swapovací (odkládací) soubor,
- zapneme swapování do tohoto souboru (můžeme kdykoliv vypnout).

Například pokud chceme vytvořit odkládací soubor o velikosti 65536 stránek po 4 kB (což je 4096 B):

```
dd if=/dev/zero of=/novy_swap bs=4096 count=65536
mkswap /novy_swap
swapon /novy_swap
```

Také bychom měli tento soubor zapsat do souboru `/etc/fstab`:

```
/novy_swap none swap sw 0 0
```

Sdílení a mapování. Tyto systémy podporují několik zajímavých prvků správy paměti, například vedle nám již známého *copy-on-write* se setkáme s mechanismem *sdílení kódu programů*: pokud je spuštěno více procesů – instancí jednoho programu, mohou sdílet část paměti, ve které je načten kód programu.

Funkce *mapování souborů* funguje tak, že do adresového prostoru procesu může být namapován kterýkoliv soubor, ať už se fyzicky nachází kdekoliv (nemusí být v operační paměti). Pro mapování obvykle máme jeden ze dvou důvodů:

- chceme, aby bylo možné se souborem přímo na disku (obecně vnějším paměťovém médiu) pracovat stejně jako kdyby byl načten do operační paměti (přesněji chceme se souborem pracovat jako s operační pamětí, samozřejmě až na rychlost), ale ve skutečnosti tento soubor v operační paměti nechceme (například z důvodu značné velikosti souboru), může jít také o spustitelný kód rozsáhlejšího programu (tedy segment kódu by zůstal na disku),
- implementace *sdílené paměti* jakékoliv velikosti – sdílená paměť přímo v operační paměti je bezpečnostním rizikem, proto je mnohem jednodušší a bezpečnější vytvořit (simulovaný) sdílený úsek paměti přístupný více procesům na vnějším paměťovém médiu.

Mechanismus mapování je velmi univerzální, ve skutečnosti se používá například i při práci s odkládacím prostorem.

Většina UNIXových systémů 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).

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

Adresový prostor. Každý proces má přidělen *deskriptor paměti* (jeden nebo více), což je struktura obsahující veškeré informace o virtuální paměti přidělené procesu a také související funkce (například přístup na strukturu usnadňující vyhledávání ve stránkách, počet namapovaných oblastí, adresa první namapované oblasti, celková velikost virtuální paměti namapované procesu, synchronizační objekty související s pamětí, funkce pro odmapování oblasti, atd.)

Deskriptor paměti je dostupný v *deskriptoru procesu*.[4]

Další informace:

O principu správy paměti v Linuxu se dočteme například na http://www.makelinux.net/reference.

[4] V deskriptoru procesu najdeme především struktury a odkazy související s přidělenými prostředky a komunikačními nástroji, například ukazatele na deskriptory souborů, ukazatele na deskriptory paměti, momentální pracovní adresář procesu, terminál, na kterém proces běží, strukturu pro evidenci signálů, atd.

Kapitola 4

Procesy

Rychlý náhled: 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.

Klíčová slova: proces, program, vlákno, PCB (Process Control Block), multitasking, kontext, multithreading, IPC (Inter-Process Communication), plánování procesoru, synchronizace, Petriho sítě, kritická sekce, producent-konzument, semafor, mutex

Cíle studia: Po prostudování této kapitoly porozumíte problematice správy procesů v operačním systému. Pochopíte způsob evidence procesů v systému, princip sdílení výpočetních prostředků formou multitaskingu a multithreadingu a plánování běhu procesů na procesoru, možnosti komunikace mezi procesy a také synchronizaci přístupu ke sdíleným prostředkům.

4.1 Evidence procesů

4.1.1 Pojmy proces a úloha

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. Zjednodušeně se dá říct, že proces je běžící program, ale to není úplně přesné (proces nemusí nutně vzniknout z programu, i když většinou tomu tak bývá, přesnější by bylo například „spuštění z kódu“).

Každý proces má svůj identifikátor – PID (Process ID). Dále je u procesu evidováno PID jeho rodičovského procesu – PPID (Parent PID).

Vlákno je relativně samostatná část procesu vykonávající svůj vlastní kód (vlákno obvykle vykonává některou funkci procesu). Proces má vždy nejméně jedno (hlavní) vlákno, které vykonává jeho hlavní funkci (main), a dále může programátor rozdělit běh procesu na více vláken, která pak mohou být prováděna navzájem nezávisle.

V současných operačních systémech je procesor přidělován přímo vláknům, nikoliv procesům. Každé vlákno by tedy mělo být jednoznačně identifikováno, k tomu slouží TID (Thread ID). Tato čísla bývají

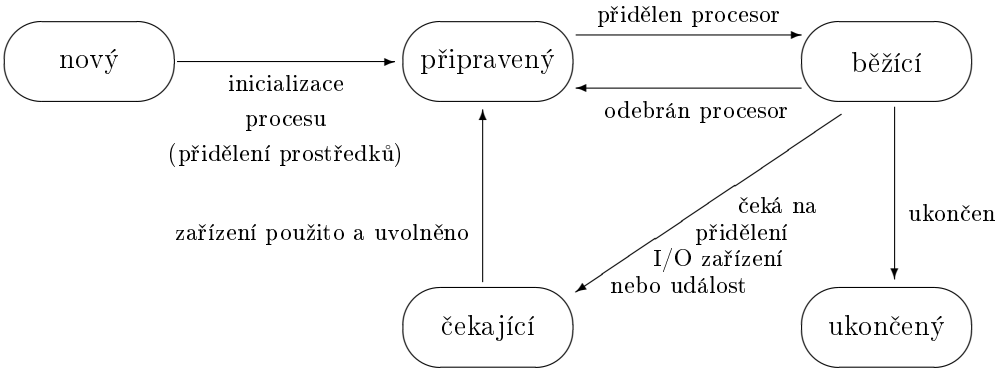
jednoznačná nejen v rámci procesu, ale i v rámci celého systému, protože (jak bylo výše uvedeno) vlákna komunikují s jádrem většinou samostatně bez zprostředkování procesem.

Pokud proces vznikl spuštěním z binárního spustitelného souboru, pak tento soubor nazýváme *obrazem procesu*. Obraz je tedy soubor, z něhož proces získal kód, který provádí. Jestliže spouštíme textový spustitelný soubor (skript), obrazem je ve skutečnosti jeho interpretační program (například Příkazový řádek ve Windows nebo některý shell v UNIXových systémech, případně Python, Pearl apod.).

Proces se 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/blokovaný* (waiting/blocked) – čeká na přístup k I/O prostředku, o který požádal, například na otevření vyžádaného souboru,
- *ukončený* (terminated, zastavený) – proces byl ukončen, ale jeho datové struktury ještě existují.

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



Obrázek 4.1: Přechody mezi stavy procesu

V rámci některých operačních systémů mohou být definovány i další stavy. Například v UNIXových systémech může být proces navíc v jednom z těchto stavů:

- *pozastavený* – provádění programu je pozastaveno, což může být požadavkem debuggeru (signál SIGTRAP) nebo uživatele či systému (SIGSTOP),
- *zombie* – program byl vpodstatě ukončen, už nemá žádný kód k provádění a nedostává procesor, ale jeho prostředky (včetně PID) ještě nebyly uvolněny, to se používá například tehdy, když si rodičovský proces tohoto procesu explicitně vyžádal tento typ ukončení, aby měl dost času na „vyzvednutí“ výsledků činnosti tohoto svého potomka (rodičovský proces byl informován signálem SIGCHILD, ale nereaguje),
- *uspaný* (sleeping) – proces (častěji vlákno) čeká na splnění podmínky, například uplynutí časového intervalu nebo některou událost.

Ve Windows se proces může dostat do stavu *suspend*, pokud jsou stránky jeho paměti odloženy do odkládacího prostoru a proces nemůže provádět svůj kód. Do tohoto stavu se proces většinou dostává ze stavu *blocked* (ve kterém čeká na přidělení některého prostředku).

Pojem *úloha* bývá vysvětlován různě (také v různých operačních systémech). Existují například také tiskové úlohy nebo úlohy plánované pro spuštění, zde však budeme hovořit spíše o úlohách běžících na procesoru, tj. vzniklých spuštěním kódu (jinými slovy, procesor plánuje úlohy, což jsou obvykle vlákna).

Ve Windows se pojem úloha v tomto smyslu moc nepoužívá, ale v UNIXových systémech je běžný a velmi důležitý. Úloha je zde objektem, jehož kód se sekvenčně vykonává. Bývá to obvykle jedno vlákno aplikace, ale v případě vláken, jejichž kód je sekvenčně propojen (například přes rouru) mohou být součástí téže úlohy i další vlákna či procesy (sekvenčně na sebe navazující).

4.1.2 Soubory s kódem

Každý binární soubor má svůj předepsaný formát, a týká se to také binárních spustitelných souborů a knihoven. Obvykle má jedno nebo více záhlaví se specifickými informacemi, například použitý formát, použité instrukční sady, odkazy na knihovny, které jsou v kódu vyžadovány (mají být dynamicky přilinkovány), atd., tedy vše, co potřebuje správce procesů znát při spouštění procesu z tohoto souboru.

Formáty ve Windows. Spustitelné soubory a knihovny ve Windows bývají v jednom z těchto formátů:

- PE (Portable Executable) je 32bitový formát pro Windows,
- PE+ je 64bitový formát pro Windows,
- .NET PE je určen pro .NET aplikace,
- UWP (Universal Windows Platform) a jeho nástupce WinUI 3 (Windows Apps SDK) je formát „univerzálních“ aplikací z MS Store, tyto aplikace taktéž patří do rodiny PE formátů a jsou 64bitové.

Pokud vidíme soubor s příponou .EXE nebo .DLL, na první pohled nepoznáme, v jakém je formátu – může to být kterýkoliv z výše uvedených.

Co se týče PE formátů, podle závislostí rozlišujeme

- *PE* vyžadující běh v subsystému Win32 (před spuštěním musí být funkční subsystém Win32 /Windows spuštěný souborem `csrss.exe`),
- nativní PE nevyžadující předem spuštěný `csrss.exe`, například `smss.exe` nebo samotný subsystém Win32 (proces `csrss.exe`).

Naprostá většina binárních spustitelných souborů a knihoven ve Windows je prvního typu (vyžadují běh subsytému Win32/Windows).

Formáty v Linuxu. Formát *ELF* (Executable and Linkable Format) je v současné době nejběžnějším formátem pro tyto účely v Linuxu, dalším obvyklým (starším a už méně častým) formátem je *a.out*. Spustitelné soubory ve formátu ELF obvykle nemají žádnou příponu.

Formát ELF se používá také u knihoven (obvykle mají příponu .SO) a modulů jádra (přípona (.KO, tedy „kernel object“).

Poznámka:

Pro správnou identifikaci formátu není dobré spoléhat na příponu souboru, ostatně v UNIXových systémech spustitelné soubory ani žádnou nemají.

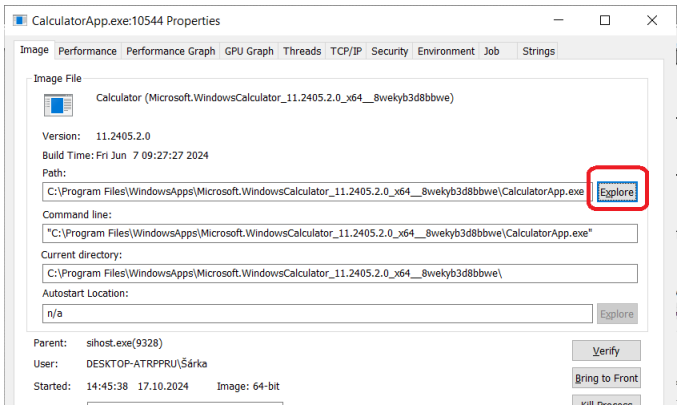
Základní identifikace se dá provést obvykle podle prvních oktetů souboru. U Linuxu je to celkem jednoduché, ELF soubor má v prvních třech oktetech znaky „ELF“. U Windows se v příponě EXE

setkáme se začátkem souboru „PE“, to je PE soubor, dále „MZ“ znamená starý DOS spustitelný soubor nebo naopak nový s formátem WinUI 3, a „NE“ (New Executable) pro starší Windows do verze 3.11 (a všechny mají opravdu stejnou příponu).

Příklad

Ke spustitelnému souboru formátu WinUI 3 je poněkud těžší se dostat, nicméně cesta existuje.

Typické místo instalace je ve složce `Program Files`, podsložce `Windows Apps`. Tam ovšem nemají přístup ani uživatelé s administrátorským účtem. Ale pokud si například v Process Exploreru najdeme takovouto spuštěnou aplikaci (například Kalkulačku nainstalovanou ze Storu), ve vlastnostech na kartě Image je odkaz na umístění (viz obrázek 4.2).



Obrázek 4.2: Vlastnosti aplikace formátu WinUI 3

Z nějakého důvodu se k těmto souborům nedá dostat přes nadřízený adresář, ale touto cestou ano. Takže není problém si ověřit, jakými oktety daný soubor začíná. Ostatně typ souboru je jasný i z umístění ve `Windows Apps`.

U hlavních formátů ve Windows (PE) a v Linuxu (ELF) rozlišujeme

- kód linkovaný *staticky* – při překladu programu, je pak součástí překládaného spustitelného souboru či knihovny, podle toho, co překládáme,
- kód linkovaný *dynamicky* – linkuje se do kódu při každém novém spuštění programu, tento kód je uložen zvlášť v (dynamicky linkovaných) knihovnách (ve Windows většinou .DLL, v Linuxu obvykle .SO, může následovat označení verze knihovny).

4.1.3 Datové struktury související s procesy

Správce procesů vede tabulku procesů (jednu nebo pravděpodobněji několik tabulek, navzájem se na sebe odkazujících), záznam v této tabulce o konkrétním procesu zde budeme pro zjednodušení nazývat *Process Controll Block* (PCB). Je to souhrn všech dat, která operační systém potřebuje k řízení procesu. Součástí PCB 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 různých 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.

Evidence procesů ve Windows řady NT. Používají se tyto datové struktury:

- `EPROCESS` (blok procesu pro potřeby exekutivy) – obsahuje některé vlastnosti procesu (PID, název procesu, stanice oken – viz cvičení předmětu Operační systémy, atd.) a také odkazy na mnoho dalších datových struktur souvisejících s procesem (prakticky k čemukoliv, co se týká procesu, se dá dostat odsud, včetně využívání paměti a bezpečnostních informací), také odkaz na záznam následujícího procesu, nachází se v jádře (oblast exekutivy),
- `ETHREAD` – totéž pro vlákno, bloky `EPROCESS` obsahují odkazy na bloky `ETHREAD` vláken svého procesu,
- `PEB` (Process Environment Block) – nachází se přímo v adresovém prostoru procesu a je dostupný z bloku `EPROCESS`, obsahuje informace, které musejí být dosažitelné v uživatelském režimu (např. adresu haldy, synchronizační informace, seznam knihoven – modulů – procesu, tabulku handlů GDI objektů – viz cvičení předmětu Operační systémy,
- `TEB` (Thread Environment Block) – obdoba pro vlákna,
- `KPROCESS`, také `PCB` – je součástí bloku `EPROCESS`, obsahuje informace potřebné pro plánování procesoru (časové údaje, prioritu, afinitu, stav procesu, kvantum, atd.),
- atd. (o procesu si vede informace také například podsystém Win32, a to zvlášť v uživatelském prostoru a zvlášť v prostoru jádra).

Jak vidíme, jedná se o víceúrovňovou a poměrně složitou evidenci, která obsahuje některá data redundantně.

Evidence procesů v Linuxu. V Linuxu se setkáme s těmito datovými strukturami:

- `task` – jedná se o vektor (pole) ukazatelů na struktury typu `task_struct` pro jednotlivé procesy, a to v režimu jádra,
- `task_struct` je hlavní datová struktura procesu, to, co v předchozím výkladu označujeme jako PCB; obsahuje PID, informace o stavu procesu a také o ukončujícím stavu procesu (zde poznáme, jestli proces skončil, případně, jestli je ve stavu zombie), „rodinné“ informace (ukazatel na svého rodiče, sourozence, potomky), plánování na procesoru, odkazy na struktury související s přidělenou pamětí a dalšími zdroji, kontext procesu,
- další datové struktury, na které vede odkaz z `task_struct`, například ve struktuře `mm_struct` (deskriptor paměti procesu) najdeme vše, co souvisí se správou paměti přidělené procesu.

Příklad

Poněkud zkrácená struktura evidující proces v Linuxu:

```
struct task_struct {
    volatile long state;
    unsigned int flags;

    void *stack;
    int prio, static_prio;
    struct mm_struct *mm, *active_mm;

    pid_t pid;
    struct task_struct *parent;
    struct list_head children;
    struct task_struct *group_leader;
    ...
};
```

Proměnná `state` zachycuje stav procesu (například `TASK_RUNNING`, `TASK_STOPPED`,...). `flags` je číslo, jehož jednotlivé bity určují, co zrovna proces provádí (například bit `PF_STARTING` je nastaven u právě spouštěného procesu, `PF_MEMALLOC` má nastaven takový proces, který zrovna alokuje paměť, atd.). `stack` je ukazatel na právě používaný zásobník. Máme tam taky hodnoty pro statickou a dynamickou prioritu, následují ukazatele na paměť procesu.

Další člen struktury je číslo PID, následuje odkaz na obdobnou stukturu rodičovského procesu a odkazy na struktury potomků.

Další informace:

- https://tampub.uta.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf
- https://www.ibm.com/developerworks/library/l-linux-process-management/index.html
- https://unix.stackexchange.com/questions/80038/what-is-the-structure-of-a-linux-process
- https://www.microsoftpressstore.com/articles/article.aspx?p=2233328
- https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/eprocess

4.1.4 Priority procesů

Každý proces má přiřazenu svou prioritu. Tato priorita je používána k různým účelům, především při plánování přidělování procesoru (proces s vyšší prioritou má přednost před procesem s nižší prioritou).

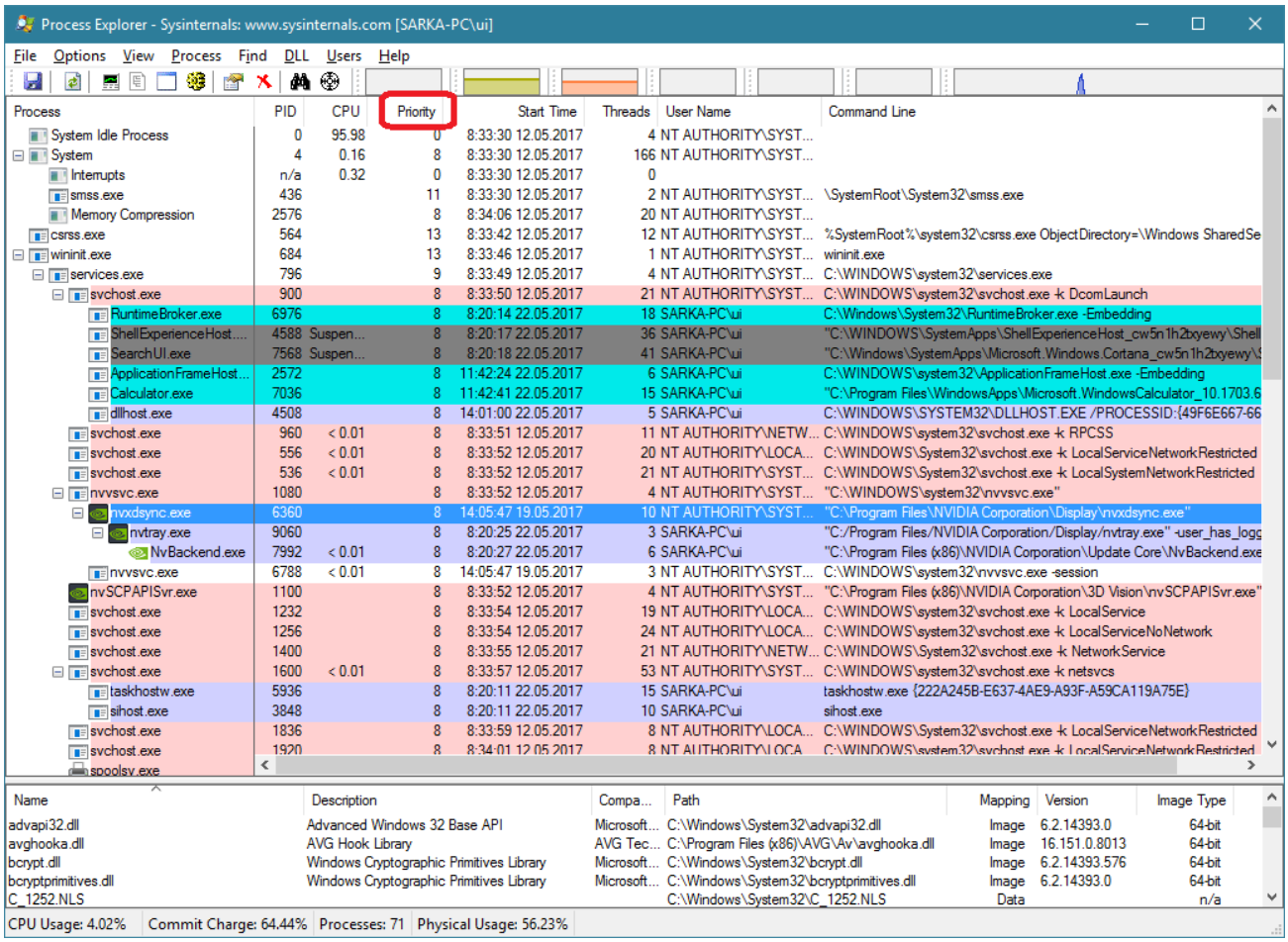
Obvykle rozlišujeme prioritu

- *základní* (base) – proces ji dostane přidělenou při svém vzniku, obvykle se po celou dobu činnosti procesu nemění (vyjma použití k tomu určených příkazů, což není až tak obvyklé),
- *dynamickou* – pohybuje se v úzkém okruhu kolem základní priority, používá se k dočasnému zvýhodňování nebo naopak znevýhodňování procesu v souvislosti s využíváním zdrojů,
- *statickou* – realtimové procesy nepoužívají dynamickou prioritu, jejich priorita se „nehýbe“ a zůstává stejná jako bázová, proto hovoříme o statické prioritě.

Priority ve Windows. Ve Windows je celkové rozmezí 1–31. Úrovně 1–15 jsou dynamické pro běžné procesy, úrovně 16–31 jsou pro procesy reálného času. Existují tyto úrovně priorit:

- 0 – systémová úroveň, používá se pro vlákno nulové stránky
- 1 – dynamická nečinná (Idle)
- kolem 4 – Nečinná (Lowest)
- kolem 6 – Nižší než normální (Below-normal)
- kolem 8 – Normální (Normal)
- kolem 10 – Vyšší než normální (Above-normal)
- kolem 13 (max. 15) – Vysoká priorita (Highest)
- 16 – nečinná reálného času
- kolem 24 – Reálného času (Time-critical)
- 31 – časově kritická reálného času

Uživatelské procesy mají obvykle bázovou prioritu 8 (normální). U systémových procesů včetně procesů pro služby `svchost.exe` je priorita 8 taky celkem běžná, jen některé nejdůležitější procesy (hlavně nativní) mají vyšší – například samotný podsystém Windows `csrss.exe` má typicky prioritu 13 nebo správce relací `smss.exe` má prioritu 11.



Obrázek 4.3: Process Explorer ve Windows 10 – priority

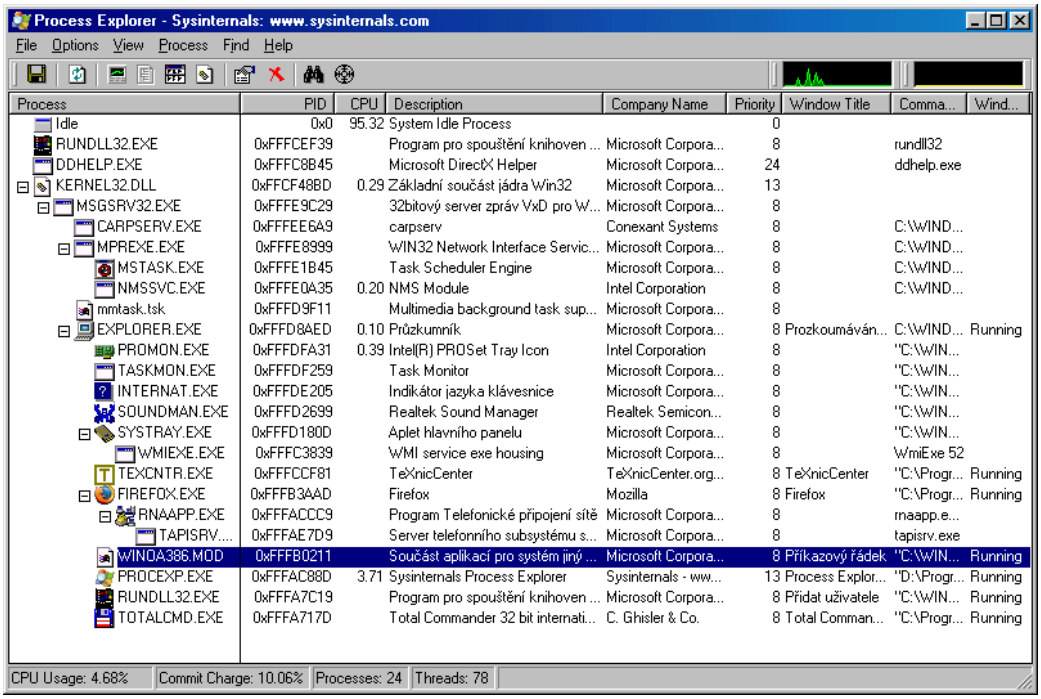
Prioritu procesů můžeme zjistit a ovlivňovat v těchto nástrojích:

1. *Správce úloh* (Task Manager) – na kartě *Procesy* je seznam procesů, sloupec *Základní priorita* (pokud není tento sloupec zobrazen, v menu zvolíme *Zobrazit – Vybrat sloupce*, zatrhneme příslušnou položku), najdeme zde pouze slovní určení priority,
2. *Process Explorer* – zobrazuje také číselnou hodnotu priority, je třeba zobrazit sloupec s prioritou.

V obou těchto nástrojích můžeme změnit prioritu procesu (kontextové menu řádku s procesem), ale pouze mezi „slovními“ úrovněmi.

Na obrázcích 4.3 a 4.4 je Process Explorer spuštěný ve Windows 10 a Windows 98. Na prvním z těchto obrázků vidíme typické hodnoty priorit procesů, jak bylo popsáno v předchozím textu. Na obrázku pro Windows 98 si všimněte sloupce označeného PID. Zde obsažená čísla ve skutečnosti nejsou PID, protože systémy s DOS jádrem PID nepoužívají. Místo něho jsou procesy identifikovány pouze *handlem* (manipulátorem) svého hlavního okna stejně jako kterýkoliv jiný objekt (včetně souborů a oken).

Pro spuštění procesu s konkrétní úrovní priority můžeme využít příkaz `start`. Například pokud chceme spustit proces z obrazu notepad.exe s nízkou prioritou, zadáme



Obrázek 4.4: Process Explorer ve Windows 98

```
start /low notepad.exe
```

Priority v Linuxu. V Linuxu je rozsah priorit 1–40 pro běžné procesy, pro realtimové procesy se používá statická priorita v rozsahu 1–99 (realtimové procesy jsou od běžných více odděleny než ve Windows, mají vlastní algoritmus, plánovač, pro přidělování procesoru). Realtimové procesy mohou být spouštěny pouze superuživatelem (administrátorem). Dále se budeme zabývat pouze běžnými procesy.

Z pohledu uživatele se priority mapují na rozsah „přidat–odebrat“, nazývaný *nice*. Nice je vlastně bázová priorita, dynamická se pohybuje kolem s odchylkou přibližně 5. Hodnoty nice jsou chápány přesně opačně než hodnoty priorit: čím vyšší priorita, tím nižší hodnota nice. Tuto metriku si můžeme představit jako stupeň „příjemnosti“ procesu vzhledem k jiným procesům, tedy proces s vyšší hodnotou nice je k ostatním procesům „příjemnější“, nechává jim více času procesoru, kdežto proces s nižší hodnotou nice je k ostatním procesům méně „příjemný“, více času procesoru si nechává pro sebe.

Nice se vyjadřuje číslem následovně:

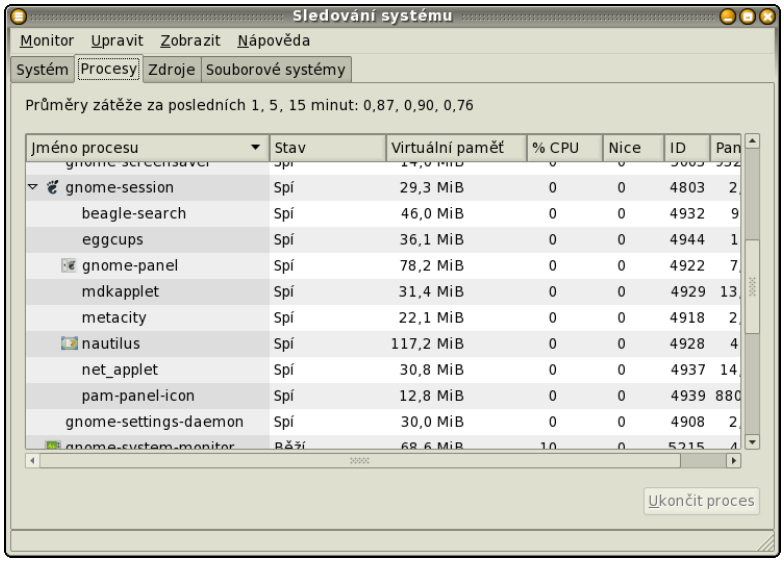
- 0 – běžná priorita,
- 1...19 – zvyšujeme *nice*, proces je „hodnější“ k ostatním procesům, jeho skutečná priorita je snižována,
- $(-20)...(-1)$ – proces je „méně hodný“, tj. jeho priorita se zvyšuje.

Obecně platí, že každý uživatel může snižovat prioritu (zvyšovat hodnotu nice) *svých* procesů, ale zvyšovat prioritu (snižovat nice) smí jen superuživatel (administrátor, obvykle root). Je to logické bezpečnostní opatření pro víceuživatelský systém (běžný uživatel by neměl svévolně ubírat čas procesoru procesům jiných uživatelů nebo dokonce systému).

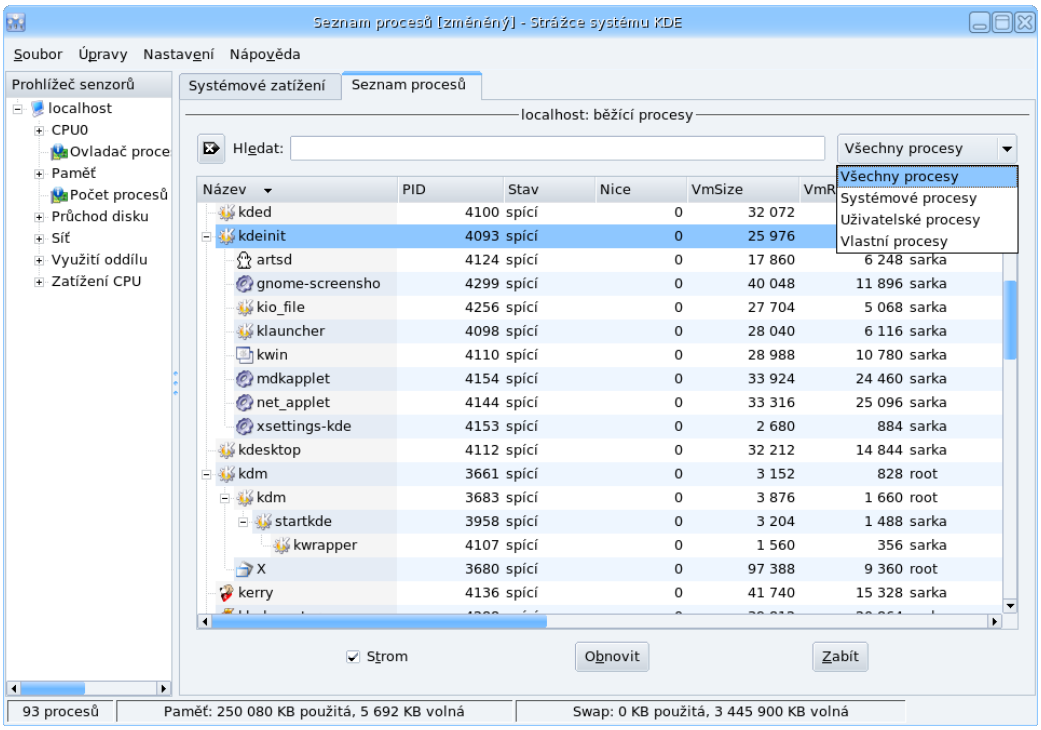
Prioritu procesu v Linuxu lze zjistit v grafickém i textovém prostředí, a to takto:

1. V nástrojích pro grafické rozhraní (podle zvoleného grafického rozhraní). Můžeme použít například program KSysGuard (v prostředí KDE, obrázek 4.6) nebo GNOME System Monitor

(prostředí GNOME, obrázek 4.5), případně se tento nástroj jinak jmenuje, ale binárně je to některý z uvedených.



Obrázek 4.5: Program GNOME System Monitor v prostředí GNOME

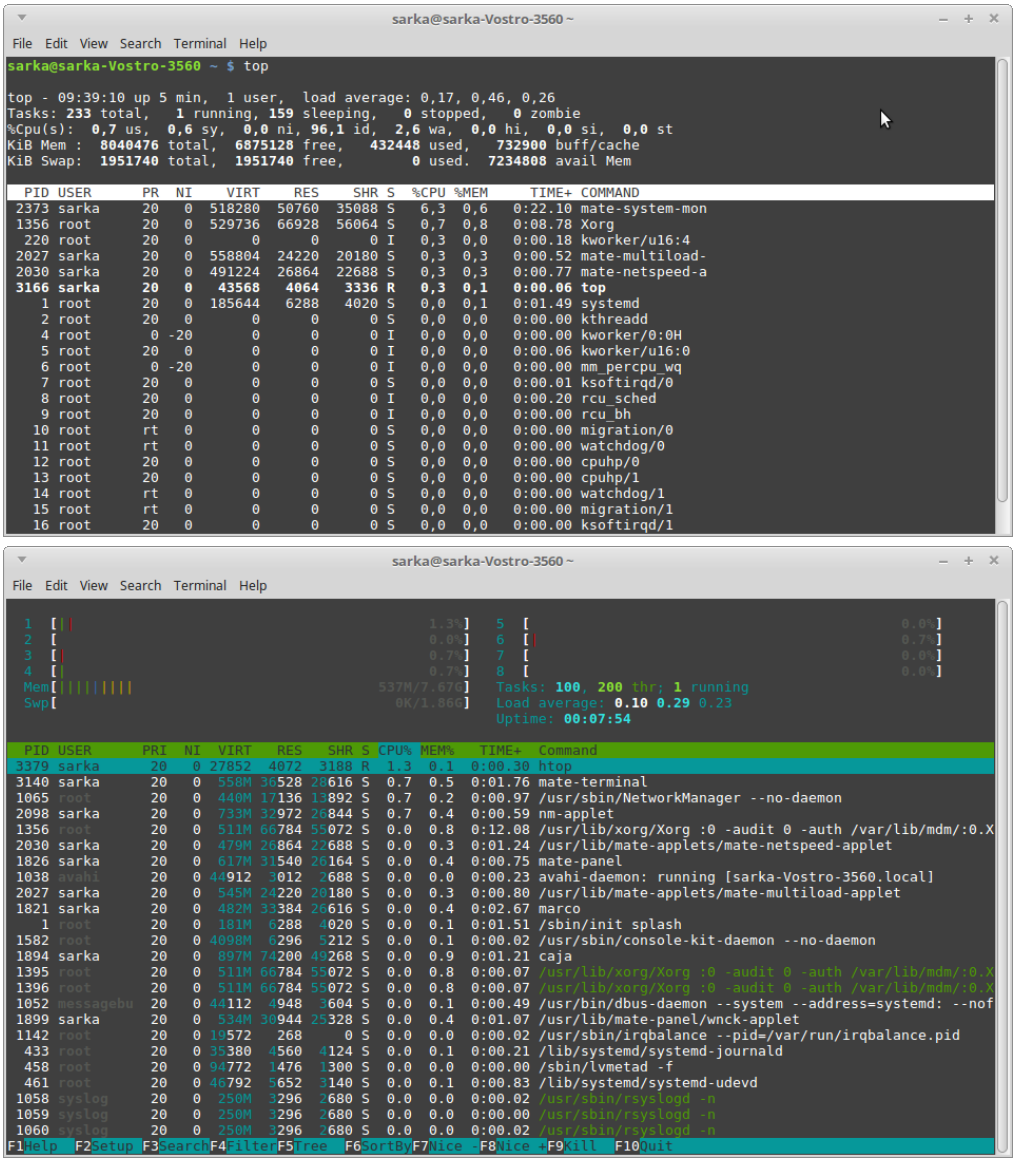


Obrázek 4.6: Program KSysGuard v prostředí KDE

2. Priority si můžeme prohlédnout ve výstupu příkazů zobrazujících seznam spuštěných procesů s různými údaji, kromě jiného i jejich prioritou:

   ```
   top
   ps -le
   ```

   (první je pravidelně se aktualizující seznam, tedy interaktivní proces, druhý jednorázově vypíše požadované údaje), také lze doinstalovat příkaz `htop`. Výstup příkazů `top` a `htop` je na obrázku 4.7.



Obrázek 4.7: Výstup příkazů `top` a `htop`

Prioritu spuštěného procesu můžeme ovlivnit buď v grafickém režimu, anebo v textovém režimu:

- spuštění s danou prioritou (resp. hodnotou nice):

  `nice -n hodnota_nice spouštěná_aplikace`,

  například `nice -n 10 ps`
  nebo `nice -n -10 ps`

  (zvyšujeme nebo snižujeme prioritu procesu `ps`, a to o hodnotu 10)

- změna priority již spuštěné aplikace:

  `renice hodnota_nice_se_znaménkem PID_procesu`,

  například `renice +10 512`

  (prioritu již běžícího procesu s PID 512 jsme snížili, přidali jsme „nice“, o 10)

Mechanismus „nice“ pro práci s prioritami je používán ve většině UNIXových systémů.

4.1.5 Vznik a zánik procesu

Proces bývá většinou spuštěn jiným procesem, čímž je určen vztah rodič-potomek mezi procesy. Sice to vypadá, že uživatelské procesy jsou spouštěny uživatelem, ale ve skutečnosti bývá rodičem některý proces běžící v relaci daného uživatele.

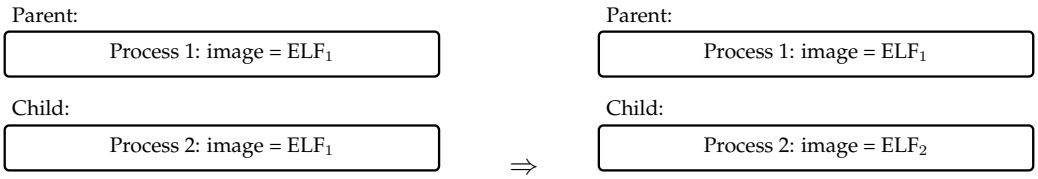
Ve Windows je nový proces (potomek) spuštěn systémovým voláním `CreateProcess()`, přesněji některou z variant této funkce. Argumenty této funkce upřesňují vlastnosti nového procesu, včetně příkazu s paramery, priority, bezpečnostních atributů apod. Tato funkce také zkontroluje, jestli obraz budoucího procesu je 64bitový PE, a pokud ne, spustí také příslušné běhové prostředí (WoW64, ntvdm apod.).

Další informace:

Funkce související s procesy a vlákny jsou popsány například zde:
https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/

V UNIXových systémech je nový proces vytvořen kombinací dvou systémových volání:

- `fork()` vytvoří kopii původního procesu, nový proces má (zatím) stejný kód, data, dokonce je v kódu na stejném místě, nový proces má ovšem jiné PID a také svůj vlastní záznam PCB,
- `execve()` nebo některá jeho varianta volaná v novém procesu vymění původní rodičovský kód za kód jiného spustitelného souboru (je jedním z argumentů této funkce) a restartuje ukazatel do segmentu s kódem (tedy nastaví provádění kódu na začátek, první instrukci hlavní funkce).



Obrázek 4.8: Obraz rodiče a potomka před a po zavolání funkce `execve()`

Příklad

Spuštění nového procesu v Linuxu může probíhat takto:

```
// rozděl mě (tento proces) na dva téměř stejné (až na PID):
pid_t child_pid = fork();
if (child_pid == 0) {   // hodnota 0 znamená, že jsme v novém procesu
  execve(...);  // v potomkovi nahraď původní (můj) kód kódem spouštěného programu
}
... // tady pokračuje kód rodičovského procesu
```

V kódu byla volána funkce `fork()`. Tato funkce vytvoří kopii původního procesu, ve kterém je volána, tedy po volání funkce máme dva téměř identické procesy.[1] Jedna kopie je rodičovský proces, druhá je potomek. V rodičovském procesu funkce `fork()` vrací PID vytvořeného potomka, v potomkovi vrací 0, tedy proces přes návratovou hodnotu „pozná, kým je“. Lze také otestovat zápornou hodnotu, ta by

[1] Jejich kód v kódovém segmentu je identický, zatím mají společné i datové segmenty, což je řešeno mechanismem copy-on-write. Shodná je také hodnota v registru čítače instrukcí.

indikovala chybu. Pouze potomek zavolal funkci `execve()` (nebo podobnou, je jich více), kterou získal svůj vlastní kód odlišný od rodiče.

Některé operační systémy (například UNIXové systémy) 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), toto číslo se právě používá při konstrukci stromu procesů.

Ve Windows je pojetí struktury procesů značně volné, vlastně zde ani neexistuje strom procesů s jediným kořenem. Naopak v UNIXových systémech je stromová struktura striktně dodržována, procesem v kořeni stromu procesů je `init`, případně může mít jiný název (což se děje např. při použití inicializačního systému systemd).

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 výjimky, 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).

To znamená, že po ukončení procesu

- jsou potomci také ukončeni,
- se jeho (bývalí) potomci stanou potomky procesu v kořeni stromu, pokračují ve své činnosti,
- se všichni potomci stanou sirotky, nemají žádný rodičovský proces.

Ve Windows se setkáme většinou s třetí možností, i když rodičovský proces může také ukončit svého potomka při svém ukončení (není to obvyklé). Proto zde místo stromu procesů máme vlastně několik nezávislých stromů, jediný kořen neexistuje.

V UNIXových systémech včetně Linuxu se naopak setkáme s prvními dvěma možnostmi. Pro většinu procesů platí, že když je jejich rodič ukončen, jsou také ukončeni (signál k ukončení činnosti se posílá rekurzívně do celého podstromu. Jestliže některý proces má pokračovat ve své činnosti i po ukončení svého rodičovského procesu, je třeba ho touto vlastností předem označit (je spuštěn příkazem `nohup`), tento proces se hned po takovémto spuštění stává přímým potomkem procesu `init`.

Proces může být ukončen jedním z těchto způsobů:

- ukončí sám sebe, například voláním funkce `exit()`,
- při ukončení rodičovského procesu, pokud se nemá stát sirotkem,
- je ukončen svým rodičem (rodičovský proces zavolá funkci `abort()`),
- je odstraněn uživatelem či systémem, a to buď „dobrovolně“ nebo je ukončen bez své spolupráce (násilně, například při zamrznutí procesu nebo po výjimce).

Ve Windows se obvykle setkáme s prvním a posledním způsobem ukončení, i když, jak bylo výše uvedeno, proces může být ukončen také svým rodičem (a to i těsně před ukončením toho rodiče). V UNIXových systémech se běžně používají všechny čtyři možnosti, pro ukončení procesu podle posledního bodu lze použít například příkaz (funkci) `kill` nebo jeho varianty.

Rodičovský proces může počkat na dokončení práce potomka (použije volání jádra `wait`) nebo pokračovat ve své činnosti bez ohledu na to, co potomek dělá. Může taktéž v kterémkoliv okamžiku potomka ukončit (`abort`), například tehdy, když potomek splnil svůj úkol, pro který byl spuštěn.

4.2 Běh procesů a multitasking

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[2], segmentové registry, zásobníkový registr[3]),
- registr příznaků[4],
- 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.

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.

Pokud máme víceprocesorový systém nebo systém s jedním vícejádrovým procesorem, mohou procesy běžet paralelně (třetí způsob), přičemž se uplatňuje i sekvenčně-paralelní běh, protože obvykle máme víc spuštěných procesů než jader procesoru.

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.

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).

U víceprocesorového systému nebo systému s více jádry jsou procesorová jádra přidělována procesům (kterých je obvykle víc než jader) podobným způsobem, taky se jedná o multitasking.

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

[3] 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.

[4] 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).

4.2.1 Pseudomultitasking

Pseudomultitasking je multiprogramování, které není přímo multitaskingem, ale jeho předchůdcem. Oproti plnohodnotnému multitaskingu systém buď vůbec neřeší vztahy mezi procesy a nechává spouštění potomků, střídání na procesoru atd. zcela v režii samotných procesů, nebo umožňuje běžet jednomu (běžnému) hlavnímu procesu a pak několika pomocným procesům naprogramovaným pro tento účel.

Příklad

Pseudomultitasking ve starém MS DOSu probíhal tak, že bylo možné mít spuštěné tzv. TSR programy (Terminate and Stay Resident) jako je antivirus či program na transformaci znakových tabulek pro zobrazování, a pak jeden „běžný“ program. Rezidentní program po načtení své rezidentní části (té, která zůstane v paměti) předává řízení příkazovému interpretu (`command.com`), který buď spustí další rezidentní program, nebo počká na další příkazy uživatele.

Nejstarší verze systému Apple MacOS používaly pro správu procesů modul `Finder`, který umožnil spustit jeden běžný proces a pak další procesy typu accessories (pomůcky), které byly pro tento účel speciálně naprogramovány. Mohla to být například kalkulačka nebo jednoduchý textový editor.

O něco novější verze systému Apple MacOS přešly na modul `MultiFinder`, který už netrval na tom, aby další spuštěné programy byly pro tento účel speciálně naprogramovány, nicméně ještě pořád nešlo o plnohodnotný multitasking. Ten se v MacOS objevil až po přechodu na UNIXovou architekturu.

Při použití pseudomultitaskingu se kontext buď vůbec nepoužívá (například u TSR programů v DOSu), nebo kontext sice existuje, ale může být jen velmi krátký. Procesy totiž na přepínání procesů na procesoru aktivně spolupracují.

4.2.2 Kooperativní multitasking

V kooperativním multitaskingu 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 anebo, 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).

Je tedy dovoleno běžet i 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 pseudomultitaskingu. Kontext taktéž nemusí být moc rozsáhlý, protože procesy na této formě multitaskingu spolupracují.

Kooperativní multitasking sice teoreticky umožňuje implementovat grafické uživatelské rozhraní, ale rozhodně není pro tento účel ideální, protože by takové prostředí nebylo dostatečně interaktivní. Představte si například, že proces zobrazující momentální datum a čas bude celé minuty čekat, až mu proces popředí (například textový editor) „dovolí“ aktualizovat své zobrazení...

V kooperativním multitaskingu pracovaly například Windows s DOS jádrem verze 3.x nebo novější

verze Apple MacOS před verzí X.

4.2.3 Preemptivní multitasking

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 (IRQ, výjimky) a procesor je přidělen tomu procesu, kterého se přerušení týká (například po přerušení od klávesnice je procesor přidělen tomu procesu, kterému patřilo stisknutí klávesy na klávesnici, třeba textovému editoru). Není řečeno, že se kontext přepíná po každém přerušení, protože přerušení může být (a často bývá) určeno momentálně aktivnímu procesu.

„Preemptivní“ znamená preventivní, předvídavý, předjímající, tj. procesy musí předpokládat (předvídat), že mohou být kdykoliv přepnuty, nesmí spoléhat na to, že budou mít jakkoliv dlouho k dispozici procesor. Reálně to znamená, že procesy ani „netuší“, že jsou přepínány, čas na procesoru berou spojitě (nepočítají přestávky).

Kontext procesu musí obsahovat i takové údaje jako stav registrů procesoru a koprocesoru, protože proces po odebrání a znovupřidělení procesoru nebývá 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 anebo ještě dřív, pokud v jemu přiděleném intervalu došlo k přerušení generujícímu událost, anebo když svou práci dokončí před koncem intervalu.

Grafické uživatelské rozhraní s rozumnou mírou interaktivity lze vytvořit prakticky jen v systému s preemptivním multitaskingem. Obrácená implikace ovšem neplatí – systém s preemptivním multitaskingem nemusí být nutně opatřen grafickým rozhraním.

Preemptivní multitasking je také podmínkou pro naprogramování *víceuživatelského systému*, ve kterém je třeba umožnit pracovat více uživatelům paralelně. Bylo by nešťastné, kdyby na popředí přetrvával proces jednoho uživatele a ostatní by museli počkat, až jejich procesům bude umožněno používat procesor.

Podobně je to i s *bezpečnostními mechanismy*: systém dokáže ochránit procesy (i navzájem) jen tehdy, pokud má jejich běh pod přímou kontrolou a nehrozí „únos“ procesoru.

Další velkou výhodou preemptivního multitaskingu je to, že ho lze snadněji implementovat (ano, kooperativní multitasking má méně výhodných vlastností a ještě k tomu se hůře navrhuje a programuje). Kontext ovšem musí být rozsáhlejší než u kooperativního multitaskingu, protože procesy nevědí předem, že ztratí procesor. Tedy opravdu všechny registry procesoru, které se nějak vztahují k procesu, musejí být součástí kontextu a jsou při odchodu procesu z procesoru uschovány do PCB.

Tuto metodu používají prakticky všechny moderní operační systémy – UNIXové systémy včetně Linuxu (pro běžné procesy), Windows NT a Windows s DOS jádrem od verze 4 (95), Apple MacOS X

a novější.

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).

4.3.1 Princip

V operačních systémech podporujících multithreading se proces (úloha) skládá z jednoho nebo více podprocesů nazývaných *vlákna* (thread), 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. Tak jako proces má své číslo PID, tak i každý thread má přiděleno číslo TID, které v operačních systémech bývá jednoznačné pro celý systém.

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 tak přísné mechanismy ochrany paměti ani další bezpečnostní metody (vlákna patří témuž procesu, spolupracují, nekonkurují si), nicméně určitá synchronizace při přístupu k prostředkům dostupným více vláknům téhož procesu může být zapotřebí.

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.

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 Apple MacOS, dále také ve Windows řady NT a v Linuxu (v Linuxu je však implicitně samotné jádro jednovláknové) s knihovnou NPTL (Native Posix Threads Library). Tato specifikace je součástí normy POSIX.

Model N:1. Vlákna jsou realizována *na uživatelské úrovni*. Podpora vláken je realizována v knihovnách, jádro je pouze jednovláknové a „vidí“ pouze procesy, nikoliv vlákna. 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 (pokud proces příliš mnoho nekomunikuje s jádrem), proto je lepší odezva jednotlivých uživatelských 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.

Toto řešení je používáno v některých jazycích jako vlastní implementace vláken (například v Javě nebo Ruby). Dále se s ním setkáme ve formě *Windows Fibers* („vlákénka“) – ve Windows totiž kromě vláken (threads) existují také vlákénka, jejichž plánování je plně v režii aplikace (tedy jsou viditelná pouze v uživatelském prostoru), a to funkcí `SwitchToFiber`.

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).

Poznámka:

Jednotlivé modely jsou odvozeny od toho, kolik kterých typů vláken je mapováno mezi uživatelským prostorem a jádrem. Model 1:1 znamená, že jedno vlákno v uživatelském prostoru je mapováno na jedno vlákno v jádře (tj. jádro pracuje přímo s uživatelskými vlákny). Model N:1 znamená, že více uživatelských vláken (tj. vlákna jednoho procesu) je mapováno na jedno společné vlákno jádra (jádro nevidí vlákna, jen procesy). Model N:M představuje řešení, ve kterém množina uživatelských vláken může být mapována na (obecně různě početnou) množinu vláken jádra, tedy může nastat dokonce situace, kdy jedno uživatelské vlákno je napojeno na více vláken jádra, což by v předchozích modelech bylo nemožné.

V mobilních operačních systémech je správa procesů ještě komplikovanější. Například v Androidu rozhodně neplatí přímý vztah mezi aplikací a procesem: jeden proces může být sdílen několika aplikacemi, a naopak jedna aplikace se může vztahovat k více procesům. Rozlišují se aplikace běžící na popředí a na pozadí, což trochu připomíná kooperativní multitasking, ale tím podobnost končí, protože přepínání mezi aplikacemi je zcela v režii systému. Jednotlivé aplikace jsou totiž vzájemně striktně odděleny virtualizací v běhovém prostředí Dalvik.

Možnost porozumět celému mechanismu se komplikuje i tím, že pojem „multitasking“ v mobilních zařízeních poněkud zlidověl a běžní uživatelé pod tímto pojmem chápou možnost přepínat mezi okny různých aplikací.

Další informace:

Problematika multitaskingu v Androidu je zajímavě popsána například zde:
https://android-developers.googleblog.com/2010/04/multitasking-android-way.html

4.3.2 Programování vícevláknových aplikací

Ve vícevláknové aplikaci mohou nastat tyto (krajní) situace:

1. Vlákna jsou vzájemně nezávislá, nesdílejí společné prostředky, navzájem nekomunikují: ideální případ, mohou bez problémů běžet zároveň na různých procesorech nebo jádrech.
2. Vlákna sdílejí prostředky nebo jiným způsobem navzájem komunikují: nutno synchronizovat, vzájemné čekání, nemohou běžet neustále zároveň.

Vícejádrový procesor využijí například tyto aplikace, pokud jsou vícevláknové:

- hry,
- videokodeky (například kodek H.264 dokáže takto efektivně využít až 8 jader), animační programy, multimediální aplikace,
- matematické programy, náročné výpočty,
- komprimace, šifrování,
- jakékoliv aplikace programované technologií *Model–View–Controller* nebo dalšími technologiemi dělícími činnosti do různých vláken, atd.

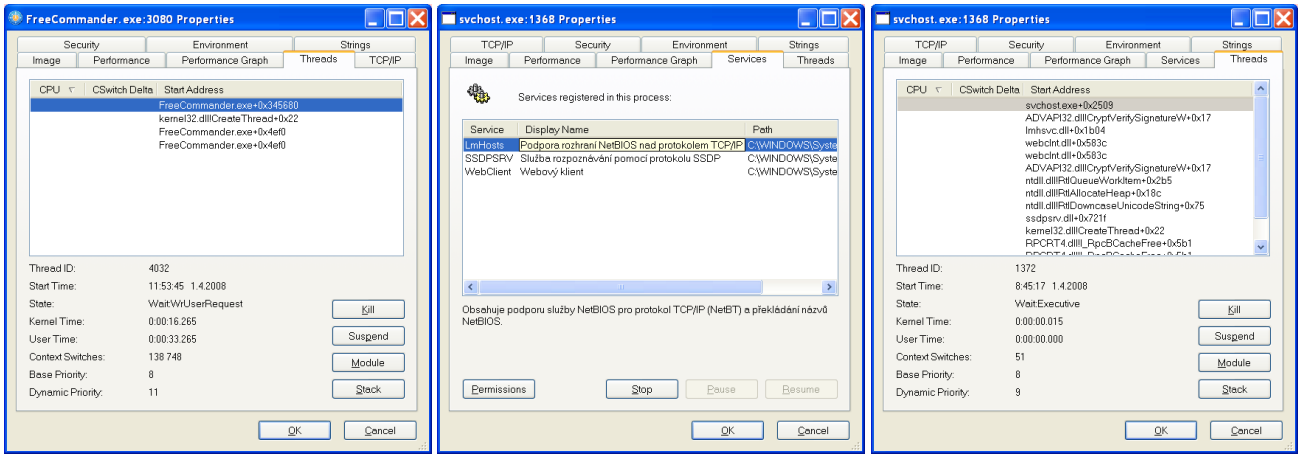
Při programování vícevláknových aplikací si především musíme dát pozor na tyto problémy:

- *Waiting* (čekání) – vlákno potřebuje k další práci výsledek činnosti jiného vlákna, což znamená degradaci paralelismu na sekvenční zpracování.
- *Synchronizační problém* – vlákno potřebuje výsledek výpočtu jiného vlákna, ale není mu známo, že druhé vlákno tento výsledek ještě nedodalo ⇒ je možné, že pracuje se špatnými hodnotami.
- *Deadlock* (uváznutí) – vlákno čeká na prostředek, který blokuje jiné vlákno, to třeba čeká na prostředek blokovaný prvním vláknem ...
- *Race-Condition* – dvě vlákna chtějí tentýž prostředek, který však může být využíván jen jedním – buď je vygenerována chyba či výjimka, třebaže obě vlákna pracují správně, nebo dojde k uváznutí, nebo zvítězí „rychlejší“ a druhé vlákno musí počkat, anebo druhé musí hledat jiný prostředek (pokud je to možné), závisí na plánovacím algoritmu plánovače procesoru.

Programátoři často používají vlákna tam, kde to nejen není potřeba, ale dokonce si takto lze „vyrobit“ mnoho problémů. Používání vláken je třeba velmi důkladně zvážit, abychom zbytečně nezhoršovali výkon a odezvu své aplikace.

Programování více vláken ve Windows: Po vytvoření procesu (funkcí `CreateProcess`) existuje jedno „implicitní“ hlavní vlákno, další vlákna lze kdykoliv vytvořit voláním API funkce `CreateThread` – Win API funkce.

Ovšem pokud pracujeme v některém rozsáhlejším vývojovém prostředí, je vhodnější používat funkce a třídy nabízené tímto prostředím, protože v konstruktorech a dalších souvisejících funkcích či metodách jsou vytvářeny a inicializovány další související objekty, které by při použití API funkcí nefungovaly správně.



Obrázek 4.9: Vlastnosti některých vícevláknových aplikací v Processs Exploreru

Další informace:

Vícevláknová aplikace ve Visual C++:
https://learn.microsoft.com/en-us/cpp/parallel/sample-multithread-c-program?view=msvc-170
Vícevláknová aplikace v C#:

- https://stackify.com/c-threading-and-multithreading-a-guide-with-examples/
- https://www.c-sharpcorner.com/article/Threads-in-CSharp/

Programování více vláken v Linuxu: V Linuxu se dřív místo vláken používala struktura podprocesů, tedy volání `fork` a `exec`:

- voláním `fork` proces vytvoří svou kopii, voláním `exec` jí předá kód z daného spustitelného souboru,
- původní proces je rodičem nového procesu, má nad ním plnou kontrolu.

Příklad

Takto se dříve vytvářela „pseudovlákna“ v Linuxu:

```
pid_t child_pid;
child_pid = fork();
if (child_pid == 0) {  // jsme v novém procesu
  execvp(název_programu, arg_list);   // v kopii nahraď kód kódem spouštěného programu
}
... // tento kód patří rodiči
```

V současné době se však běžně používají pravá vlákna, většinou z knihovny, jejíž hlavičkový soubor je `pthread.h`.

```
pthread_t id_cislo_threadu;
pthread_create (&id_cislo_threadu, NULL, &funkce_threadu, NULL);
```

Jednotlivé parametry jsou

1. reference na proměnnou, do které se uloží TID vlákna,
2. ukazatel na objekt „atribut vlákna“, ve kterém lze určit způsob, jakým bude vlákno komunikovat s ostatními vlákny, hodnota `NULL` nastaví výchozí hodnoty,

3. reference na funkci s kódem, který bude vlákno vykonávat, obvykle typu `void *`,
4. ukazatel na argument dat předávaných vláknu (typu `void *`), může být `NULL`.

Dále je třeba program přeložit. Kromě vývojových prostředí (například KDevelop) máme k dispozici především překladač `gcc` (pro zdroje v C) nebo `g++` (pro zdroje v C++), s přepínačem `-pthread` nebo `-threads` (podle hardwarové a softwarové architektury).

Například:

```
gcc -pthread -o vystupni_soubor zdroj.c
```

Příklad

Jednoduchý způsob naprogramování vláken podle standardu POSIX v jazyce C nebo C++ je například tento, zde na rozdíl od předchozího postupu používáme jinou knihovnu:

```
...  // knihovny
#include <thread>

void funkce_vlakna(string argument1, int argument2) {
  ... // kód, který má dané vlákno provádět
}

int main() {
  ...
  thread vlakno(funkce_vlakna, "argument 1", cislo); // vznikne vlákno, začne pracovat
  ...              // mezitím hlavní vlákno může provádět vlastní kód
  vlakno.join();  // toto použijeme, pokud zde chceme počkat na ukončení vlákna
  ...
}
```

Při překladu uvedeného kódu překladačem gcc je třeba přidat parametr `--pthread` (podle POSIX thread).

Další informace:

- `man gcc`
- https://www.root.cz/serialy/programovani-pod-linuxem-pro-vsechny/
- http://www.linux.cz/noviny/1998-0809/index.html

4.3.3 Další možnosti programování více vláken

Multiplatformní řešení. Některé multiplatformní jazyky obsahují také podporu vláken, například

1. *Java* podporuje dva typy vláken:
   - native threads – využívá možnosti operačního systému, jde obvykle o kernel threads,
   - green threads – vlastní řešení.
2. Skriptovací jazyky často implementují vlastní vlákna (ve vlastní režii, ve skutečnosti něco jako user threads), například *Lua* (Coroutines), *Ruby* (třída `Thread`).

OpenMP je open-source projekt, který umožňuje snadno programovat vícevláknové aplikace v jazycích C, C++ a Fortran. V kódu pomocí direktiv (to jsou speciální metapříkazy) určujeme, který kód má být prováděn vícevláknově, tedy rozdělen do více samostatných výpočetních vláken (v direktivě

určíme počet vláken). Daný blok označíme pomocí direktiv, tento kód je pak automaticky paralelizován. Programový kód je pak přeložen s parametrem určujícím, že se během překladu má použít OpenMP modul.

Tento přístup se nedoporučuje kombinovat s klasickými způsoby programování vláken. V tomtéž zdrojovém souboru se kombinuje jak samotný kód k provádění, tak i direktivy pro řízení paralelizace.

SYCL (čte se „sikl“) je specifikace jazyka určená pro podobné účely jako OpenMP. Používá se podobně (přímo do kódu k provádění se vpisuje řízení paralelizace), jen místo direktiv má příkazy ze speciální knihovny. Toto řešení prosazuje hlavně Intel ve svém překladači DPC++.

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 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.

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.

Příklad

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

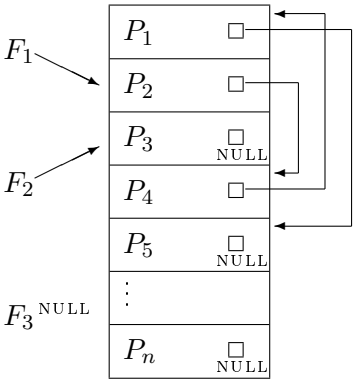
Udržování takové fronty je jednoduché. V určitých časových intervalech je snižován údaj u prvního procesu ve frontě, a když dosáhne nuly, proces je „probuzen“, odstraněn z fronty. Protože u druhého procesu v pořadí byl evidován rozdíl vlastního času čekání a času čekání prvního procesu, který je teď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

Jedním z možných způsobů *implementace* je 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.

Na obrázku 4.10 je příklad takové implementace front procesů. Ve frontě F_1 čekají procesy P_2, P_4 a P_1, ve frontě F_2 pouze proces P_3 a fronta F_3 je momentálně prázdná. Ve struktuře popisující danou frontu je ukazatel na první proces ve frontě, v PCB každého procesu je odkaz na následujícího člena fronty.



Obrázek 4.10: Fronty procesů jako řady ukazatelů na PCB

Ve Windows máme dva typy front – fronty připravených procesů (čekají na procesor) a fronty procesů čekajících na konkrétní zařízení (I/O fronty, každé zařízení může mít svou frontu).

V UNIXových systémech je správa front poměrně flexibilní. Existují samozřejmě fronty připravených procesů a I/O fronty, ale také fronta typu delta-list pro procesy čekající na uplynutí stanovené doby (tu spravuje tzv. *time manager*) a fronty související se síťovými službami.

4.5 Přidělování procesoru

4.5.1 Základní pojmy pro plánování procesoru

U přidělování procesoru budeme obecně hovořit o procesech, ale ve většině současných operačních systémů se procesor přiděluje vláknům.

Na přidělování procesoru se podílejí dva moduly systému:

- *plánovač procesoru* (CPU Scheduler) řídí frontu (fronty) připravených procesů a určuje, kterému procesu je přidělen procesor a na jak dlouho,
- *dispatcher* 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. Dispatcher také přepíná mezi uživatelským a privilegovaným režimem.

Definice (burst time, kvantum)

Burst time (execution time) daného procesu je (souvislá) doba, kterou proces reálně stráví na procesoru, kde nechá provádět svůj programový kód. Obvykle jde o desítky až stovky milisekund.

Časové kvantum (time slice) daného procesu při použití preemptivního multitaskingu se sdílením času je časový interval, který proces může postupně využívat prováděním svého kódu na procesoru. Z kvanta se vždy odečte každý využitý interval burst time. Po vypotřebování časového kvanta dostane proces další časové kvantum, typicky při určité penalizaci.

Časové kvantum můžeme tedy chápat jako „předplacenou“ dobu, ze které si proces postupně ukrajuje formou burst time stráveného reálně na procesoru. V preemptivním multitaskingu se sdílením času dostává proces na procesoru opravdu jen krátké souvislé úseky, jejichž délka je právě první z definovaných pojmů, a tuto délku určuje správce procesů.

Účelem časového kvanta z druhého definovaného pojmu je zamezit situaci, kdy by jeden velmi aktivní proces „okupoval“ procesor do takové míry, že by se na další procesy moc nedostávalo. Zmíněná penalizace může být například mírné snížení priority procesu (protože používáme dynamické priority), což by znamenalo, že dotyčný velmi aktivní proces by sice mohl dál pracovat (získal by další časové kvantum), ale jiné procesy by ho mohly v případě potřeby předběhnout ve frontě na procesor (jejich priorita by nebyla snížená).

Na vhodné délce intervalu burst time 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, má horší propustnost. Jestliže je naopak burst time zbytečně velký, pak procesy hodně používající I/O zařízení využívají jen malou část přiděleného časového intervalu a procesor musí být stejně přepínán častěji, a navíc je systém méně interaktivní.

✎ Procesy můžeme rozdělit na

- *CPU-bound* – procesy, které hodně využívají procesor (například výpočetní úlohy nebo služby),
- *I/O-bound – interaktivní procesy*, které více využívají I/O zařízení (včetně grafického výstupu nebo různých typů vstupu),
- realtimové procesy (vlastně je to specifická podmnožina první skupiny).

Každý z těchto typů procesů má trochu jiné nároky na procesor, CPU-bound procesy obvykle využijí celou dobu burst time, u I/O-bound procesů je zase mnohem větší pravděpodobnost přerušení běhu. Realtimové procesy jsou ve svých požadavcích na burst time ještě striktnější než CPU-bound. Plánovač procesoru by měl rozlišovat mezi těmito skupinami procesů, aby bylo využití procesoru optimální a aby byl přidělován procesům ze skupin rovnoměrně.

✎ Plánování procesů můžeme rozdělit do tří oblastí:

1. *Dlouhodobé plánování* souvisí se samotným návrhem multitaskingu (tedy provádí se při návrhu architektury systému), 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 v konkrétní situaci odložen (suspended, resp. jeho paměťové stránky) a tedy mu není po určitou dobu 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* – proces právě využívající svůj burst time (tj. je na procesoru) může být kdykoliv přerušen a umístěn zpět do některé fronty procesů čekajících na procesor, to se typicky děje při přerušení (včetně přerušení od timeru) nebo výjimce,
- *nepreemptivní* – proces, který právě zpracovává svůj kód, nemůže být přerušen, může z procesoru odejít podle svého rozhodnutí.

Pokud proces odchází z procesoru při preemptivním plánování, pak většinou opět do fronty procesů čekajících na procesor, kdežto pokud proces odchází z procesoru při nepreemptivním plánování, je to obvykle do jiné fronty (například čekání na přístup k souboru, některému objektu či zařízení, zpracování systémového volání, atd.).

4.5.2 Metody

Dále probereme základní metody plánování procesoru. Samotné popisované metody jsou pouze základem, typické použití je kombinace několika těchto metod, což uvidíme na konkrétních implementacích pro Windows a Linux.

 **Fronta (FCFS, First Come First Served, FIFO).** 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 samy neodevzdají procesor. To se stává například tehdy, když se potřebují přesunout do jiné fronty (třeba čekají na otevření souboru).

Nevýhodou je, že CPU-bound procesy si vyhrazují příliš mnoho času procesoru, a tedy I/O-bound procesy jsou znevýhodněny.

 **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 (burst time), je to tedy preemptivní metoda. Po odebrání procesoru je tento proces zařazen na konec fronty připravených procesů, pokud není (například při přerušení jemu určeném) zařazen do jiné fronty nebo převeden do stavu sleeping či suspended.

Každému procesu je procesor přidělen na stejnou dobu (burst time). Pokud je tento časový interval příliš velký, metoda svou funkčností odpovídá předchozí metodě. Opět jsou zvýhodněny CPU-bound procesy, protože předbíhají I/O-bound procesy čekající na přidělení zařízení.

 **Nejkratší úloha (SJF, Shortest Job First).** Také se nazývá Shortest Process Next (SPN). Procesor je přidělen tomu procesu, u kterého se předpokládá nejkratší doba jeho využívání (nejmenší burst time). Fronta je vedena jako prioritní s tím, že priority jsou zde určeny velikostí předpokládaného využitého burst time.

Metoda má preemptivní i nepreemptivní verzi. Pokud je do fronty připravených řazen proces, u něhož se předpokládá menší využití intervalu burst time 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.

Tato metoda zvýhodňuje I/O-bound procesy, jejichž využití burst time 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).

 Je nutné co nejlépe odhadnout využití burst time pro nastávající úsek běhu procesu. Pro odhad existuje více možností (označme n počet dosavadních přidělení procesoru danému procesu, R pole

hodnot skutečně využitých časových úseků, zatím o délce n, a S pole odhadů časových úseků):

1. Následující burst time bývá často stejný 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 vhodný burst time odhadujeme výpočtem aritmetického průměru všech předchozích skutečných časových úseků. Aby nebylo nutné vždy počítat aritmetický průměr všech hodnot, lze vzorec zjednodušit využitím předchozího odhadu a odpovídajícího skutečně využitého časového intervalu.

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

3. Zkombinujeme oba přístupy (podle vzorců 4.1 a 4.3), tedy budeme předpokládat, že další skutečně využitý úsek v době burst time 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 úseku, a čím blíže je nule, tím větší váhu mají rozdíly v dřívějších úsecích. 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:

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

Plánování podle priorit. 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 využívání časového intervalu burst time (čím menší množství využitého času, tím vyšší priorita).

Priorita procesu může být určena staticky (*statická priorita*, nemění se za běhu 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.

Priority jsou běžnou součástí algoritmů plánování procesoru v současných operačních systémech. Windows do verze XP pracují pouze s prioritami procesů na procesoru, v UNIXových systémech a ve Windows od verze Vista a Server 2008 existují také I/O priority (pro plánovač přístupu ke zdrojům, například paměti či disku).

Kombinace metod s více frontami. Je vedeno více front připravených procesů, procesy jsou rozdělovány dle určité vlastnosti (většinou podle priority). Pro každou frontu je stanovena některá metoda plánování, a dále jedna z metod je uplatňována při rozhodování mezi frontami.

Plánování v jednotlivých operačních systémech. Je třeba si uvědomit, že plánovač procesoru je vlastně jednou z nejdůležitějších komponent jádra – na něm záleží, jak efektivně bude pracovat systém i procesy. V operačních systémech může být i víc plánovačů, přičemž každý může být vhodný pro jiný způsob využití systému (klasický, server, embedded, atd.), podle toho, jaké typy procesů je třeba upřednostňovat, aby byl běh systému co nejplynulejší a aby procesy pracovaly efektivně.

Plánovač (scheduler) je vlastně implementací všeho, co jsme si dosud řekli o multitaskingu a multithreadingu: určuje, jak se bude zacházet s frontami procesů/vláken, jak to je s prioritami procesů, kdy se má přepínat kontext, jaké má být kvantum a burst time, ...

4.5.3 Plánování procesoru ve Windows

Plánovač procesoru ve Windows (CPU Scheduler, `mstask.exe`) plánuje výhradně vlákna bez ohledu na počet vláken v jednotlivých procesech (tj. vlákna čekají ve frontách). V jádru existuje modul pro plánování přidělování procesoru (jeho jader) vláknům, a od verze Vista také scheduler pro I/O (plánuje přístup k dalším prostředkům).

Dispatcher, který provádí přepínání kontextu podle požadavků scheduleru, není konkrétní funkce či knihovna, jeho součásti jsou v různých modulech jádra. Vychází se z toho, že přepínání probíhá při události (událostmi řízené přepínání).

Při plánování vláken se používá preemptivní plánování s více frontami, vlákno na procesoru může být kdykoliv přerušeno, pokud o procesor žádá vlákno s vyšší prioritou. Pokud vlákno nevyužije celý interval, po který má přidělen procesor, může přenechat tento nevyužitý čas jinému vláknu z téhož procesu voláním funkce `SwitchToThread()`.

Pro každou prioritu existuje jedna fronta připravených procesů, tedy celkem 32 priorit, 0–31. Pokud má vlákno čekat na procesor, je zařazeno do fronty podle své dynamické priority (základní prioritu vlákna dědí od svého procesu). Vlákna s vyšší prioritou mají vždy přednost před vlákny s nižší prioritou, tedy přednostně je procesor přidělován vláknům čekajícím ve frontách s vyššími čísly priorit.

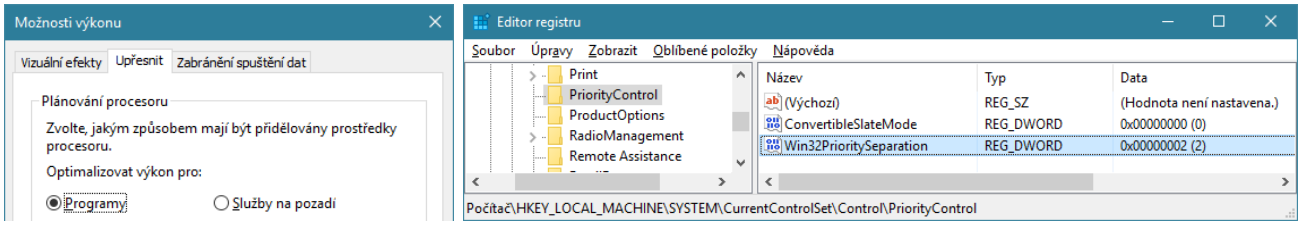
Procesor je přidělován na dobu odvozenou od *intervalu tiku systémových hodin*. Tento interval (burst time) je pevně stanoven na hodnotu někde mezi 10–15 ms, skutečnou hodnotu lze zjistit například programem `clockres` od Sysinternals. Standardně běží vlákno po dobu 2 intervalů na desktopu, na serveru 12. Při každém dalším tiku se odečte z kvanta běžícího vlákna pevná hodnota, kvantum se mírně snižuje i při čekání na zařízení. Po vyčerpání kvanta je vláknu přiděleno nové kvantum.

Dynamická priorita vlákna může být snížena, když vlákno vyčerpá dříve přidělené kvantum a je mu přiděleno nové. Priorita může být naopak vláknu zvýšena například při dokončení I/O operace nebo interaktivnímu vláknu při probuzení v důsledku události v okně. Realtimové procesy používají statickou prioritu, tedy nemění ji po celou dobu svého běhu (až na zásahy „zvenčí“, například od uživatele nebo od jádra).

Krátký interval burst time je výhodný v systému s mnoha interaktivními procesy (zvyšuje propustnost systému, typicky na desktopu se spoustou „okenních aplikací“), dlouhý interval je výhodný pro systém s mnoha výpočetními procesy často běžícími na pozadí (což jsou typicky servery).

Příklad

V grafickém rozhraní můžeme určit, zda bude výchozí délka burst time krátká (2 tiky) nebo dlouhá (12 tiků), a to v nástroji *Systém*, dále odkaz *Upřesnit nastavení systému*, karta *Upřesnit*, v části okna *Výkon* tlačítko *Nastavení*, pak opět karta *Upřesnit*, jak vidíme na obrázku 4.11 vlevo.

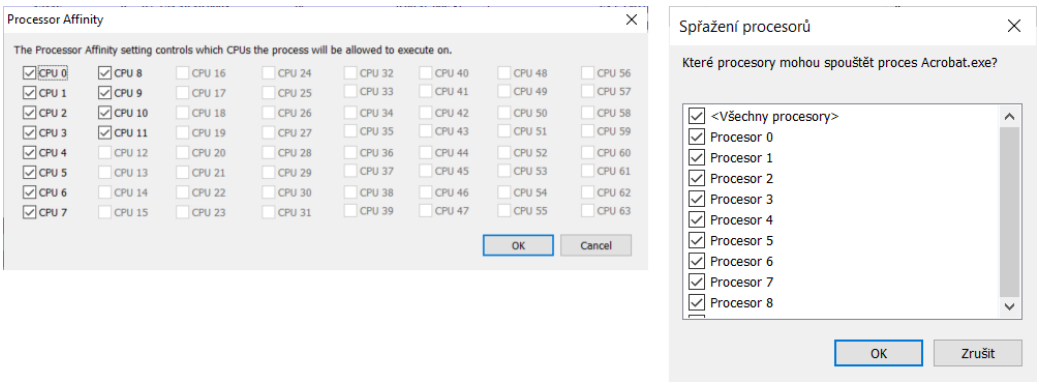


Obrázek 4.11: Stanovení délky kvanta vlákna a nastavení kvanta v registru

Podrobnější nastavení je v registru v klíči *HKLM/System/CurrentControlSet/Control/PriorityControl*, kde najdeme hodnotu *Win32PrioritySeparation* (na obrázku vpravo) – skládá se z 6 bitů, jejich hodnoty určují, zda má být délka burst time krátká nebo dlouhá, proměnná nebo pevná, zda lze navýšit intervaly pro procesy na popředí.

Jak bylo výše uvedeno (v sekci o vláknech), ve Windows jsou kromě vláken (thread) také *vlákénka* (fibers). Plánování vlákének neprovádí centrální plánovač z jádra, ale provádí je samotná aplikace ve vlastní režii. Toto plánování je nepreemptivní, vlákno skládající se z více fiberů přepne na jiný fiber (vše v času procesoru tohoto vlákna), až když původní fiber „dobrovolně“ odevzdá procesor dalšímu vlákénku v rámci tohoto vlákna voláním funkce `SwitchToFiber()`.

Afinita procesu či vlákna je určení procesorů (nebo jader procesoru), na kterých může procesor běžet. Tuto množinu procesorů (jader) lze omezit nastavením *masky afinity*, například v Process Exploreru (v kontextovém menu procesu, volba *Set Affinity*) – vidíme na obrázku 4.12 vlevo nebo ve Správci úloh (na kartě *Podrobnosti*, v kontextovém menu, volba *Nastavit spřažení*), na obrázku vpravo.



Obrázek 4.12: Nastavení masky afinity procesu v Process Exploreru a ve Správci úloh

V programu lze použít buď API funkci `SetThreadAffinityMask()` nebo `SetProcessAffinityMask()`. Toto se nazývá „hard affinity“ (napevno omezujeme množinu procesorů), „soft affinity“ představuje pravidlo, že vlákno je přednostně plánováno na ten procesor (jádro), na kterém běželo naposledy.

4.5.4 Plánování procesoru v Linuxu

Linux (stejně jako většina jiných UNIXových systémů) je vícevláknový využívající model 1:1, tedy jádro plánuje vlákna, nikoliv procesy. V dalším textu sice píšeme o procesech, ale myslí se tím vlákna.

V uživatelském prostoru se používá preemptivní multitasking se sdílením času. Samotné jádro může být nepreemptivní, tj. proces běžící v režimu jádra nemůže být přerušen (ostatní ano), jádro lze přeložit s příznakem určujícím preempci jádra, pak bude preemptivní celý systém.

Jádro se může chovat buď plně nepreemptivně (příznak `CONFIG_PREEMPT_NONE` při překladu jádra), plně preemptivně (příznak `CONFIG_PREEMPT`) nebo dobrovolně preemptivně (příznak `CONFIG_PREEMPT_VOLUNTARY`). Pokud jde o desktopový systém, je pro jádro vhodná volba dobrovolně preemptivního chování (úloha běžící v režimu jádra, třeba obsluha systémového volání, se může dobrovolně vzdát procesoru), pro server se doporučuje nepreemptivní chování (úlohy v jádře se nevzdávají procesoru samy, ale protože jsou velmi krátké a dobře odladěné, nevadí to a procesor je lépe využíván, méně zatěžován režií přepínání). Plně preemptivní chování znamená vysokou odezvu, ale vyšší režii přepínání (častěji dochází k přepínání kontextu), je vhodné například pro embedded systémy.

Třídy procesů a schedulery. V Linuxu jsou procesy rozděleny do několika tříd, podle toho se k těmto procesům přistupuje včetně jejich přidělování na procesor.

- `SCHED_FIFO` je třída určená pro realtimové procesy, které mají běžet nepreemptivně.
- `SCHED_RR` je třída určená pro realtimové procesy běžící preemptivně.
- `SCHED_NORMAL` nebo `SCHED_OTHER` bývá pro všechny ostatní, tedy procesy s normální prioritou. Obvykle je používána jedna z těchto dvou tříd, ale mohou být používány obě, přičemž lze běžné procesy rozdělit do dvou podskupin.

Realtimové procesy jsou tedy zařazeny do některé z prvních dvou tříd. V obou se ve skutečnosti používají i priority: realtimové priority jsou z rozsahu 1–99. Pokud má běžet na procesoru proces ze třídy `SCHED_FIFO` a ve frontě je jich více, vybírá se ten, který má nejvyšší prioritu. Je doporučeno vhodně omezit dobu těchto procesů na procesoru, aby příliš neokupovaly výpočetní prostředky.

Postup

Pro tyto účely máme v systému `/proc` tyto „soubory“ s parametry:

```
root@sarka-pc:~#cat /proc/sys/kernel/sched_rt_period_us
1000000
root@sarka-pc:~#cat /proc/sys/kernel/sched_rt_runtime_us
950000
```

První vypsané číslo (v mikrosekundách, tj. 1 sekunda) znamená referenční dobu trvání 100 % intervalu pro následující parametr. Druhé číslo určuje, jakou část referenčního intervalu maximálně mohou využívat realtimové procesy, v tomto případě 0,95 sekundy, tedy většinu (ale ne všechen). Poslední dvě písmena názvu souborů znamenají mikrosekundy.

Pro plánování procesů ze třídy `SCHED_RR` platí téměř totéž, jen jde o preemptivní plánování se stanoveným burst time.

 Velikost burst time pro procesy plánované ve třídě `SCHED_RR` najdeme v souboru `/proc/sys/kernel/sched_rr_timeslice_ms`.

 Pro plánování procesů z realtimových tříd se tedy používá výše uvedený postup, ale pro procesy s běžnou prioritou (třídy `SCHED_NORMAL` a `SCHED_OTHER` existuje více různých plánovačů, mezi kterými si obvykle vybírá tvůrce linuxové distribuce.

V nejnovějších verzích Linuxu (od verze 6.6) se používá plánovač *EEVDF* (Earliest Eligible Virtual Deadline First). Tento plánovač se pokouší přiřazovat čas procesoru tak, aby všechny procesy se stejnou prioritou měly k dispozici stejný čas na procesoru. Každý proces má přiřazenu „virtuální dobu běhu“ (podle své priority, obdobu kvanta), která se odvíjí od priority procesu, kterou lze ovlivňovat určením hodnoty *nice*.

Dále ke každému procesu je evidován parametr „zpoždění“ (lag), podle kterého se hlídá míra využití jeho virtuální doby běhu. Pouze procesy s kladnou dobou zpoždění („eligible“) mohou dostat čas na procesoru. Proces s větší hodnotou zpoždění zatím neměl dostatečnou možnost využívat svou virtuální dobu běhu, kdežto proces se záporným zpožděním už „přečerpal“ a po určitou dobu nedostane procesor. Tato doba je taktéž algoritmem stanovena („eligible time“), po jejím uplynutí se procesu navýší hodnota zpoždění do kladných čísel.

Postup

Pro každý proces můžeme zjistit, jakým konkrétně způsobem je plánován. Stačí s právy roota postupovat takto:

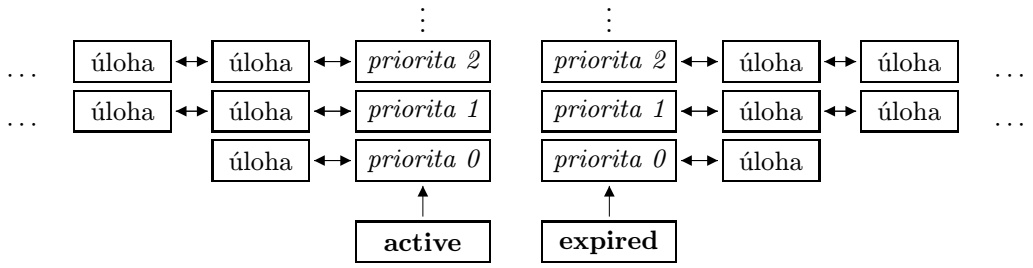
```
sarka@sarka-pc:~$pgrep firefox
1654
sarka@sarka-pc:~$pgrep migration
11
sarka@sarka-pc:~$sudo su -

root@sarka-pc:~#chrt -p 1654
pid 1654' current scheduling policy: SCHED_OTHER
pid 1654' current scheduling priority: 0

root@sarka-pc:~#chrt -p 11
pid 11' current scheduling policy: SCHED_FIFO
pid 11' current scheduling priority: 99

root@sarka-pc:~#exit
sarka@sarka-pc:~$
```

Vybrali jsme dva procesy. Jeden s běžnou prioritou (firefox), druhý s realtimovou prioritou (migration). O prvním z nich jsme se dozvěděli, že je plánován podle třídy `SCHED_OTHER` a jeho priorita (přesněji hodnota nice) je 0, vybraný realtimový proces je plánován podle `SCHED_FIFO` a jeho realtimová priorita je maximální, 99.

 V předchozích verzích se postupně vystřídalo několik plánovačů:

- *Completely Fair Scheduler* (CFS) inspirovaný řízením front přeposílaných paketů na síťových zařízeních, fronta procesů je implementována jako red-black tree, proto s ní algoritmus dokáže operovat velmi rychle, jeho časová složitost je O(log N),
- *plánovač O(1)* nazvaný podle své konstantní časové složitosti, byl efektivní zejména při velkém počtu běžících vláken,

- *plánovač O(n)* nazvaný podle své lineární časové složitosti, používal velké množství front (pro každou prioritu dvě fronty: jednu pro aktivní procesy, druhou pro procesy s vyčerpaným kvantem). Používal se pojem „epocha“: na začátku epochy dostaly všechny procesy nové kvantum, nová epocha začala poté, co všechny procesy vyčerpaly své kvantum nebo po uplynutí stanoveného intervalu.



Obrázek 4.13: Plánování procesoru v Linuxu s využitím plánovače O(n)

Proces (resp. jeho programátor) může volit plánovač, prioritu a afinitu (na kterých procesorech či jádrech má proces běžet). Existují funkce pro zjištění používaného plánovače, jeho nastavení (pro zjištění je funkce `sched_getscheduler()`, pro nastavení je funkce `sched_setscheduler()`) a podobně pro afinitu (například pro nastavení je funkce `sched_setaffinity()`). O ovlivňování hodnoty *nice* bylo psáno v sekci o prioritách procesů (str. 55). Další funkcí pro nastavení priority (obecnější, nastavuje nejen nice běžných úloh, ale i prioritu realtimových úloh) je `setpriority`.

To, zda je úloha *interaktivní*, systém určuje podle průměrné doby, kterou úloha strávila ve stavu spánku (`sleep_avg`, je to položka v `task_struct`) – interaktivní úlohy stráví v režimu spánku hodně času, protože hodně čekají například na to, než uživatel klepne na tlačítko na klávesnici nebo pohne myší. Hodnota se zvyšuje při každém probuzení ze spánku o hodnotu odpovídající době spánku, snižuje se za běhu úlohy na procesoru. Pomocí hodnoty `sleep_avg` se dá také podchytit (zjistit) proces, který zamrzl, protože spotřebovává velmi mnoho času procesoru a „nikdy nespí“.

4.6 Komunikace procesů

4.6.1 Princip 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),
- odkaz na data (adresa v paměti nebo na pevném paměťovém médiu, může jít o dočasný soubor),
- 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 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. Typicky se jedná o posílání zpráv nebo signálů. 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.

Poznámka:

Názvy funkcí jsou pouze ilustrativní. Skutečné názvy funkcí závisejí na operačním systému a konkrétním typu komunikace. Například v UNIXových systémech se především jedná o posílání signálů. Signál může být poslán pomocí funkce `kill()`, `killpg()` nebo `sigqueue()`, pro tento účel není žádná funkce `send()`. V jazyce C například pošleme signál SIGTERM (pro ukončení cílového procesu) takto:

```
kill(pid_cílového_procesu, SIGTERM);
```

Taktéž není žádná funkce `receive()`, místo toho existuje mechanismus pro obsluhu příchozích signálů, který mají procesy automaticky k dispozici a který určuje výchozí reakci na signály (v případě signálu SIGTERM je výchozí reakcí ukončení procesu). Navíc programátor může napojit (hang up, zavěsit) na signál svou vlastní funkci. Například pokud chce, aby se při obsluze příchozího signálu SIGTERM spustila funkce `on_sigterm(int num)`, použije tento příkaz v jazyce C:

```
signal(SIGTERM,on_sigterm);
```

Přímou komunikaci dělíme do dvou skupin:

- *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.

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. Komunikace probíhá pomocí funkcí

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

✎ *Socket* v síťovém (ale také lokálním) smyslu slova je brána, přes kterou probíhá komunikace, jde o komunikaci typu „klient-server“. Existuje více druhů socketů: síťové sockety a UNIX domain sockety.

Síťový socket je určen pro komunikaci přes počítačovou síť. Adresa socketu je uspořádaná dvojice (IP adresa, číslo portu), číslo portu je to, které známe z transportní vrstvy modelu ISO/OSI. Každé (jednosměrné) spojení na transportní vrstvě je jednoznačně určeno dvěma sockety – zdrojovým a cílovým. Sockety jsou tedy jednosměrné, pro obousměrnou komunikaci musíme použít dvojici socketů, jeden pro každý směr.

UNIX domain sockety jsou implementovány v UNIXovém jádře, nikoliv v síťových protokolech. Tento koncept je určen pro komunikaci uvnitř systému pro přenos dat mezi procesy, na rozdíl od síťových socketů v obou směrech. API pro tento typ socketů je podobné API pro síťové sockety, ale adresy těchto socketů jsou speciální typy souborů v souborovém systému (soubory typu „socket“).

✎ *Roura* (pipe, pipeline) je podobná socketu, ale její implementace je odlišná. Roura je vždy jednosměrná, tedy odesílatel vždy jen odesílá data, příjemce vždy jen přijímá data. Roury jsou taktéž implementovány jako speciální typ souborů (typ „pipe“), odesílatel zapisuje do tohoto souboru, příjemce z něj čte.

Roury jsou buď anonymní nebo pojmenované. Anonymní roury jsou obvykle vytvořeny mezi rodičovským procesem a jeho potomkem, nebo uživatelem v textovém shellu pomocí známého symbolu „|“. Anonymní roura je soubor, ale nachází se pouze v operační paměti.

Pojmenované roury jsou skutečné soubory na disku a mohou propojovat jednoho odesílatele s jedním nebo více příjemci, příjemci mohou být kterékoliv procesy, které znají název souboru pro rouru a mají přístup pro čtení.

4.6.2 Komunikace ve Windows

✎ **Zprávy oknům.** Komunikace v uživatelském prostoru probíhá především pomocí zpráv oknům. Každé okno má proceduru nazývanou *Window procedure* (procedura okna), která je volána vždy, když tomuto oknu byla doručena zpráva. Například pokud klikneme na symbol křížku v rohu okna, posíláme zprávu „zavřít“ danému oknu (pokud jde o hlavní okno aplikace, pak „ukončit aplikaci“).

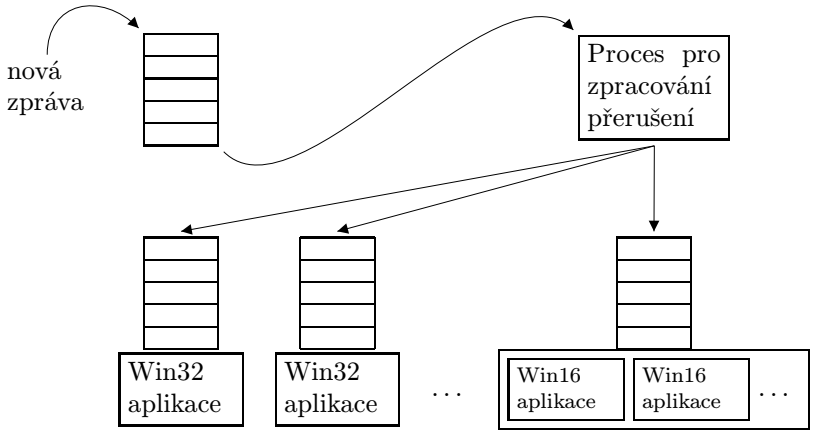
Důležitou roli hraje především procedura hlavního okna aplikace – pokud přestane odpovídat (vyzvedávat si zprávy), pak aplikace „neodpovídá“, tedy možná zamrzla.

Každé okno je jednoznačně identifikováno svým manipulátorem (handle), jako ostatně kterýkoliv jiný objekt ve Windows, v tomto případě jde o handle typu HWND (Handle to Window). Součástí posílané zprávy je handle okna, kterému je zpráva určena, dále identifikátor zprávy (například `WM_PAINT` pro informaci, že aplikace má toto okno překreslit, protože došlo ke změně dat, která jsou v něm zobrazena), a další parametry upřesňující obsah zprávy.

Zprávy oknům mohou být od systému nebo od aplikací. Takovou zprávu tedy může poslat i aplikace (sama sobě nebo jiné aplikaci), ale jen tehdy, pokud zná handle okna (pokud možno handle hlavního okna aplikace).

Další informace:

- https://docs.microsoft.com/en-us/windows/desktop/winmsg/window-procedures
- https://docs.microsoft.com/en-us/windows/desktop/learnwin32/writing-the-window-procedure



Obrázek 4.14: Fronty zpráv ve Windows

Ve Windows řady NT má každá Win32 a Win64 aplikace svou vlastní frontu zpráv. Struktura celého doručování zpráv je znázorněna na obrázku 4.14 – události jsou nejdřív zařazeny do hlavní fronty, odkud je postupně vybírá proces zpracovávající přerušení, vytváří zprávy a zasílá je do front jednotlivých aplikací. Starší Win16 aplikace nemají své vlastní fronty zpráv (neumějí s nimi zacházet, ve Windows do verze 3.11 neexistovaly vlastní fronty), mají jednu společnou. Nicméně v NT systémech bývají uzavřeny do vlastních virtuálních strojů, takže reálně mají také každá svou vlastní frontu.

Existují dvě kategorie zpráv:

- zprávy určené do fronty zpráv,
- zprávy, které nejsou určeny do fronty zpráv.

Většina zpráv je prvního typu, například výše zmíněné `WM_PAINT`, nebo také `WM_KEYDOWN` nebo `WM_QUIT` (pro běžné uzavření okna či ukončení aplikace). Tyto zprávy mohou chvíli „počkat“ ve frontě.

Časově kritické zprávy jsou druhého typu, tedy nečekají ve frontě a musejí být zpracovány hned, například `WM_SETFOCUS` (okno získalo zaměření – focus, třeba díky IRQ od klávesnice směřovanému tomuto oknu), `WM_WINDOWPOSCHANGED` (umístění okna se změnilo), `WM_ACTIVE` (okno bylo aktivováno, například klávesnicí nebo myší), atd. Tento typ zpráv je poslán přímo proceduře okna bez čekání ve frontě.

Zprávy souvisejí s *událostmi* – aplikace jsou událostmi řízené, a při vzniku či vygenerování události, která se týká konkrétní aplikace, je tato aplikace informována zprávou. Proto je window procedure velmi důležitá součást aplikace a ve struktuře kódu stojí velmi vysoko. Cyklicky (v době, kdy aplikace nemá další kód na vykonávání) kontroluje, zda je aplikaci doručena nová zpráva, když ano, pak tuto zprávu

vyzvedne a zpracuje. Jde o cyklus typu *while*, v podmínce cyklu je čekání na vyzvednutí zprávy, v těle cyklu jsou funkce zpracování zprávy (zprostředkovaně se volají obslužné funkce, například pro akce při stisknutí klávesy, klepnutí myší nebo překreslení okna).

Systémová volání. Stejně jako v jiných operačních systémech, také ve Windows komunikuje běžný proces s jádrem formou systémových volání. Systémová volání jsou vlastně funkce v API, vlákna je používají, když je třeba provést kód jádra. Nemají možnost přímo volat kód jádra, systémová volání jsou jakýmsi překladatelem či zabezpečeným rozhraním.

LPC (Local Procedure Call). Tento mechanismus není podporován v API, jde o vnitřní mechanismus jádra (konkrétně je sice implementován v `NTDLL.DLL`, ale je nedokumentován, tudíž běžné uživatelské procesy k němu nemají přístup). Je to komunikace typu klient-server, rozlišuje se odesílající a přijímající strana.

Slouží především ke komunikaci v rámci jádra (například Winlogon takto komunikuje s podsystémem LSASS). Systémové procesy běžící v uživatelském režimu mají k tomuto mechanismu přístup, uživatelské procesy ne. Například proces `CSRSS.EXE` (Client-Server Runtime Subsystem, část podsystému Windows běžící v uživatelském režimu) využívá LPC ke komunikaci s některými knihovnami přes jádro.

RPC (Remote Procedure Call). Jde o volání procedury, která může být v adresovém prostoru jiného procesu (vlákna) či v aktivované knihovně v podporovaném formátu (PE, PE+, atd.). Může jít o lokální volání (v rámci jednoho systému, neplést si s LPC – to je něco jiného) nebo o fyzicky vzdálené volání (na jiný počítač v síti). Ovšem vzdáleně lze volat proceduru běžící na jiném počítači jen tehdy, když je spuštěna služba Remote Procedure Call (Vzdálené volání procedur).

RPC je podporováno v API, je tedy určeno i pro procesy běžící v uživatelském režimu (především pro ně).

APC (Asynchronous Procedure Call). APC je mechanismus, který umožňuje provádět „externí“ kód (nepatřící do aplikace) v kontextu této aplikace. Když proces čeká na událost (třeba I/O), může čas čekání vyměnit za obsluhu volání APC, tedy ve svém kontextu nechat běžet cizí kód.

Rozlišujeme systémové a uživatelské procedury APC (pro uživatelské musí existovat „povolení“ od vlákna vlastnícího daný adresový prostor). Většinou jde o provádění obsluhy systémového volání (to je kód z jádra, tedy externí) v kontextu vlákna, které toto volání provedlo (technicky to znamená, že vlákno zavolalo funkci, která je implementována v jádře, a tedy poskytuje své zdroje pro běh této funkce).

DPC (Deferred Procedure Call). DPC (zpožděné volání procedury) je dodatečná obsluha přerušení, kdy by samotná obsluha přerušení trvala příliš dlouho. Obvykle to probíhá takto: pokud obsluha přerušení vyžaduje větší transfery dat, které jsou časově náročné, nebo nastala chyba a některou činnost je nutné opakovat či ošetřit chybu, část obsluhy přerušení bude ve formě DPC čekat na procesor jako běžné procesy.

Zatímco rutina obsluhy přerušení je zpracována okamžitě a bez přerušování, DPC je plánována na procesoru jako jiné procesy, jen s o něco vyšší prioritou.

Roury a sockety. Princip rour (pipes) byl převzat z UNIXu, ale ve Windows je jinak (jednodušeji) implementován. Roura je sdílená paměť, do které odesílatel zapíše celý svůj výstup, pak příjemce

dostane celý tento blok dat na svůj vstup. Příjemce a odesílatel nepracují paralelně, příjemce je spuštěn až po ukončení odesílatele.

Mechanismus Windows Sockets (WinSock) je implementací UNIXových síťových socketů, je to součást implementace transportní vrstvy pro TCP/IP.

Mapování souborů a sdílení paměti. Účelem mapování souborů je přenos obsahu souboru bez nutnosti skutečného transferu dat v paměti. Je to výhodné především pro velmi dlouhé soubory – mapované soubory neblokují prostor v operační paměti, jen je jim přiřazena adresa a mapovací mechanismus určuje, kam konkrétně tato adresa míří. Tento koncept taktéž pochází z UNIXových systémů.

Sdílení paměti je speciální typ mapování souborů, kde data nacházející se v adresovém prostoru jednoho procesu jsou mapována do adresového prostoru jiného procesu.

Další možnosti. *Schránka* (clipboard) je sdílené rozhraní pro výměnu dat přístupné všem procesům s uživatelským rozhraním (vlastnícím objekt `WinSta0`).

DDE (Dynamic Data Exchange) je dynamická forma sdílení dat mezi procesy, přičemž obě strany musejí běžet zároveň. Ve skutečnosti jde o jakousi nástavbu schránky, která umožňuje vytvořit vazbu mezi klientem (příjemcem DDE dat) a serverem (poskytovatelem DDE dat). DDE se používá pro vkládání externích dat, se kterými si daná aplikace sama neporadí. DDE vazba není považována za příliš bezpečnou.

Účelem modelu *COM* (Component Object Model) bylo původně vytvořit komponenty nezávislé na programovacím jazyce a sdílet je mezi procesy, tyto komponenty se nacházejí v některých dynamicky linkovaných knihovnách. Každá komponenta má svůj identifikátor – CLSID (Class ID), a je definovaná svým rozhraním.

Na modelu COM jsou založeny i další (novější) technologie, například OLE, OCX, ActiveX, částečně také .NET Framework.

OLE (Object Linking and Embedding) je objektová technologie, která umožňuje vložit do datové struktury procesu či souboru objekt, se kterým tento proces (či provozovatel souboru) neumí zacházet. OLE klient vyhradí ve svém prostoru místo pro vkládaný objekt, a pokud je nutné s objektem něco provést, je přes OLE zavolán kód, který označujeme jako OLE server.

Mechanismus OLE používá mnoho aplikací, včetně kancelářských aplikací (pro sdílení objektů mezi různými částmi kancelářského balíku, například takto můžeme vložit excelovskou tabulku do wordovského dokumentu), webové prohlížeče používají OLE k zobrazení obsahu souborů ve formátu, se kterým si samy neporadí. Sdílený objekt se může nacházet buď ve zdrojovém dokumentu, nebo v cílovém dokumentu.

4.6.3 Komunikace v Linuxu

V UNIXových systémech mají procesy k dispozici velmi mnoho různých komunikačních prostředků. Zde najdeme jen ty nejdůležitější, ale přesto bude tato podsekce velmi dlouhá.

Signály. Pro vzájemnou komunikaci mezi procesy (vlákny) se velmi často používají signály. Jsou ideální pro posílání jednoduché informace, která se dá celá reprezentovat malým číslem.

Počet typů signálů závisí na hardwarové architektuře (především 32/64), obvykle se setkáme s alespoň 30 typy signálů. Jsou reprezentovány číslem nebo slovním označením (v příkazech lze používat obě formy, podle toho, co si lépe pamatujeme). Obvyklé hodnoty najdeme v tabulce 4.1. Některé signály

(SIGUSR1, SIGUSR2) lze definovat podle vlastních požadavků, například rodičovský proces si definuje vlastní význam těchto signálů pro komunikaci se svými potomky.

Signál může přijít kdykoliv, je to vlastně typ přerušení, programátor s tím musí počítat. Dokonce i systémové volání může být přerušeno signálem (obvyklou reakcí je okamžité ukončení ošetření systémového volání s tím, že se toto volání dá buď navázat nebo restartovat).

Proces může u konkrétního signálu

- tento signál *ignorovat* (proces vůbec nereaguje) – tato akce je u některých signálů výchozí, například u SIGCHILD, protože tento signál například nemá pro daný proces význam, ale některé signály ignorovat nelze (například SIGKILL),
- *blokovat* s pozdějším doručením – signál není „zahozen“, ale čeká na odblokování, pak je zpracován,
- nechat zpracovat *implicitní obslužnou rutinou* – ta u většiny signálů ukončí proces, například pro SIGTERM nebo SIGHUP, pozastaví proces (SIGSTOP), nebo ukončí proces a spustí debugger (SIGILL, SIGFPE),
- implementovat *vlastní obslužnou rutinu* – například při obdržení SIGTERM chceme uzavřít soubory, s nimiž pracujeme.

Signály lze chápat jako řízení jednoduchými událostmi bez nutnosti vytváření obslužného cyklu. Obslužná rutina by vždy měla být co nejkratší, vlastně pro ni platí totéž jako pro obsluhu hardwarových přerušení.

Název	**Číslo**	**Význam**
SIGHUP	1	změna v nadřízeném procesu (například byl ukončen), načti znovu své konfigurační soubory (pro démony) nebo se ukonči (běžné procesy), v současné době se používá spíše u démonů
SIGINT	2	přerušení (ukončení) běhu procesu z klávesnice, totéž, jako když zadáme Ctrl+C
SIGILL	4	Nesprávná (ilegální) instrukce
SIGFPE	8	výjimka související s racionálními čisly (floating point exception)
SIGKILL	9	signál pro okamžité ukončení (proces je nemůže ignorovat, často ani nestačí uklidit přidělené prostředky) – používá se u nereagujícího procesu
SIGTERM	15	ukonči se (regulérní ukončení, proces stačí uklidit své prostředky), tento signál může proces ignorovat (neměl by)
SIGPIPE	13	zápis do roury, ze které nikdo nečte
SIGUSR1	30,10,16	uživatelem (procesem) definovaný signál
SIGUSR2	31,12,17	uživatelem (procesem) definovaný signál
SIGCHLD	20,17,18	potomek ukončen, vyzvedni si výsledek
SIGSTOP	19,23	pozastav se (ekvivalent klávesy Ctrl+Z)
SIGCONT	18,25	pokud jsi byl předtím pozastaven, pokračuj v činnosti

Tabulka 4.1: Obvyklé signály v UNIXových systémech

Pokud má proces ovládat některý signál:

- potřebujeme knihovnu `signal.h` a případně `unistd.h`,
- naprogramujeme funkci ošetřující příjem signálu (měla by být co nejkratší, podobně jako funkce pro obsluhu IRQ),

- napojení na signál se realizuje použitím struktury `struct sigaction`, která obsahuje kromě čísla signálu také adresu ošetřující funkce nebo příznak, že má být signál ignorován, a další prvky.

Signál lze poslat procesu, jehož PID známe. To lze provést jak v binárním kódu spuštěného procesu, tak i například v textovém shellu (máme k dispozici funkce `kill`, `killall`, `pkill` apod.). Parametrem je číslo nebo slovní označení signálu (bez „`SIG`“), a dále určení procesu, kterému je signál určen. To je obvykle PID, ale funkce `pkill` přijímá také jiné typy identifikace procesu – název procesu, PID, GID, SID, apod., tedy tento příkaz můžeme využít také k ukončení celého podstromu procesů.

Pokud proces nemá být ukončen s koncem relace, musíme provést jednu ze dvou akcí:

1. Spustíme tento proces v jiné relaci – volání jádra `setsid()`, to je použitelné pouze tehdy, když volající proces není hlavním procesem skupiny.
2. Zajistíme, aby proces nebyl v seznamu aktivních úloh a nebyl mu zasílán signál SIGTERM (příp. SIGKILL) – máme k dispozici příkazy `nohup` a `disown`:

   ```
   nohup nějaký_program &
   disown %číslo_úlohy_nějaký_program
   ```

   Když si pak vypíšeme seznam úloh, tato v seznamu nebude (protože není aktivní).

Roury (pipes). Vynálezcem mechanismu roury je Doug McIlroy, jeden z nejdůležitějších tvůrců raného UNIXu. S mechanismem rour jsme se už seznámili na cvičeních předmětu Operační systémy I, tedy už víme, co to je a jak to funguje. Aktivace a po ukončení komunikace odpojení se provádí jen v jednom procesu (obvykle rodičovském), otevření a uzavření je nutné provést v obou komunikujících procesech. Platí vše, co bylo napsáno v sekci před částí o Windows.

Roury mohou být buď pojmenované nebo nepojmenované. S *pojmenovanými rourami*, které jsou vlastně rozšířením původních anonymních rour, se zachází podobně jako s jinými soubory, je to vlastně soubor typu *pipe*.

To znamená, že po aktivaci souboru roury (API funkcí `mkfifo`) tuto rouru (soubor) otevřeme příkazem `fopen` (jako jakýkoliv jiný soubor) v jenom procesu pro čtení, v druhém pro zápis, a až je „otevřena“ na obou koncích, můžeme přenášet data. Po použití soubor uzavřeme a pak odpojíme příkazem `unlink`.

Anonymní roury jsou podobné dočasným souborům, a na rozdíl od pojmenovaných rour je jejich velikost limitovaná (u starších jader Linuxu na 4 KiB, u novějších jader 64 KiB). Anonymní rouru lze vytvořit příkazem `pipe` v programovém kódu, případně použitím symbolu | v shellu.

Postup

Anonymní roura je poměrně běžný způsob komunikace v UNIXových systémech. Nejde jen o využití při řetězení v textovém shellu, ale především by rouru měl umět používat programátor. Ukážeme si vytvoření anonymní roury.

```
int roura_deskriptory[2];
int roura_vstup;
int roura_vystup;

// vytvoření roury, v parametru máme deskriptory pro konce roury:
pipe (roura_deskriptory);

roura_vstup = roura_deskriptory[0];   // konec roury pro čtení, výstup
roura_vystup = roura_deskriptory[1];  // konec roury pro zápis, vstup
```

Dále s deskriptory zacházíme stejně jako s deskriptory souboru, tj. používáme je ve funkcích pro zápis do souboru nebo čtení ze souboru.

Postup

Anonymní roura často slouží ke komunikaci mezi rodičovským procesem a jeho potomkem.

```
... // vložíme stdlib.h, stdio.h a unistd.h
int main() {
  int     r_deskriptory[2];
  pid_t   pid_potomka;

  pipe(r_deskriptory);  // taky můžu otestovat návratovou hodnotu: -1 znamená chybu
  pid_potomka = fork();

  if (pid_potomka == (pid_t) 0) {         // *** child ***
    FILE *soubor;
    char buffer[1024];
    close(r_deskriptory[1]);
    soubor = fdopen (r_deskriptory[0], "r");
    while (!feof (soubor) && !ferror (soubor) &&
            fgets (buffer, sizeof (buffer), soubor) != NULL)
      zpracuj(buffer);   // něco provádí
    close (r_deskriptory[0]);
  }
  else {                                  // *** parent ***
    FILE *soubor;
    close (r_deskriptory[0]);
    soubor = fdopen (r_deskriptory[1], "w");
    ...  // zápis
    close (r_deskriptory[1]);
  }
  return 0;
}
```

Filtry a standardní vstupy/výstupy. Filtr je takový program, který dokáže pracovat v „rouře“. Má svůj vstup a výstup, který může být přesměrován například pomocí roury. Úkolem filtru je svůj vstup transformovat (například pozměnit, seřadit, něco vyhledat, rozdělit na stránky, vytisknout, přeložit apod.) a pak předat na svůj výstup.

Filtr tedy pracuje se standardním vstupem (*stdin*), standardním výstupem (*stdout*) a standardním chybovým výstupem (*stderr*). Tyto tři předdefinované kanály jsou ve skutečnosti souborové deskriptory, například stdout je definován takto:

```
FILE *stdout;
```

a taky jsou otevřeny pro daný typ komunikace (stdout pro zápis).

Příklad

Pokud chceme něco vypsat na standardní výstup (a řešíme vstupy a výstupy knihovnou `stdio.h`), můžeme to provést takto:

```
fprintf(stdout, "Toto půjde na výstup.\n");
fflush(stdout);
```

Druhý uvedený příkaz vyprázdní výstupní buffer daného deskriptoru. Kdyby tam ten příkaz nebyl, mohlo by se stát, že výstup „počká“, až se v bufferu nashromáždí více dat, a vypíše to až potom. Stává se to u *stdout*, kdežto *stderr* se vypisuje vždy okamžitě.

Pokud použijeme *stdin*, *stdout* a *stderr*, pak bude bez problémů možné v textovém režimu volně přesměrovávat vstupy, výstupy a chybové výstupy.

Pokud programujeme v C++ a používáme knihovnu `iostream.h`, pak se standardním vstupem, výstupem a chybovým výstupem pracujeme pomocí objektů `cin`, `cout` a `cerr`.


Možnosti využití rour jsou rozsáhle probírány jak na mnoha stránkách na internetu, tak i například ve zdroji [36] (najdeme v seznamu literatury).


Sockety. Socket si můžeme představit jako rouru s mnoha vlastnostmi navíc. Také se jedná o používání předem definovaných komunikačních bodů (bran) a jedná se o komunikaci typu klient-server. Komunikace může být spojová (streamová), která pracuje podobně jako roura (posílá se proud dat), anebo datagramová, kdy se nejdřív sestaví balík dat (datagram) a pak se odešle jako celek – dávka.

Po vytvoření a aktivování (obvykle na straně serveru) se získaný identifikátor používá jako deskriptor souboru (vlastně jde o deskriptor). Server na socketu naslouchá, když zjistí, že klient se pokouší o spojení, akceptuje spojení a přijme data. Po ukončení komunukace je nutné socket zavřít a na straně serveru odpojit.


Existuje také třetí typ: *netlink sockets*. Je to IPC rozhraní mezi jádrem a uživatelskými procesy. Adresy netlink socketů jsou PID procesů. Tento typ socketů je používán novějšími programy jako je například sada příkazů `iproute2`.


Sockety jsou v Linuxu implementovány v knihovně `sys/socket.h`. Adresa socketu může být typu `AF_INET`, `AF_INET6` (pro síťové sockety), nebo `AF_UNIX` či `AF_LOCAL` pro UNIX domain sockety (tyto dva typy jsou synonyma), netlink sockety používají typ adres `AF_NETLINK`. Síťové sockety jsou pár IP adresy a čísla portu, UNIX domain sockety jsou deskriptory souborů.


Další informace:

- https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html
- https://www.gnu.org/software/libc/manual/html_node/Sockets.html
- http://www.linuxhowtos.org/C_C++/socket.htm


Zprávy (POSIX Message Queues). Tento mechanismus umožňuje procesům vytvářet vlastní (pojmenované) fronty zpráv. Každá zpráva má určitou prioritu (používá se nejméně 32 úrovní priorit, ale může být i více, v Linuxu až 32 768 úrovní), zprávy s vyšší prioritou jsou doručovány přednostně.

Vytvoření fronty zpráv je obdobou vytvoření souboru (včetně přístupových oprávnění), a s frontami se také zachází podobně jako se soubory nebo spíše s rourami (ale názvy funkcí jsou odlišné).

Proces si své fronty může buď hlídat sám, anebo lze nastavit upozorňování formou signálu anebo přímo zadat obslužnou funkci (ta se spustí automaticky, pokud do fronty dorazí nová zpráva).


Mapování paměti a sdílení. Mapování paměti je určeno pro přenos dat z jakéhokoliv zdroje (v paměti, na disku atd.) do adresového prostoru některého procesu.

Jsou dva typy mapování: soukromé (mmap) a sdílené (shm). Obojí je implementováno v `sys/mman.h`.

Tento koncept je také používán jako základ pro další mechanismy, včetně implementace swapu a sdílení paměti.

Další informace:

- https://www.poftut.com/mmap-tutorial-with-examples-in-c-and-cpp-programming-languages/
- https://gist.github.com/drmalex07/5b72ecb243ea1f5b4fec37a6073d9d23

Systémová volání, knihovna stdlib, RPC. Také v Linuxu máme systémová volání, pomocí kterých mohou procesy komunikovat s jádrem. API jádra je představováno sadou funkcí, jejich volání má za důsledek přepínání mezi uživatelským režimem a režimem jádra. Volající proces pouze obdrží výslednou hodnotu (obvykle nulu nebo kladné číslo v případě úspěchu, záporné číslo pro selhání volání).

Nejdůležitější knihovnou v Linuxu je *standard C Library* (stdlib). Jsou zde definovány některé základní datové typy a konstanty, také makro `NULL` a spousta základních funkcí (například pro konverzi řetězce obsahujícího číslice na číslo, funkce pro práci s dynamickou pamětí, ukončení programu, funkce `system(příkaz)` pro spuštění příkazů operačního systému, práce s náhodnými čísly,...).

RPC (Remote Procedure Call), tedy vzdálené volání funkcí nacházejících se v procesech (či v knihovnách) na jiných počítačích v síti, funguje také v Linuxu. Nicméně funguje trochu jinak než ve Windows.

Další informace:

- https://www.thegeekstuff.com/2012/07/system-calls-library-functions/
- https://www.linuxjournal.com/article/2204

Kapitola 5

Synchronizace procesů

Rychlý náhled: V multitaskovém systému se běžně stává, že více procesů (nebo více vláken téhož procesu) 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 přístupu ke sdíleným prostředkům a metody, kterými je lze řešit.

Klíčová slova: synchronizace, prostředek, konzistentní stav, race-condition, Bernsteinovy podmínky, petriho síť, kritická sekce, producent-konzument, model-obraz, čtenáři-písaři, Pekařův algoritmus, mutex, mutant, spinlock, IRQL, semafor

Cíle studia: Po prostudování této kapitoly porozumíte problematice ochrany prostředků před narušením konzistenčního stavu a naučíte se používat různé synchronizační mechanismy pro tento účel používané.

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 zčásti modifikovaná data. Data jsou v konzistentním stavu před začátkem zápisu a po dokončení zápisu.

Zdroje, které nejsou v konzistentním stavu, je třeba chránit, tedy přístup ke zdrojům musí být synchronizován. Synchronizujeme přístup ke zdrojům:

- mezi vlákny jádra (přístup k hardwaru a různým strukturám v jádře),
- mezi procesy sdílejícími zdroje (například section = sdílenou paměť),
- mezi vlákny téhož procesu (společné proměnné, otevřené soubory, deskriptory socketů, rour, atd.).

Race-Condition (souběh s nejednoznačnými prioritami, „závod o prvenství“) nastává tehdy, když dva procesy (či vlákna) dorazí ke sdílenému prostředku v nerozlišitelném pořadí, tedy nelze spolehlivě a jednoznačně určit, kdo má přednost. Pokud s tím systém nepočítá, pak může nastat situace, kdy je prostředek přidělen oběma žadatelům, nebo žádnému (ale je označen jako přidělený),... Tento problém

může nastat ve víceprocesorovém (vícejádrovém) systému, za určitých okolností také ve víceúlohovém jednoprocesorovém systému.

Race-Condition často nebývá patrná na první pohled. Například se může skrývat za nepředvídaným chováním za určitých (těžko definovatelných) okolností – jenom někdy, nebo za tím, že z neznámých důvodů program dává při každém spuštění trochu jiné výsledky, i když by měl dávat výsledky stejné.

Můžeme například používat atomické proměnné a atomické operace, aby práce s proměnnými byla co nejrychlejší, případně také uzavřít rizikové objekty do kritických sekcí. Systém by při přidělování prostředků měl počítat s tím, že žádost o přístup může přijít zároveň z více stran.

Kritériem pro přístup k prostředkům jsou *Bernsteinovy podmínky*. Označme

- *read(P,t)* množinu všech prostředků, ze kterých se proces P pokouší číst v čase t,
- *write(P,t)* množinu všech prostředků, na které se proces P pokouší v čase t zapisovat.

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, souboru), ať už pro čtení nebo zápis, pokud zde v té chvíli provádí zápis jiný proces.

Bernsteinovy podmínky říkají jen to, čeho je nutné dosáhnout, ale už nic neříkají o tom, jakým způsobem se toho dá dosáhnout. Proto se obvykle používají jiné typy reprezentace přístupu k prostředkům, které přímo určují, jak by celý mechanismus měl fungovat.

V následujícím testu hovoříme obvykle o procesech. V současných operačních systémech se, zvláště v uživatelském režimu, synchronizují spíše vlákna.

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, tedy pro popis synchronizačních úloh.

Petriho síť je orientovaný graf s dvěma typy uzlů:

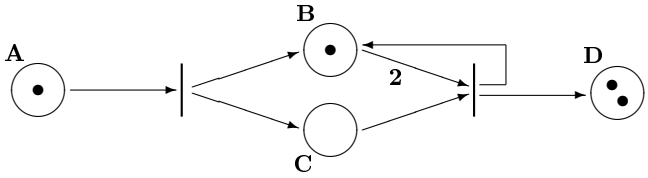

místa, představují stavy procesu nebo stavy systému,


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.

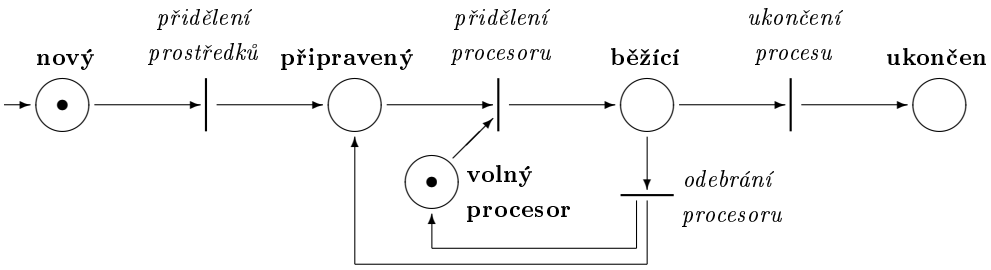


Obrázek 5.1: Příklad Petriho sítě

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 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.

Příklad

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* (jejich vlastní stavy by tvořily cesty souběžné s cesou zobrazeného procesu). 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. V současných operačních systémech se synchronizují spíše vlákna, přesto budeme pro obecnost používat pojem proces.

5.3.1 Kritická sekce

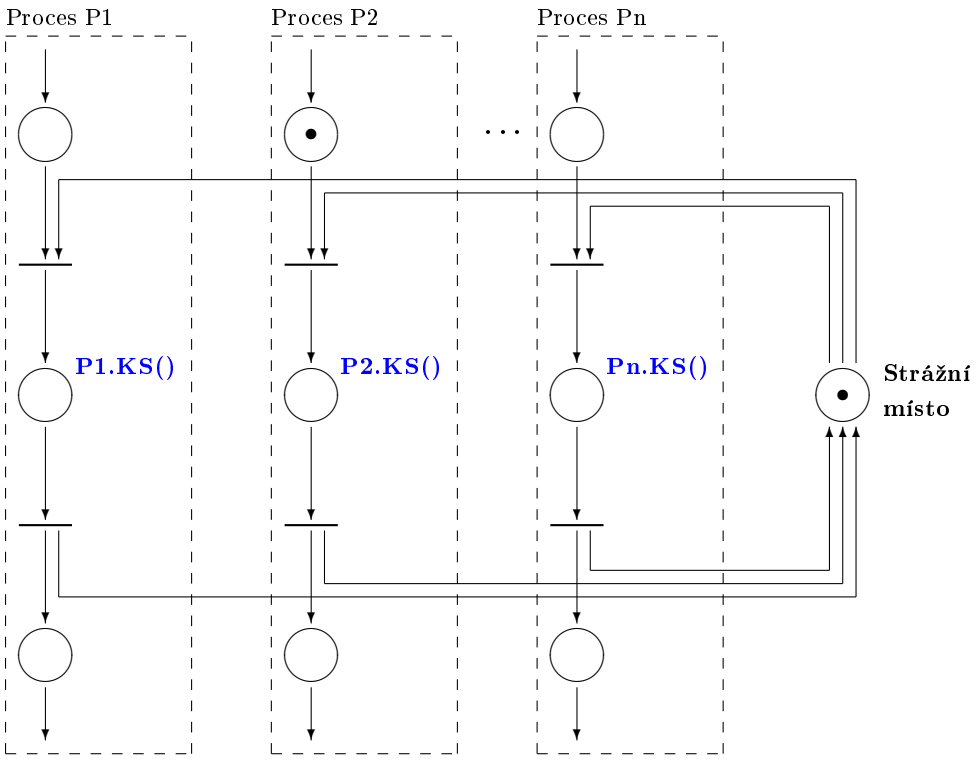
Úlohu typu *kritická sekce* je třeba vyřešit, pokud chceme umožnit výlučný přístup ke sdílenému prostředku – kritické sekci (například sdílenému místu v paměti). Je třeba zajistit, aby k tomuto prostředku v jednom okamžiku přistupoval nejvýše jeden proces a aby tento prostředek mohl bez přerušení využívat po potřebnou nebo předem stanovenou dobu. Kritická sekce je tedy kód ochraňující určený prostředek.

Předpokládejme tedy, že určitý prostředek je sdílen množinou procesů $\{P_0, P_1, \ldots, P_{n-1}\}$. Kód každého z těchto procesů má následující strukturu:

... | Entry section | Critical section | Exit section | ...

Část, která nás zde nejvíc zajímá, je ta modrá – Critical section. Právě v této části kódu jsou instrukce pracující se sdíleným prostředkem. V předchozí části proces prochází přípravou na vstup do kritické sekce, v následující zase odchází (uvolňuje místo jinému procesu).

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:

- data musí být v konzistentním stavu, pokud to proces předpokládá,
- v kritické sekci smí být nejvýše jeden proces (vzájemné vyloučení – mutual exclusion),
- proces se nachází v kritické sekci konečnou dobu,
- proces čeká na vstup do kritické sekce konečnou dobu (závisí na předchozím).

Tato úloha je základem pro další úlohy, v podstatě vždy jde o to – jak zajistit konzistentnost dat, ke kterým přistupuje více různých procesů nebo vláken.

Poznámka:

Jak něco takového naprogramovat? Například *strážní místo* reprezentujeme celočíselnou proměnnou (ideálně atomickou, tedy deklarovanou jako `atomic`), počet teček ve strážním místě bude odpovídat hodnotě této proměnné. V nejjednodušším případě by každý proces před vstupem do této sekce v cyklu testoval proměnnou (dokud je její hodnota 0, cyklus pokračuje), po ukončení cyklu (větší než 0, tj. nejméně jeden token ve strážním místě) by snížil hodnotu proměnné o 1, provedl by kód kritické sekce a následně by zpětně zvýšil hodnotu proměnné o 1.

Tak by to šlo jen v případě, že si procesy navzájem důvěřují (což není zcela běžné, snad jen mezi vlákny téhož procesu). Navíc by mohl nastat problém na systému s vícejádrovým procesorem, protože pak by mohla nastat situace, kdy se dva paralelně běžící procesy (vlákna) pokusí zároveň „sebrat“ poslední token, tedy dekrementovat proměnnou na 0. Způsoby řešení těchto problémů se budeme zabývat v další části této kapitoly.

5.3.2 Producent–konzument

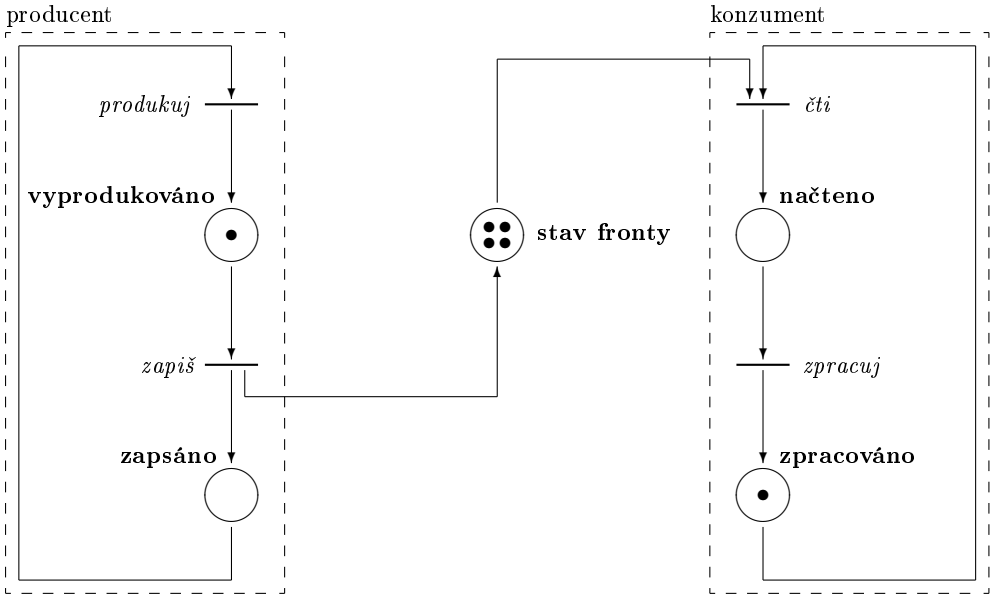
Producent je proces produkující data a *konzument* je proces, který tato data přijímá a dále zpracovává. Účelem je, aby producent (producenti) a konzument (konzumenti) mohli pracovat každý jinou rychlostí, do určité míry na sobě nezávisle, tedy obvykle *asynchronně*.

Dále bude popisován případ s jedním producentem a jedním konzumentem, ale tento případ lze rozšířit na prakticky jakýkoliv počet producentů i konzumentů. Na nákresech by bylo třeba přidat cesty k dalším producentům a konzumentům...

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 (dynamická fronta).
2) Omezený buffer – máme k dispozici určitý počet paměťových míst, je stanovena horní hranice (statická fronta).
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 ve frontě jsou čtyři položky a jsou proveditelné přechody *zapiš* a *čti* (tedy pracovat mohou oba procesy).



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

Jak bychom úlohu rozšířili na víc než jednoho producenta a konzumenta? Jednoduše bychom další producenty a konzumenty napojili na místo *stav fronty* stejným způsobem jako stávající.

Požadavky na řešení:

- zachování konzistence dat,
- konzument se zastaví, když je buffer prázdný,
- zajištění omezeného čekání (podobně jako u kritické sekce).

Poznámka:

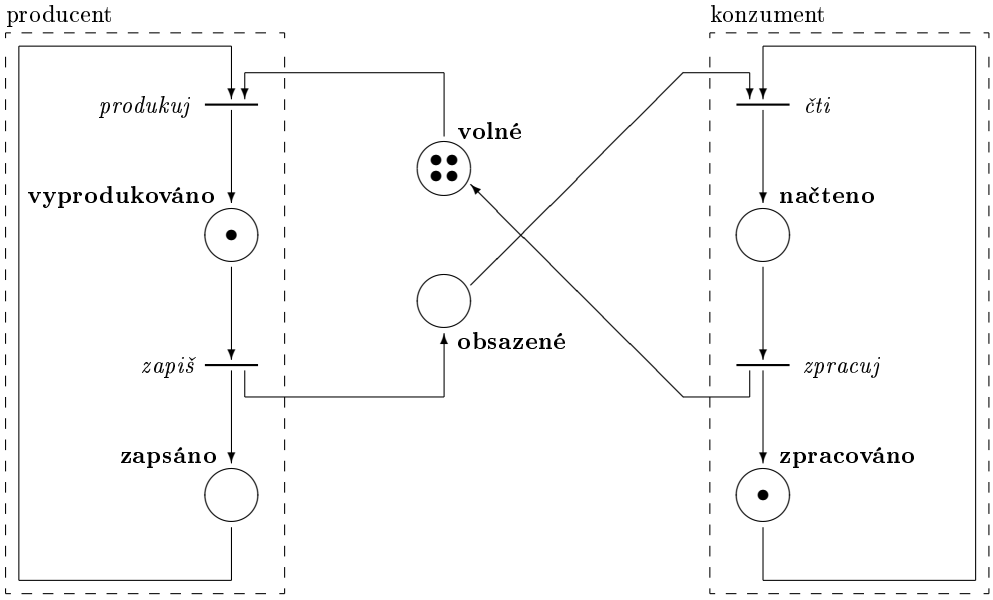
Jak to naprogramovat? Budeme potřebovat celočíselnou proměnnou reprezentující místo *stav fronty*, pak samotnou frontu (třeba jako dynamický seznam, podle toho, co nám příslušný programovací jazyk nabídne).

Producent *nejdřív* uloží nový prvek do fronty a *pak* zvýší hodnotu proměnné o 1 (pořadí je důležité, aby byla zachována konzistentnost dat, hlavně pro případ, že před ukládáním byla fronta prázdná). Konzument *nejdřív* načte (odstraní) prvek z fronty a *pak* sníží hodnotu proměnné o 1 (pořadí je důležité

ze stejného důvodu – zachování konzistence dat, tentokrát při plné frontě, aby se producent nepokoušel přidávat nový prvek do fronty, ve které zatím není místo).

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

Na obrázku 5.5 je proveditelný pouze přechod *zapiš*, konzument musí čekat před přechodem *čti* (není nic ke čtení, fronta je prázdná). 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*.



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

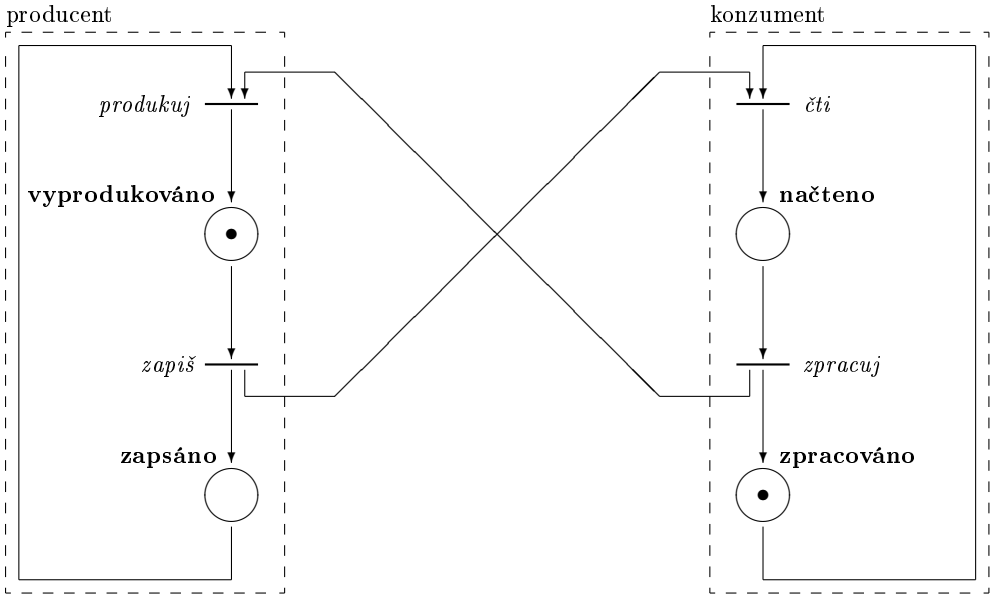
Jsou zde tři požadavky na řešení:

- zachování konzistence dat,
- producent se zastaví, když je buffer plný,
- konzument se zastaví, když je buffer prázdný,
- zajištění omezeného čekání.

Poznámka:

Způsob naprogramování by byl podobný, jen budeme potřebovat dvě proměnné, pro každé sdílené místo jednu, a pak tu statickou frontu (pole apod.). Pořadí operací (ukládání či vyjímání prvku a práce s proměnnými) jsou podobné jako v předchozím případě, účelem je zachování konzistence dat v daném prvku fronty (prvním nebo posledním).

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.



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

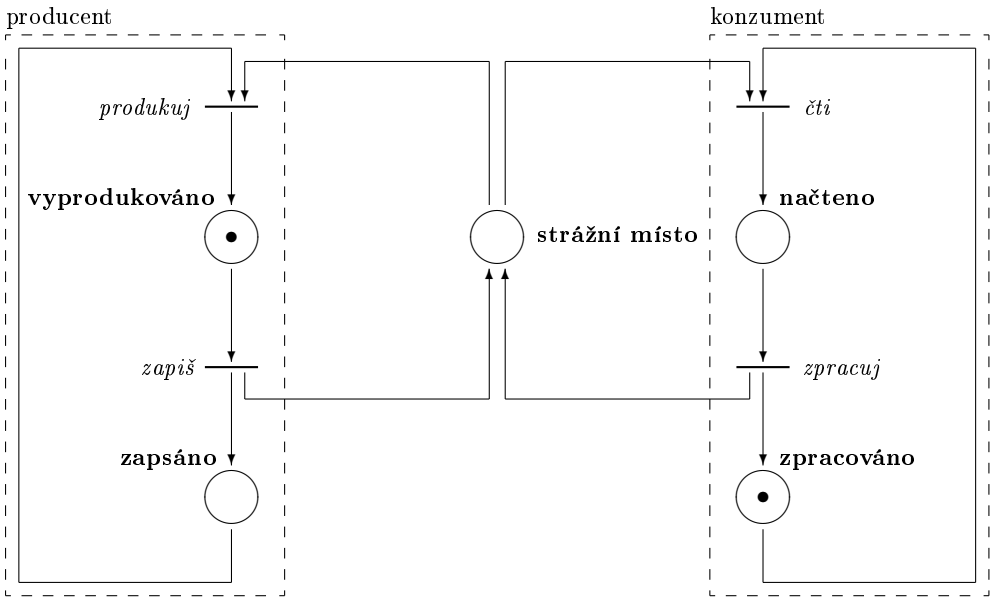
Producent a konzument si navzájem posílají zprávy, producent posílá položky a konzument potvrzení o zpracování (žádost o další položku). Jedná se tedy o symetrickou synchronní komunikaci. Řešení petriho sítí je na obrázku 5.6, procesy na sebe navzájem čekají.

Protože nemáme frontu (sdílený buffer), jediným požadavkem je zachování konzistence dat.

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.

Typické použití je například takové, kdy producent sleduje stav některého čidla (teplota, vlhkost, množství čehokoliv, apod.), zjištěnou hodnotu ukládá do sdílené proměnné (jen jediné, žádná fronta),



Obrázek 5.7: Petriho síť pro úlohu Model–obraz

a konzument v pravidelných intervalech (nebo kdy stíhá) provádí načítání hodnoty této proměnné (vždy má k dispozici její aktuální stav) například pro účely zobrazení nebo spuštění alarmu.

Jedna skupina procesů neustále provádí změny na datech a další skupina procesů zobrazuje aktuální stav těchto dat. 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.

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.

Požadavky na řešení:

- zachování konzistence dat,
- producent se zastaví, když zrovna konzument čte data,
- konzument se zastaví, když zrovna producent modifikuje data.

Poznámka:

Zde by stačil jeden prvek pro data (nemusí být celá fronta), tento prvek by byl producentem neustále přepisován (aktualizován). Dále bychom potřebovali celočíselnou proměnnou pro *strážní místo* (vlastně by stačily jen dvě hodnoty, jako červená a zelená na semaforu, takže klidně typ boolean třeba s názvem *obsazeno*, prokud v programovacím jazyce něco takového máme). Jak producent, tak i konzument, bude v cyklu čekat, dokud v proměnné bude 0 (false), pak ji dekrementuje (nastaví na true), provede příslušnou operaci a následně inkrementuje (nastaví na false).

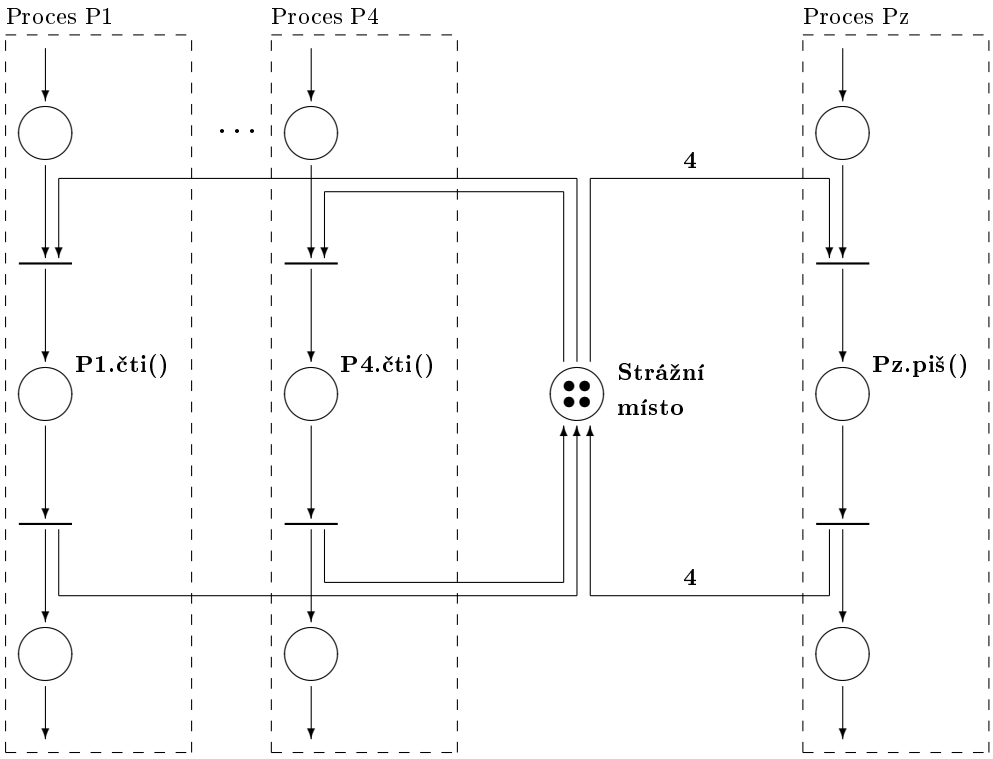
5.3.4 Čtenáři–písaři

V této úloze jsou procesy rozděleny vzhledem k přístupu ke sdílenému prostředku (paměti) do dvou skupin – skupiny *čtenářů* a skupiny *písařů*. 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).

Na obrázku 5.8 je úloha řešena pro čtyři čtenáře a jednoho písaře. Každý čtenář si před čtením vyzvedne token ze strážního místa. Zapisující proces (písař) si při pokusu o zápis musí vyzvednout čtyři tokeny, tedy tolik, kolik je celkem čtenářů. Tím je zajištěno, že zapisovat lze pouze tehdy, když žádný čtenář nečte (ve strážním místě jsou všechny tokeny). Pokud některý písař zapisuje, musí všichni čtenáři počkat, až písař dokončí zápis a vrátí tokeny 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. Ze strážního místa by k němu vedla hrana ohodnocená počtem procesů schopných čtení.



Obrázek 5.8: Petriho síť pro úlohu Čtenáři–písaři

Pokud bychom tento mechanismus využili například ve vícevláknovém procesu pro synchronizaci přístupu k prostředku (třeba proměnné) sdílenému všemi vlákny procesu a nechtěli bychom vlákna klasifikovat na pouze čtoucí a pouze zapisující, ve *strážním místě* by bylo tolik tokenů kolik je vláken. Každé vlákno by při přístupu pro čtení (tj. v kódu těsně před příkazem čtení ze sdíleného prostředku) vzalo jeden token, kdežto při přístupu pro zápis (v kódu těsně před příkazem zápisu) by vzalo plný počet tokenů (podle počtu vláken). Tentýž počet tokenů by po provedení operace vrátilo.

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.

Poznámka:

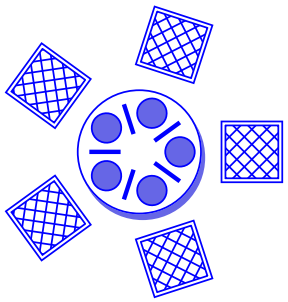
Nyní k implementaci. Nejjendodušší možnost naprogramování by byla opět celočíselná proměnná. Před přístupem do sdílené sekce by proces (vlákno) ve smyčce čekal, než proměnná nabyde hodnoty 1 (pro čtení) nebo maximum (pro zápis), pak je třeba proměnnou dekrementovat o 1 nebo maximum, provést operaci a inkrementovat o 1 nebo maximum.

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 filozof nemá k dispozici hůlku po pravé i levé ruce, nezbývá mu než přemýšlet.

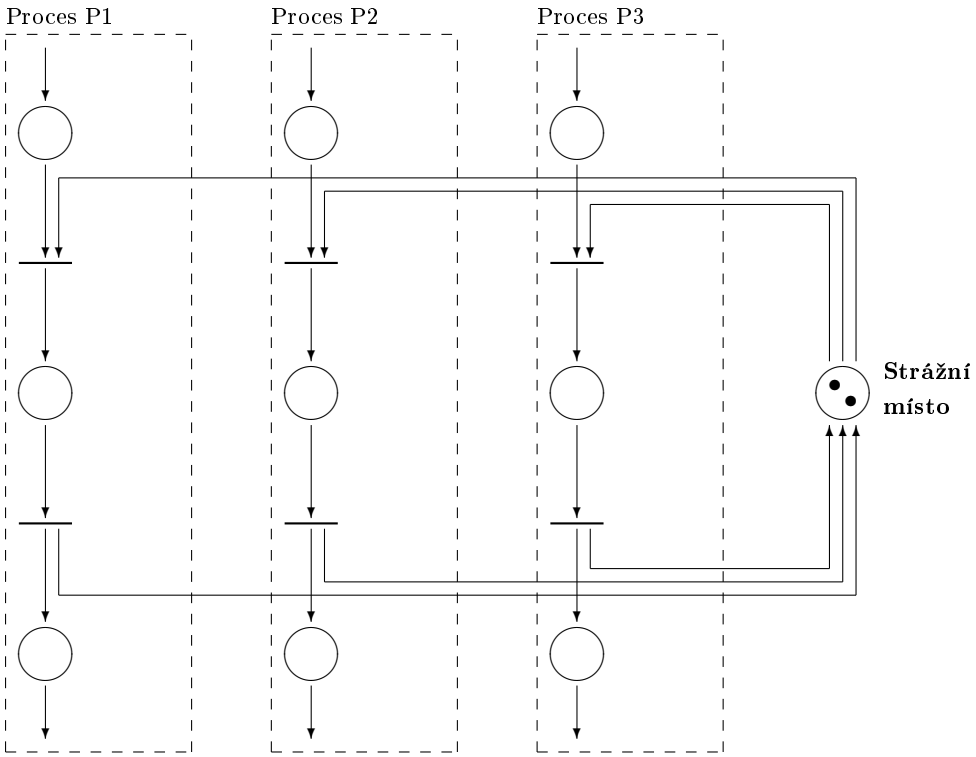
Filozof, jehož sousedé mu střídavě berou hůlky, nemá šanci se najíst, dochází ke *stárnutí procesů* (proces neustále čeká na potřebné zdroje). Pokud všichni najednou zvednou hůlku po své pravé ruce, dojde k uváznutí, protože všichni drží v pravé ruce hůlku a čekají na levou, která je však zrovna držena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



Obrázek 5.9: Pět hladových filozofů

Po aplikaci na procesy máme pět (obecně n) prostředků (hůlek) využívaných pěti (obecně n) procesy (filozofy). Předpokládá se, že tyto prostředky jsou vzájemně zaměnitelné (je jedno, o kterou hůlku konkrétně jde, ať už to zní jakkoliv nehygienicky).

Řešení problému spočívá v tom, že ke stolu nepustíme všech pět filozofů najednou (a tedy k n prostředkům nepustíme všechny procesy najednou), ale maximálně čtyři ($n-1$). 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 je pak pro další proces. Jinou možností je nařídit jednomu filozofovi, aby bral hůlky v opačném pořadí než ostatní (což u procesů není aplikovatelné).



Obrázek 5.10: Petriho síť pro úlohu Pět filozofů (pro tři procesy a tři prostředky)

Na obrázku 5.10 je řešení naznačeno na skupině tří procesů a tří prostředků. Je dovoleno najednou pracovat pouze dvěma procesům. Obecně je množství procesů irelevantní, důležité je, že ve *strážním místě* máme maximálně o 1 token méně než kolik je prostředků.

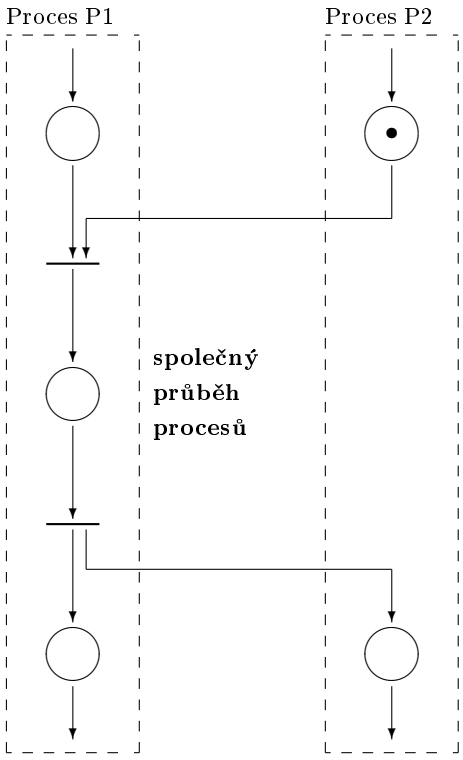
Požadavky na řešení:

- v jednom okamžiku může být přiděleno jen $n-1$ prostředků (celkem je jich n),
- zachování konzistence dat (tj. je důležité, v jakém pořadí se provádějí operace a práce s číselnou proměnnou).

5.3.6 Souběh procesů

Úloha souběhu procesů je řešena v paralelním systému, kdy je třeba synchronizovat činnost dvou procesů běžících na různých procesorech nebo na různých uzlech v síti (tedy souběžně pracujících procesů). Tyto procesy je třeba sesynchronizovat tak, aby určitou část kódu prováděly společně.

Pokud se jeden proces ke společné části kódu dostane dřív než druhý, musí počkat (na obrázku 5.11 musí čekat proces P2), tento kód lze provést až ve chvíli, kdy k přechodu na začátku společné části dospějí oba procesy.



Obrázek 5.11: Petriho síť pro úlohu Souběh procesů

Tato metoda synchronizace procesů se může používat například v mechanismu RPC (volání vzdálené procedury, na obrázku 5.11 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 a aktivní čekání.. Pasivně čekající proces nedostává přidělen procesor (tj. je suspendován, blokován), aktivně čekající proces dostává přidělen procesor buď na „čekací smyčku“ nebo na provádění jiné činnosti.

5.4.1 Pasivní čekání

Při pasivním čekání je proces blokován nebo suspendován, sám se nepodílí na čekání. Dále probereme různé možnosti implementace pasivního čekání před kritickou sekcí.

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 hardwarově závislé 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.

Zákaz přepnutí kontextu. 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, reálně se degraduje multitasking (preemptivní se mění na nepreemptivní formu).

Navýšení priority. Některé operační systémy řeší omezení přepnutí kontextu poněkud elegantnějším způsobem. Pokud proces využívá sdílený prostředek, k němuž musí být synchronizován přístup, je tomuto procesu priorita zvýšena nad úroveň všech procesů, které mají oprávnění o tento prostředek žádat. Proces má na procesoru vždy přednost před všemi procesy, které mají nižší prioritu, čímž je zaručen jeho výhradní přístup při využívání prostředku.

Ovšem může docházet ke stárnutí procesů, pokud systém nemá mechanismus kontroly doby strávené procesem na procesoru.

Mutex (mutual exclusion, vzájemné vyloučení). Mutex je mechanismus zamknutí prostředku. Pokud proces požádá o průchod mutexem (resp. pokusí se mutex uzamknout) a přitom je mutex právě uzamknutý, tento proces bude suspendován (odložen mezi čekající procesy).

Mutex nemusí být jen pasivní metodou čekání, v některých operačních systémech existuje i varianta, kdy proces pravidelně testuje stav mutexu a mezi testováním může provádět svůj kód.

5.4.2 Aktivní čekání

Při aktivním čekání se proces podílí na čekání (nebo provádí jinou činnost podle svého kódu), může jít o některou variantu cyklu s prázdnou operací.

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.

Příklad

Podíváme se na implementaci sdílené zamykací proměnné.

```
shared int obsazeno = 0;
...
// uvnitř procesu:
while (obsazeno) {};       // čekáme s prázdným cyklem, dokud je obsazeno
// už nečeká:
obsazeno = 1;              // rezervujeme si prostředek
KS();                      // používáme prostředek (kritická sekce)
obsazeno = 0;              // uvolnili jsme prostředek
```

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).

Tento mechanismus může selhat na systému s více procesory nebo s vícejádrovým procesorem – může nastat situace, kdy dva paralelně běžící procesy zároveň „zjistí“, že je kritická sekce volná, a zároveň se tedy pokusí přenastavit proměnnou `obsazeno`.

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).

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.

Příklad

Implementace pekařova algoritmu je následující:

```
shared int poradi[n] = (0,0,...,0);
shared int prirazuje[n] = (0,0,...,0);
...
prirazuje[i] = 1;                          // po dobu přiřazování pořadí budeme "chráněni"
poradi[i] = DejNejvyssi(poradi) + 1;       // získáme pořadové číslo
prirazuje[i] = 0;                          // konec přiřazování pořadového čísla

for (j=0; j<n; j++) {                      // zjišťujeme, které procesy mají přednost
  while (prirazuje[j]) {};                     // j-tému procesu je přiřazováno pořadové číslo
  while ((!poradi[j]) &&                       // j-tý proces žádá o tentýž prostředek
         ((poradi[j]<poradi[i]) ||                 // je před námi
          ((poradi[j]==poradi[i])&&(j<i)))) {};    // stejné číslo, ale menší PID
}
KS();
poradi[i] = 0;          // o prostředek už nežádáme
```

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.

Hardwarové řešení. Některé procesory nabízejí hardwarové řešení pro aktivní čekání, instrukci `TSL` (Test and Set Lock), `swap` nebo `XCHG` (na různých hardwarových architekturách).

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.

Postup

Ukážeme si způsob využití instrukce `XCHG`.

```
// proměnná zamek je sdílenou zamykací proměnnou, když = 1, je zamčeno
KS: mov EAX, 1h             // do registru EAX uložíme hodnotu 1
    xchg zamek, EAX         // voláme instrukci, která přehodí obsah parametrů
    jnz KS                  // cyklus -- Jump if Not Zero (dokud se do EAX nedostane 0)

    // pokud při výměně byla v zámku původně 0, přehozením se tam dostane 1, tedy
    //   odemknutý zámek okamžitě znovu zamkneme a pokračujeme za cyklem svým kódem
...                         // používáme prostředek
odchod: mov zamek, 0h       // při svém odchodu z kritické sekce odemkneme
```

Operační systémy také obvykle nabízejí systémová volání (funkce) zapouzdřující podobnou instrukci, protože obecně v současných operačních systémech má programátor jen omezený přístup k instrukcím procesoru.

Příklad

Při programování se v různých jazycích můžeme setkat se systémovým voláním `swap`:

```
shared int zamek = 0;
...
// v kódu procesu:
int kontrola;                   // kontrolní proměnná pro výměnu
...
kontrola = 1;
while (kontrola = 1)
  swap (zamek, kontrola);
                                // kontrola má hodnotu 0, konec čekání:
...                             // kritická sekce, používáme prostředek
zamek = 0;                      // konec kritické sekce
```

5.5 Synchronizační nástroje operačního systému

Prostředky popsané výše v podkapitole 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 a mutexy,
- 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 kapitoly 5.3 se používá jeden nebo více semaforů.

Semafor samotný si můžeme představit jako tuto datovou strukturu:

```
typedef struct TSemafor {
  int stav;                    // stav semaforu (volno, obsazeno, apod.)
  TFrontaProcesu fronta;       // fronta, ve které čekají procesy
}
extern struct TSemafor semafor;
```

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, semafor je nastaven na „červenou“,
- 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 (když je fronta prázdná, nastaví semafor na „zelenou“).

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);          // žádáme o prostředek, pokud není volný, jdeme do fronty
KS();                   // pracujeme s prostředkem
signal(semafor);        // prostředek vrátíme a pokračujeme ve svém kódu
...
```

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].

Existují dva typy semaforů – binární a obecné.

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 (protože je obsazeno).

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 si můžeme představit jako strukturu obsahující čítač, který může nabývat různých celočíselných hodnot.

Z hlediska procesu se obecné semafory používají stejně jako binární, ale 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

[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.

projít bez čekání. Hodnota 0 má stejný význam jako u binárních semaforů, tedy prostředek je některým procesem využíván (červená).

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.

Příklad

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

```
extern struct TSemafor  volne, obsazene;
```

Význam těchto semaforů je následující:

- 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 (volne);
  signal (obsazene);
} while (1);
```

Konzument:

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

Postup

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                  // počet sdílených prostředků nastaven na 5

struct TSemafor sem[N];        // semafory pro hlídání prostředků (hůlek)
struct TSemafor S;             // semafor pro hlídání počtu procesů

for (i=0; i<N; i++)            // inicializace, zatím žádný prostředek není používán
  sem[i].stav = 1;
S.stav = N-1;                  // předplacení semaforu podle počtu prostředků

// pro i-tý proces:
do {
  mysli(i);                    // i-tý proces zatím prostředky nepotřebuje
  wait (S);                    // už ano, tedy snížíme předplacení počtu procesů
  wait (sem[i]);               // rezervujeme jeden prostředek (hůlku)
  wait (sem[(i+1)%5]);         // ještě další
  jez (i);                     // máme oba, tedy je používáme
  signal (sem(i+1)%5]);        // vrátíme oba prostředky, ale v obráceném pořadí
  signal (sem[i]);
  signal (S);                  // dáme šanci dalšímu procesu
} 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 či distribuované prostředí včetně synchronizace v rámci počítačové sítě. Tomu musí být přizpůsobena adresace komunikujících procesů či objektů.

Procesy se zprávami pracují pomocí funkcí (systémových volání) obvykle nazvaných `send` a `receive`. Při použití přímého adresování komunikace funguje výše popsaným způsobem (kapitola 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.

Příklad

Následující ukázka 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                    // maximální počet zpráv ve schránce
struct TSchranka  volne, obsazene;  // schránky
for (i=0; i<MAX; i++)               // inicializace, předplacení
  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). Volání funkce `send` nebo `receive` způsobí zablokování volajícího procesu, proces je odblokován tehdy, když je volána párová funkce, tedy až když jsou volány obě funkce `send` a `receive`, může komunikace proběhnout (vzájemné čekání).

Příklad

Následující kód je řešením úlohy Producent–konzument bez sdílené paměti. Oproti předchozím řešením musí být komunikace synchronní. Všimněte si, že nedeklarujeme žádné sdílené proměnné ani datové struktury.

Producent:

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

Konzument:

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

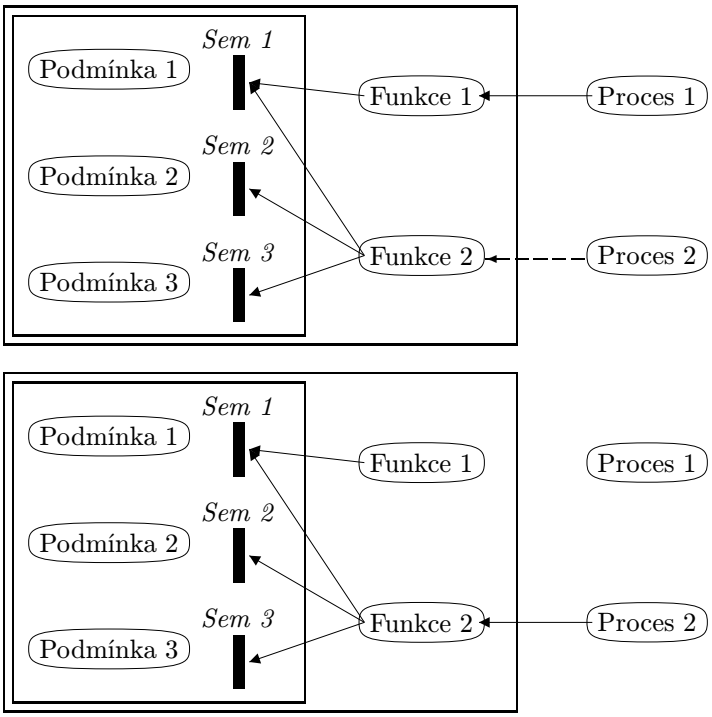
5.5.3 Monitory

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`.

Na obrázku 5.12 nahoře je jednoduchý monitor, ve kterém první proces volá funkci 1. Tato funkce uzamkne (nastaví na červenou) jeden ze semaforů, a tedy znemožní volání druhé funkce (tu volá druhý proces). Druhá funkce čeká, až bude tento semafor odemknut, a až pak může provést svůj kód (ke své činnosti potřebuje pro sebe uzamknout všechny tři semafory), což vidíme na tomtéž obrázku dole.



Obrázek 5.12: Jednoduchý monitor se dvěma přístupovými funkcemi

5.6 Synchronizační nástroje v různých operačních systémech

Z hlediska programátora jsou zajímavé především synchronizační mechanismy pro synchronizaci přístupu k objektu sdílenému vlákny jednoho procesu, případně zajištění sdílení objektu přes několik „příbuzných“ procesů.

V jakémkoliv operačním systému si programátor může vytvořit jednoduchou sdílenou proměnnou, kterou budou vlákna testovat při přístupu ke sdílenému objektu. To je ovšem spíše primitivní metoda, která není vždy bez problémů, zvláště na víceprocesorovém systému. Také není problém naprogramovat kterýkoliv z algoritmů, které byly popisovány výše v této kapitole.

5.6.1 Windows

Ve Windows máme k dispozici tyto synchronizační prostředky:

- maskování přerušení, v současných verzích ve formě IRQL (klasické maskování přerušení bylo použitelné jen na jednoprocesorovém systému),
- otočný zámek (spinlock), funguje i na víceprocesorových systémech,
- mutexy a semafory (souhrnně nazývané dispatcher objects),
- události (podmíněné proměnné), atd.

Úrovně přerušení IRQL. Maskování přerušení pomocí IRQL je mechanismus zastřešující různé typy událostí často napojené na přerušení (IRQ). Dělí je do tzv. úrovní (levels) – *IRQL* (Interrupt Request Level).

Zde pozor – mechanismus IRQL je něco trochu jiného než IRQ a priority procesů, je na vyšší úrovni. V tabulce 5.1 jsou existující *úrovně IRQL*. Ovšem ve skutečnosti se číslování a obsah jednotlivých úrovní liší na různých architekturách (z používaných x86, amd64).

	31	Vysoká („catastrophic errors“)	Hardwarová přerušení
	30	Selhání napájení	
	29	Meziprocesorové přerušení	
	28	Hodiny (timer)	
	27	Synchronizační mechanismy	
	⋮	Přerušení od zařízení (běžná IRQ)	
	2	DPC (Deferred Procedure Call), plánování CPU	Softwarová přerušení
Pro priority vláken 0–31	1	APC, Page Fault	
	0	Nízká: uživatelské procesy a část operací jádra	

Tabulka 5.1: Úrovně IRQL

Uživatelské procesy (běžící v user mode) s běžnou prioritou mají IRQL 0 (to znamená, že běží na IRQL 0). Vlákna jádra provádějící volání APC jsou na IRQL 1, vyšší IRQL souvisejí pak s dalšími činnostmi v jádře prováděnými v souvislosti s ošetřením většinou hardwarových přerušení.

Úroveň „vysoká“ je použita pro ukončování systému většinou v důsledku vážné chyby v jádře (BSOD), proces ukončování má před vším ostatním přednost. Úroveň 30 je sice dokumentována, ale nepoužívá se, teoreticky by se do ní mělo přecházet při výpadku proudu. Úroveň 29 slouží ke komunikaci s jinými procesory. Úroveň 28 je naopak používána běžně – pro aktualizaci systémových hodin a obecně pro různé činnosti související s přidělováním času. Profilování (IRQL 27) je metoda vzorkování stavu vykonávání procesu, je tedy možné sledovat činnost vybraného procesu. IRQL 3–26 jsou určena pro priorizaci přerušení od zařízení.

Při použití mechanismu IRQL lze maskovat vždy všechna IRQL až do určité úrovně. Například pokud jsou maskována IRQL do 27, tak jsou ignorovány požadavky všech běžných procesů (IRQL 0), volání APC (IRQL 1), atd., až do úrovně 27, vyšší čísla IRQL již nejsou maskována. Z toho vyplývají přednosti zpracování – APC má přednost před běžným procesem, hardwarová přerušení mají přednost před softwarovými. Po většinu času běhu systému je používána IRQL 0.

Procesory (jádra) si při výskytu přerušení vyžádají index do výše uvedené tabulky. Tento index jim říká, až po kterou úroveň jsou přerušení maskována, tedy která se mají ignorovat.

Další informace:

Vše výše uvedené platí pro 32bitový systém. Na 64bitovém systému vypadá tabulka IRQL trochu jinak (paradoxně má méně úrovní – jen do 15). Podrobnější informace získáte například na https://blogs.msdn.microsoft.com/doronh/2010/02/02/what-is-irql/.

Spinlock (otočný zámek). Otočné zámky se používají pouze v jádře k synchronizaci ve víceprocesorovém systému a svým založením jsou vlastně mutexy. Mají dva stavy – volno a obsazeno (zamčeno a odemčeno). Samotný otočný zámek je velmi jednoduchá struktura a používá se vždy zároveň s jinou složitější globální datovou strukturou, jejíž konzistentnost chrání, typicky frontou volání procedur nebo datovými strukturami ke konkrétnímu zařízení.

Například pokud ve víceprocesorovém systému procesor vybírá z fronty požadavků na volání procedur (DPC) položku, aby mohla být spuštěna, tato fronta musí být během vybírání vlákna uzamknuta spinlockem a odemknuta až po dokončení vyjmutí z fronty, kdy je tato fronta v konzistentním stavu.

Nebo pokud ovladač zařízení, který právě běží na jednom procesoru, pracuje s datovými strukturami svého zařízení (například posílá data na vstup zařízení), tyto struktury uzamkne, protože ve stejnou chvíli by jiná část tohoto ovladače běžící na jiném procesoru také mohla s těmito strukturami chtít pracovat.

Spinlocky v jádře obvykle mají přiřazenu úroveň IRQL 2 (DPC) nebo vyšší.

Mutexy a semafory. Semafor je ve Windows vnímán jako mutex, který umožňuje předplacení (resp. naopak mutex může být brán jako binární semafor). Jsou vždy spojeny s konkrétním objektem, k němuž je synchronizován přístup. V uživatelském prostoru jsou typicky používány pro synchronizaci činnosti vláken v jednom procesu.

Mutex se vytvoří voláním funkce `CreateMutex()`, jejíž návratová hodnota (typu DWORD) je identifikátor vytvořeného mutextu. Obvykle se používá pro synchronizaci přístupu k místu sdílenému vlákny téhož procesu. V každém vlákně používajícím sdílené místo se před kritickou sekcí zavolá funkce `WaitForSingleObject()`, podle jejíž návratové hodnoty vlákno zjistí, jestli „má právo čekat“ (může být vrácena hodnota `WAIT_ABANDONED`, což znamená, že mu bylo odepřeno právo mutex používat). Touto funkcí dává vlákno najevo, že chce získat kontrolu nad mutexem. Příkaz `ReleaseMutex()` slouží k ukončení kontroly nad mutexem, tedy opuštění kritické sekce.

Se semafory se zachází podobně, jen se v názvech funkcí místo řetězce „Mutex“ objevuje řetězec „Semaphore“ a parametrů je více.

Další informace:

Příklad použití mutextu v C++ pro synchronizaci přístupu vláken k databázi najdeme na
https://learn.microsoft.com/en-us/windows/win32/sync/using-mutex-objects
Je tam naznačeno jak vytvoření a použití mutexu, tak i vytvoření vlákna aplikace. V menu v levém panelu pak najdeme podobné příklady pro další synchronizační objekty či metody.

V režimu jádra se místo pojmu „mutex“ také používá pojem „mutant“. Existuje několik druhů mutexů (například rychlé mutexy), těmto odlišnostem se však již nebudeme věnovat.

Kritická sekce. Zřejmě nejjednodušším mechanismem je *kritická sekce*. Nejde o objekt, tedy inicializační funkce nevrací handle.

Kritické sekce jsou určeny pro synchronizaci jednoduchých objektů či proměnných, například čítačů.

Příklad

Aby bylo jasné, v jakém pořadí se volají funkce pracující s kritickou sekcí:

```
CRITICAL_SECTION CriticalSec;     // pomocná proměnná pro určení stavu KS

int main() {
  ...
  if (! InitializeCriticalSectionAndSpinCount(&CriticalSec, 0x400))
    hlaseni_chyby(...);
  ...
  EnterCriticalSection(&CriticalSec);   // vstup do kritické sekce
  ...                                   // kód kritické sekce
  LeaveCriticalSection(&CriticalSec);   // odchod z kritické sekce
  ...
  DeleteCriticalSection(&CriticalSec);
}
```

Event. Lze vytvořit událost související prakticky s čímkoliv u různých objektů, na tuto událost se pak napojují vlákna (čekají na výskyt události se zadanými parametry). Na rozdíl od kritické sekce je Event blíže mutexům a semaforům, jedná se objekt a v obslužných funkcích používáme handle tohoto objektu.

Ve Windows lze také čekat na ukončení konkrétního procesu či vlákna, na I/O operaci (obvykle souvisí s používáním souborů), na časovač (Timer), atd.

5.6.2 Linux

Sice to nebylo výše rozebíráno, ale v Linuxu existují také objekty. K objektům jádra patří kromě jiného také synchronizační objekty jako je například mutex.

Mutex a futex. Základním synchronizačním objektem je mutex. Mutexy jako takové jsou objekty jádra, ale mohou být exportovány do uživatelského prostoru, kde se jim říká *futex* (fast mutex, rychlý mutex). Futexy jsou reprezentovány atomicky měnitelnou proměnnou (tj. deklarovanou jako atomic), účelem existence této proměnné je urychlit zjišťování momentálního stavu mutexu (jinak by se zjišťování muselo provádět systémovými voláními, která jsou časově náročná).

Na futexech je založena většina ostatních synchronizačních mechanismů, atomická proměnná futexu je praktické a rychlé řešení. Implementaci najdeme v knihovně `libthread` (protože se v uživatelském prostoru obvykle synchronizují vlákna jednoho procesu).

Mutexy lze použít pro aktivní i pasivní čekání.

Postup

Ukážeme si, jak vytvořit mutex (futex) pro synchronizaci vláken a jak ho používat. Předpokládejme, že je načtena potřebná knihovna – `libthread`.

```
pthread_mutex_t  mujmutex;                          // datový typ pro mutexy

if (pthread_mutex_init (&mujmutex, NULL) != 0) {
  ...  // ošetření chyby při vytváření mutexu
}
pthread_mutex_lock (&mujmutex);                     // zamknutí mutexu
...                                                 // kód v "kritické sekci"
pthread_mutex_unlock (&mujmutex);                   // odemknutí mutexu
...
pthread_mutex_destroy (&mujmutex);                  // mutex už nepotřebujeme
```

Pokud při volání zamykací funkce je mutex zamknutý jiným vláknem, místo opětovného zamknutí proces přechází do pasivního čekání, je suspendován.

Předpokládejme, že chceme využívat aktivní čekání. Pak místo volání uzamykací funkce budeme pouze testovat stav mutexu a podle návratové hodnoty poznáme, zda byl odemknutý (pokud byl, tak ho tato testovací funkce rovnou uzamkne, tentokrát pro nás):

```
...   // deklarace, inicializace
if (pthread_mutex_trylock (&mujmutex) == 0) {     // je odemknuto?
  ...                                             // kód v "kritické sekci"
  pthread_mutex_unlock (&mujmutex);               // odemknutí mutexu
} ...
```

Mutexy jsou v Linuxu rozsáhle konfigurovatelné. Existuje například verze pro zamykání s časovým limitem (objekt je vždy uzamknut na zadaný časový interval), verze nerekurzívní, která dovoluje více-

násobné uzamknutí mutexu, robustní mutex schopný fungovat i po ukončení vlákna, které ho uzamklo a pak už neodemklo, atd.

Priority. Jednou z vlastností mutexu je také možnost nastavení *stropu priorit*. Priorita procesu (vlákna), který provedl uzamknutí, může být dočasně zvýšena nad úroveň všech ostatních procesů (vláken), které se přihlásily k používání tohoto mutexu. Funkce `pthread_mutex_getprioceiling()` slouží ke zjištění stropu priorit pro daný mutex, obdobná funkce (`set` místo `get`) tento strop mění.

Rwlock. Tento mechanismus umožňuje rozlišit mezi uzamknutím pro čtení a uzamknutím pro zápis, je to tedy obdoba řešení úlohy *Čtenáři–písaři*.

Postup

Ukážeme si použití zámku typu rwlock. Všimněte si rozdílu v uzamykání pro čtení či zápis. Uzamykání pro čtení se musí provádět

ve smyčce, protože počet možných uzamknutí jednoho zámku pro čtení je omezen a vlákno tedy musí tak dlouho uzamykat, dokud nebude „propuštěno“ dál do kódu, který je takto chráněn, nebo suspendováno. U zamykání pro zápis to není potřeba, protože tento typ uzamknutí může provést jen jedno vlákno (ostatní jsou hned suspendována).

```
pthread_rwlock_t  zamek;

if (pthread_rwlock_init (&zamek, NULL) != NULL) {
  ...                                   // ošetření chyby při vytváření zámku
}

// zamykáme pro čtení (read -> rd):
if (pthread_rwlock_rdlock (&zamek) == EAGAIN) {}
...                                     // používáme pro čtení
pthread_rwlock_unlock (&zamek);

// zamykáme pro zápis (write -> wr):
pthread_rwlock_wrlock (&zamek);
...                                     // používáme pro zápis
pthread_rwlock_unlock (&zamek);
...
pthread_rwlock_destroy (&zamek);
```

Spinlock. Je to synchronizační objekt používaný spíše v jádře, a představuje možnost aktivního čekání. Nedoporučuje se ho používat příliš často (mutexy jsou ve většině případů lepší), je určen spíše pro zajišťování meziprocesorových operací (ostatně jako ve Windows). Se spinlockem se zachází podobně jako s mutexem, jen se v názvech funkcí místo řetězce „mutex“ vyskytuje řetězec „spinlock“ a při čekání je nutné použít smyčku (například s prázdným příkazem, i když obecně tam může být jakákoliv sada příkazů).

Bariéra. Jde o jednoduchý synchronizační objekt, který se moc nepoužívá. Slouží nikoliv k synchronizaci přístupu k objektu, ale spíše k synchronizaci dosažení určitého bodu v kódu. Bariéra je uzamčena až do chvíle, kdy zadaného bodu ve svých kódech dosáhnou všechna synchronizovaná vlákna. Typické použití je při potřebě doběhnutí všech vláken k bodu, ze kterého mají pokračovat společně, případně ve kterém mají provést některý společný kód (vzpomeňte si na synchronizační úlohu „Souběh procesů“).

Podmínková proměnná (událost). Podmínkové proměnné (condition) jsou obdobou událostí (event) ve Windows. Používáme je tehdy, když chceme probuzení jednoho nebo více stanovených vláken podmínit splněním určité podmínky. Příslušná proměnná je vláknem otestována, a pokud má hodnotu „nesplněno“, testující vlákno je automaticky suspendováno.

Protože musí být zajištěna konzistence této podmínky, při každém přístupu k ní včetně testování její hodnoty a také změny při splnění podmínky je nutné používat mutex.

Semafor. Semafory jsou zobecněné mutexy. Ale zatímco všechny výše zmíněné synchronizační mechanismy přišly do Linuxu ze standardu POSIX, semafory pocházejí z System V, proto se zde setkáváme s jinou syntaxí.

Existují dva druhy semaforů – pojmenované a anonymní (nepojmenované).

Postup

Ukážeme si práci s pojmenovaným semaforem. Semafor je třeba nejen deklarovat, ale i inicializovat, také musíme počítat s případnou chybou při inicializaci.

```
// deklarujeme a inicalizujeme semafor, předplacení na hodnotu 5:
sem_t *semafor = sem_open ("/mujsemafor", O_CREAT, S_IWUSR | S_IRUSR, 5);
if (semafor == SEM_FAILED) {
  ...                           // ošetření chyby při inicializaci semaforu
}

if (sem_wait (semafor) == 0) {
  ...                           // kód, který chceme provádět s chráněným objektem
  sem_post (semafor);           // konec činnosti v chráněné oblasti
}

// uzavření a odpojení semaforu:
sem_close (semafor);
sem_unlink ("/mujsemafor");
```

Pojmenované semafory mají své jméno, které jsme uvedli jako parametr při vytváření a odpojování. Toto jméno je vlastně speciální soubor, který bychom našli v adresáři `/dev/shm`.

Postup

Anonymní semafory jsou vlastně nástavba nad sdílenou oblastí paměti. Lze takto řídit také dynamicky alokovanou paměť (resp. přístup k ní), vlákna by měla opět znát jak sdílenou paměť, tak i semafor. V příkladu je semafor opět předplacen na hodnotu 5.

```
// vytvoříme paměť, která bude základem pro semafor:
void *pamet = ......                    // nějaká vhodná funkce pro alokaci
// deklarujeme a inicalizujeme semafor:
sem_t *semafor = pamet;
if (sem_init (semafor, 1, 5) != 0) {
  ...                                   // ošetření chyby
}

if (sem_wait (semafor) == 0) {
  ...
  sem_post (semafor);
}

sem_destroy (semafor);                  // uzavření  semaforu
...                                     // případné uvolnění alokované paměti
```

Vidíme, že použití anonymního semaforu je podobné, liší se jen ve způsobu používání. Všimněte si způsobu propojení semaforu s pamětí při deklaraci.

Všechny výše popsané synchronizační objekty lze *sdílet* nejen mezi vlákny jednoho procesu, ale také mezi procesy. Aby to bylo možné, je potřeba podle toho nastavit atributy daného objektu (povolit sdílení) a umožnit jiným (vybraným) procesům přístup do synchronizované paměti. Popis těchto technik je však nad rámec tohoto textu.

Všechny programovací techniky, které jsme si zde popsali, jsou určeny pro synchronizaci v uživatelském režimu (kromě spinlocku). V jádře se také používají mutexy, semafory a další synchronizační mechanismy, ale s využitím jiných datových struktur a funkcí.

V jádře se také používají další mechanismy, které nejsou použitelné v uživatelském režimu, jako sekvenční zámky, RCU (Read-Copy-Update, pro data, která jsou často čtena, ale málo se mění), Completion (čekání na ukončení některé úlohy), atd.

Další informace:

- http://www.informit.com/articles/article.aspx?p=2085690&seqNum=6
- https://docs.oracle.com/cd/E26502_01/html/E35303/sync-11157.html
- https://0xax.gitbooks.io/linux-insides/SyncPrim/

Kapitola 6

Uváznutí procesů – Deadlock

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

Uváznutí je samozřejmě nežádoucí, proto je vhodné buď navrhnout systém tak, aby nemohlo nastat, nebo této situaci předcházet pokusy o předpovídání uváznutí, anebo, pokud nastane, ji řešit co nejšetrněji vzhledem k systému i procesům.

Klíčová slova: uváznutí (deadlock), čekání, třída prostředků, bezpečný stav, graf přidělení prostředků, kruhové čekání, graf nároků a přidělení prostředků, Bankéřův algoritmus, graf čekání.

Cíle studia: Po prostudování této kapitoly porozumíte mechanismům vzniku uváznutí, prevence jeho vzniku, předpovídání a detekce uváznutí.

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 a tedy zaměnitelné, třídě *čas CPU* příslušejí jako instance časové cykly procesoru, které mohou být procesům přidělovány, ...

Používání prostředku některým procesem lze rozdělit do několika fází:

1. *Žádost* – proces požádá o přidělení prostředku, obvykle formou systémového volání. To může dopadnout následovně:
   (a) přidělení je povoleno, tedy proces obdrží požadovaný prostředek, pokračuje bodem 2.
   (b) přidělení není povoleno, proces musí čekat (formou aktivního nebo pasivního čekání).

2. *Používání* – proces používá přidělený prostředek.
3. *Uvolnění prostředku* – proces už prostředek nepotřebuje (nebo prostředek musí být z nějakého důvodu odebrán), tento prostředek může být přidělen jinému procesu.

Definice

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

6.2 Popis stavu přidělení prostředků

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

- proces – každý proces má v grafu svůj uzel,
- třída prostředku.

Uvnitř uzlu každé třídy jsou tečky (tokeny) reprezentující jednotlivé instance daného prostředku, například uzel • • • má tři instance.

Orientované *hrany* jsou také dvojího druhu:

- *hrana žádosti o prostředek* (request edge) – vede od procesu k prostředku, o který tento proces žádá, ○ ⟶ □, proces tedy čeká na přidělení prostředku,
- *hrana přidělení prostředku* (assignment edge) – vede od instance prostředku (tedy od konkrétní tečky) k procesu, kterému byla tato instance přidělena: • • •⟶ ○

Pokud proces požádá o některý prostředek, je vytvořena hrana žádosti o prostředek vedoucí od procesu ke třídě požadovaného prostředku. Až ve chvíli, kdy je prostředek přidělen, je tato hrana otočena a posunuta ke konkrétní instanci v dané třídě, změní se na hranu přidělení prostředku.

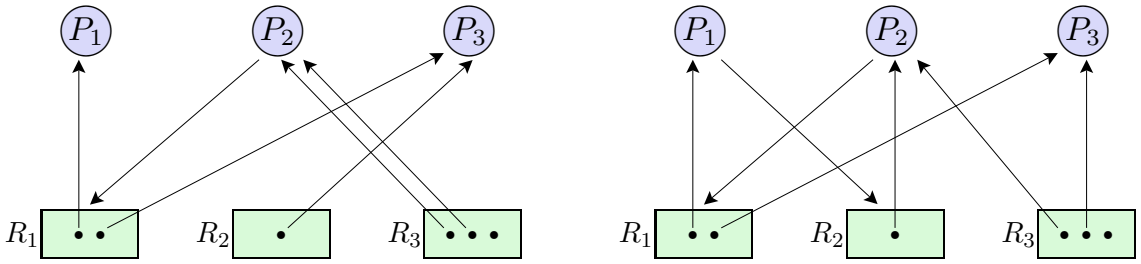
Jestliže v grafu není žádný cyklus (kružnice), pak můžeme říct, že systém není ve stavu uváznutí. Pokud je v grafu cyklus, pak systém *může* být ve stavu uváznutí (ale nemusí).

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.

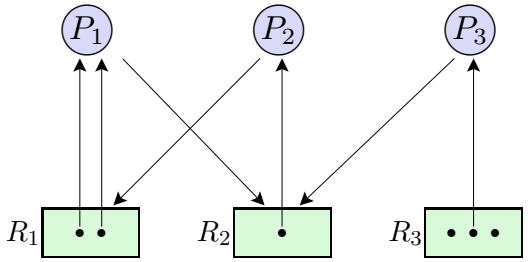
Obrázek 6.1 ukazuje dvě různé situace: v té první proces P_1 má přidělen jeden z prostředků ve třídě R_1 a o žádný další prostředek nežádá, tedy nečeká, může používat procesor. Podobně proces P_3 má své prostředky přiděleny (jednu instanci prostředku R_1 a jednu instanci R_2), taktéž může pracovat. Ovšem proces P_2 žádá o prostředek R_1, jehož obě instance jsou přiděleny jiným procesům, musí tedy čekat. Žádná kružnice (cyklus) v grafu není.

Na druhém obrázku je kružnice přes uzly $P_1 \to R_2 \to P_2 \to R_1 \to P_1$. Procesy P_1 a P_2 čekají, každý z nich na jiný prostředek. Ovšem ani v tomto případě nejde o deadlock, protože až proces P_3 nebude potřebovat prostředek R_1 (což jednou reálně nastane, protože tento proces nečeká, může



Obrázek 6.1: Grafy přidělení prostředků bez deadlocku

používat procesor), může být jeho instance prostředku R_1 přidělena procesu P_2 a kružnice přestane existovat.



Obrázek 6.2: Graf přidělení prostředků s deadlockem

Na obrázku 6.2 je již situace s deadlockem. V grafu vidíme cyklus $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$. Proces P_3 nepomůže, protože taky čeká – na prostředek R_2.

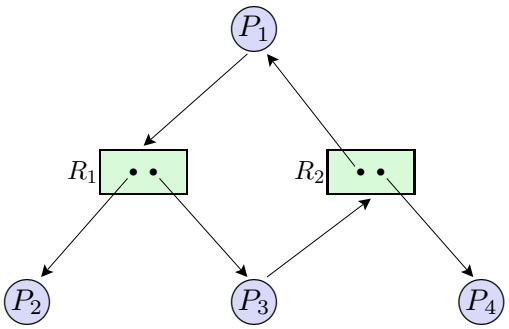
Poznámka:

Jestliže je v grafu nějaká kružnice, může, ale nemusí dojít k uváznutí. Existence kružnice je tedy nezbytnou, ale nikoliv postačující podmínkou vzniku uváznutí.

K uváznutí v případě kružnice dojde, pokud každá třída prostředků na kružnici má právě jednu instanci. Pokud tedy máme graf, kde v každé třídě prostředků je právě jedna instance, pak existence kružnice je nezbytnou a postačující podmínkou vzniku uváznutí a můžeme ji tedy využít při testování.

Úkol

Podívejte se na obrázek vpravo. Jde o uváznutí?



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 (hold-and-wait).
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* – existuje taková posloupnost 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, $0 \leq i \ \leq n)$, $P_{n+1} = P_0$.

S nebezpečím uváznutí se dá vypořádat třemi různými způsoby:

- *prevence uváznutí* – už při návrhu systému je snaha omezit riziko uváznutí, za běhu systému se už nic moc neřeší; metoda spočívá v potlačení vzniku některé z výše jmenovaných podmínek vzniku uváznutí,
- *předpovídání uváznutí* – řešíme vždy, když některý proces žádá o další prostředek, provedeme simulaci a podle jejího výsledku buď prostředek přidělíme (když je nulové riziko uváznutí) nebo necháme proces čekat,
- *detekce uváznutí* – nepokoušíme se uváznutí zabránit, ale pravidelně nebo podle potřeby testujeme, zda už k uváznutí nedošlo; když ano, pokusíme se uváznuté procesy uvolnit.

První možnost je do určité použitelná na některé typy prostředků, druhá je příliš restriktivní (vyžaduje po procesech, aby při svém spuštění deklarovaly veškeré své možné budoucí požadavky na procesy), třetí je vcelku dobře implementovatelná a používaná.

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 (abstraktní počítač), 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).

Hlavní nevýhodou této metody je, že prostředky přidělené podle prvního bodu mohou být zbytečně málo využívány, protože je má rezervovány některý proces pro použití „do budoucna“.

Nicméně přesto se s využitím této metody můžeme setkat, třebaže s úpravou: každý proces po svém spuštění dostane určitý čas procesoru (kvantum), který postupně spotřebovává. Tento čas má rezervován předem, a předpokládá se, že jeho využívání je víceméně rovnoměrně rozloženo v čase. Úprava spočívá v přizpůsobení algoritmu plánování přidělování procesoru pro různé typy procesů – interaktivní vs. výpočetní.

Nepreemptivní plánování: Řešením může být přechod na preemptivní plánování v případě, že je riziko uváznutí (tj. použijeme možnost odebrat prostředek procesu, třebaže proces by ještě chtěl prostředek používat).

Příklad

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 (exists Q: (používá(Q, R) && exists S: (čeká(Q,S))) {
    uvolni (R), přiděl (P, R)
  }
```

Slovně: pokud prostředek, o který proces žádá, je volný, přidělíme mu ho. 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 periferie) 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.

Algoritmy uvedené v této sekci vyžadují, aby proces při svém vzniku deklaroval *maximální množství* prostředků každé třídy, které bude potřebovat pro řádné splnění své úlohy (tedy jakési maximální požadavky do budoucna). Nemusí nutně tyto prostředky dostat přidělené hned na začátku, dokonce je ani nemusí při svém běhu všechny použít, je to jakási horní hranice, za kterou ovšem při svém běhu nesmí jít (později nedostane víc než kolik na začátku deklaroval).

Definice

Bezpečná sekvence pro daný stav alokace prostředků je posloupnost (všech) procesů $\{P_0, P_1, \ldots, P_n\}$ taková, že pro každé i, $1 \leq i \leq n$, platí: pokud všechny procesy $P_0, \ldots, P_{i-1}$ budou řádně ukončeny při splnění své úlohy a uvolní přidělené prostředky (přičemž počítáme s výše zmíněným maximem), pak proces P_i také může řádně ukončit svou úlohu (při využití prostředků, které získal po předchozích procesech posloupnosti).

Stav systému je *bezpečný*, jestliže existuje alespoň jedna bezpečná sekvence, tedy pokud existuje alespoň jedno pořadí procesů takové, že postupně všechny při využití deklarovaného maxima přidělených prostředků dokážou řádně ukončit svou úlohu bez uváznutí.

Stav, který není bezpečný, ještě nemusí znamenat uváznutí, ale může k němu vést.

Účelem předpovídání uváznutí je udržet systém v bezpečném stavu. Jsou dvě možná řešení:

- použití grafu nároků a přidělení prostředků,
- Bankéřův algoritmus.

První 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. V obou případech jde o to, že v reakci na žádost o prostředek v rámci simulace zjišťujeme, jestli přidělením prostředku nepřestane být stav systému bezpečným (tj. zda riziko uváznutí přestane být nulové).

Procesy musejí předem (při svém spuštění) deklarovat, které prostředky (a v jakém množství) mohou za svého běhu potřebovat – *nároky*. Část z nich si mohou vyžádat hned, s částí počítají „do

budoucna“. Je to jakési maximum, které za svého běhu nesmějí překročit (například maximální množství paměti, které za svého běhu budou potřebovat). V obou zmíněných metodách se nároky projevují – v grafu přidáváme nový typ hrany, v Bankéřově algoritmu máme pro evidenci nároků 2D pole.

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 (výhled „do budoucna“).

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

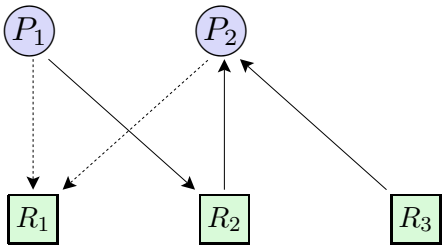
Hrany nároku pro určitý proces vznikají při spuštění tohoto procesu, pokud proces požádá o prostředek, ke kterému od něho vede hrana nároku (nemůže požádat o prostředek, ke kterému hrana nároku nevede), tato hrana se změní na hranu žádosti o prostředek, v případě přidělení prostředku se mění na hranu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

Příklad

Na obrázku 6.3 je znázorněn stav systému se dvěma procesy a třemi různými prostředky. Protože v každé třídě prostředků máme právě jednu instanci (ano, jinak bychom nemohli použít tuto metodu), nebudeme zde pro zjednodušení zobrazovat tečky.

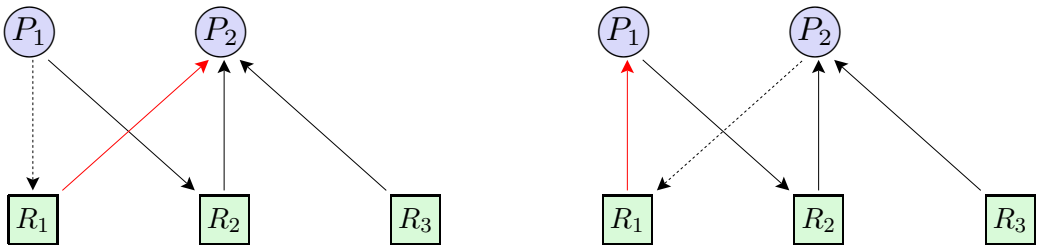
První proces nemá přiděleny žádné prostředky, ale žádá o prostředek R_2, má nárok požádat o prostředek R_1. Druhý proces má přiděleny prostředky R_2 a R_3, má nárok požádat o prostředek R_1. V grafu není žádná kružnice.



Obrázek 6.3: Graf nároků a přidělení prostředků s hranami nároku

Předpokládejme, že proces P_2 se rozhodne využít svůj nárok a požádá o prostředek R_1. Tím se změní hrana nároku na hranu přidělení prostředku (protože tím spouštíme simulaci přidělení prostředku), to vidíme na obrázku 6.4 vlevo. V grafu není cyklus, tedy deadlock nehrozí a prostředek může být přidělen.

Po ukončení simulace zůstane „testovací“ hrana přidělení prostředku tak, jak jsme ji v simulaci určili, proces P_2 může začít využívat prostředek.



Obrázek 6.4: Graf nároků a přidělení prostředků během simulace

Ovšem pokud o nárokovaný prostředek R_1 požádá proces P_1 (vidíme na obrázku 6.4 vpravo), při simulaci zjistíme, že v grafu vznikla kružnice. Znamená to, že přidělení prostředku by mohlo (nemuselo) vést k deadlocku. Pak by se „testovací“ hrana přidělení prostředku změnila na hranu žádosti o prostředek a proces by musel počkat na změnu situace.

Pokud tedy v rámci simulace zjistíme, že v grafu vznikla kružnice, znamená to nebezpečný stav (nemusí nutně dojít k deadlocku, ale jeho pravděpodobnost je nenulová). Potom tedy prostředek nepřidělíme a proces bude muset počkat – jestliže se změní situace, například jiný proces dokončí svou činnost, může být algoritmus spuštěn znovu.

6.5.2 Bankéřův algoritmus

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`$[i]$),

`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`.

`NAROKY` – 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` + `NAROKY` 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`$[i]$),

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

Poznámka:

Dále budeme pro zjednodušení zápisu používat relaci $\leq$ pro vektory definovanou takto: nechť V_1 a V_2 jsou vektory o délce m. $V_1 \leq V_2$ právě tehdy když $\forall i(V_1[i] \leq V_2[i]),\ 1 \leq i \leq m$. Slovy: první vektor je menší nebo roven druhému, jestliže všechny jeho prvky jsou menší nebo rovny prvkům se stejným indexem druhého vektoru.

Dále se v postupu objeví operace sčítání a odečítání vektorů a matic.

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$[i]$ $\leq$NAROKY$[i]$, 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$[i]$
   NAROKY$[i]$ =NAROKY$[i]$−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) NAROKY$[j]$ $\leq$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.

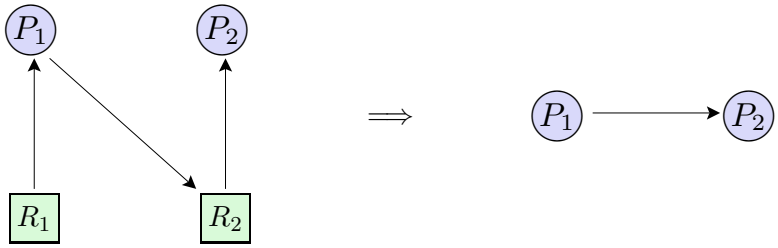
Jestliže po simulaci stav bezpečný, systém přidělí požadované prostředky procesu P_i (vlastně to už udělal v bodu 4), jinak proces musí čekat, než bude zase dostatek prostředků a je nutné vrátit změny z bodu 4 (vektor POZADAVKY$[i]$ bude nadále obsahovat nevyplněné požadavky procesu).

6.6 Detekce uváznutí

Opět rozlišíme dva případy: první metoda (s použitím grafu) je určena pro systém, kde v každé třídě prostředků je právě jeden prostředek, druhá metoda (modifikace Bankéřova algoritmu) pro systém, kde je ve třídách prostředků povoleno i více instancí.

6.6.1 Úprava grafu přidělení prostředků

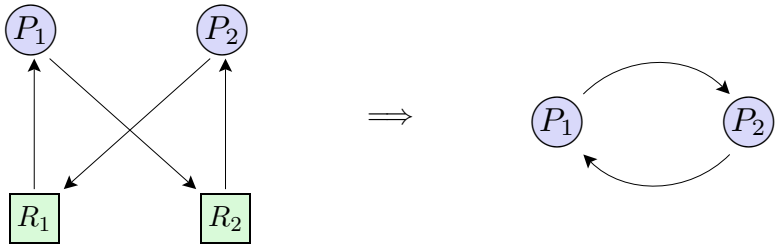
Vytvoříme *graf čekání*, ve kterém bude zachyceno vzájemné čekání mezi procesy (jeden proces čeká, až jiný uvolní nějaký prostředek). Protože nás momentálně zajímá jen to, který proces uvázl, nepotřebujeme informaci o tom, na které prostředky které procesy čekají (snadněji se detekuje kružnice).



Obrázek 6.5: Graf přidělení prostředků a ekvivalentní graf čekání

Graf čekání získáme z grafu přidělení prostředků 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). Jednodušeji: pokud mezi dvěma procesy vedla cesta (před prostředek) v grafu přidělení prostředků, pak mezi nimi bude cesta (přímá) i v grafu čekání.

Na obrázku 6.5 je ukázka vytvoření grafu čekání pro graf přidělení prostředků. V současné situaci proces P_1 má přidělen prostředek R_1 a žádá o prostředek R_2, nicméně čeká (nedostává procesor), protože tento prostředek je právě přidělen procesu P_2. V odpovídajícím grafu čekání vidíme, že proces P_1 čeká na proces P_2, resp. prostředek vlastněný daným procesem. V tomto grafu čekání není kružnice, tedy systém není ve stavu uváznutí (pozor, jde o detekci, nikoliv simulaci jako v předchozím případě).



Obrázek 6.6: Graf přidělení prostředků a ekvivalentní graf čekání

Na obrázku 6.6 je v grafu přidělení prostředků stav, kdy procesy P_2 čekají na přidělení prostředku vlastněného „tím druhým“. 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 pro detekci

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

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

Procesy nemusejí předem deklarovat, kolik kterých prostředků budou maximálně potřebovat pro dokončení své činnosti. Také půjde o simulaci, ale budeme v ní brát v úvahu pouze prostředky přidělené nebo ve stavu žádosti.

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 $false$.

6.6.3 Reakce při zjištění uváznutí

Některým z algoritmů z předchozí sekce 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 $false$). Detekční algoritmus lze spouštět například v pravidelných intervalech nebo v souvislosti s některou konkrétní událostí.

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.

Tyto problémy je tedy třeba řešit:

- Výběr obětí: které prostředky kterých procesů mají být přednostně odebírány? U některých prostředků/procesů to je menší problém, u jiných větší.

- Při uvolnění prostředků některého procesu musí daný proces zůstat v bezpečném stavu, proto je třeba zajistit, aby proces mohl po určitém čekání pokračovat v práci. I to zohledňujeme při výběru prostředků k uvolnění.
- Pokud často dochází k deadlockům, je možné, že se některý proces stane obětí opakovaně. Musíme tedy zajistit, aby se některý proces nestával obětí příliš často.

 Pokud používáme *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. Dá se zohlednit například priorita a typ procesu (například zda je interaktivní nebo jestli jde o službu), množství a typ prostředků přidělených procesu, jak dlouho proces běží, jak dlouho nepracoval, atd.

Poznámka:

Běžné operační systémy jako Windows a UNIXové systémy spíše deadloky ignorují. Používají některé z metod prevence uváznutí (například pro přístup k tiskárnám mají speciální tiskové procesy) a počítají s tím, že pravděpodobnost uváznutí je velmi malá. Ve Windows XP a vyšších existuje určitá omezená možnost detekce deadloku pro ovladače (tedy zjišťování, zda ovladače na sebe navzájem nečekají).

Kapitola 7

Správa periferií

Rychlý náhled: 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. Zbývající část kapitoly je věnována koexistenci více operačních systémů na jednom zařízení.

Klíčová slova: I/O systém, periferie, I/O buffer, ovladač, model WDM, WDDM, frameworky ovladačů, IRQ, výjimka, obslužná rutina, řadič přerušení, IDT, tasklet, DPC, virtualizace, virtuální počítač, paravirtualizace, hypervizor typu 1 a 2, emulátor, podsystém, serverová a desktopová virtualizace.

Cíle studia: Po prostudování této kapitoly se budete orientovat v řízení vstupně-výstupních operací v operačním systému. Porozumíte systému správy ovladačů, ale také principu virtualizace a funkcionalitě různých typů virtualizačních řešení včetně použití hypervizora.

7.1 I/O systém

7.1.1 Druhy periferií

Vstupně-výstupní zařízení lze dělit podle různých kritérií, na některá se zde podíváme.

Podle struktury přenášených dat: *Znaková zařízení* (character-stream) jsou taková zařízení, která komunikují po oktetech (byte) nebo slovech (několik oktetů), přičemž tyto úseky dat jsou ve streamu bez další vnitřní struktury. Typickým příkladem takového zařízení je klávesnice nebo myš. Do této kategorie můžeme zařadit i různá virtuální zařízení, včetně `/dev/random` v UNIXových systémech.

Bloková zařízení (block) komunikují po blocích dat, přičemž sada souvisejících bloků je opatřena metadaty. Například disky (HDD, SSD, RAM disky) jsou typickými blokovými zařízeními, protože při ukládání souboru (jehož délka může být různá, různý počet bloků) se pracuje také s názvem souboru, vlastníkem, přístupovými oprávněními, atd.

Třetí skupinu tvoří *speciální zařízení*, u kterých nemá moc smysl zvažovat strukturu generovaných dat, například timer, který v pravidelných intervalech generuje elektrický impuls. Tato zařízení nicméně můžeme brát jako speciální případ znakových zařízení.

Podle směru komunikace: Vstupní zařízení slouží ke vstupu dat od uživatele, výstupní předává data uživateli. Existují také zařízení vstupně-výstupní, která přenášejí data v obou směrech.

Klávesnice a myš jsou typickými vstupními zařízeními. Běžný monitor je výstupní zařízení, ale dotykový monitor je vstupně-výstupní zařízení. Paměťová média jsou většinou vstupně-výstupní.

Podle možností sdílení zařízení: V kapitole o synchronizaci jsme se zabývali řízením přístupu k takovým prostředkům (souborům, paměťovým sekcím, zařízením,...), ke kterým nemůže v jednom okamžiku přistupovat více než jeden proces (nebo jen určitý maximální počet – předplacení). Takže zařízení lze rozdělit i takto:

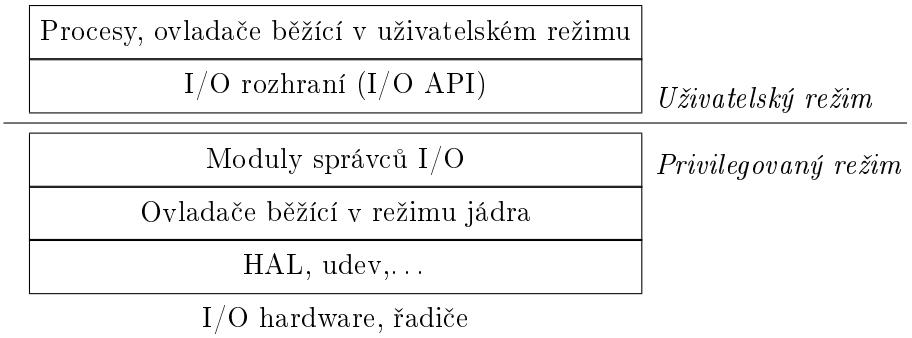
- *zařízení s vyhrazeným přístupem* – přístup k těmto zařízením je třeba synchronizovat (například tiskárna),
- *sdílená zařízení* mohou být rozdělena na části, tyto části bývají zpravidla vyhrazovány (například operační paměť rozdělená na paměťové stránky, proces má přiděleny vždy určité paměťové stránky, u sdílených se řeší synchronizace),
- *zařízení s volným přístupem (společná zařízení)* nevyžadují žádnou synchronizaci; obvykle jde o zařízení, kde má smysl pouze přístup pro čtení, například teplotní senzor.

Zastavme se u prvního typu, tedy u zařízení s vyhrazeným přístupem. Přístup k těmto zařízením se dá řešit následovně:

- 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) – proces ve skutečnosti komunikuje s jakousi „virtuální náhradou“, speciálním procesem, a teprve tento proces komunikuje se samotným zařízením; toto řešení také známe pod názvem „abstraktní počítač“.

7.1.2 Struktura I/O systému

I/O systém (angl. I/O kernel subsystem) je systém určený k řízení I/O zařízení. Jeho součástí je také *I/O rozhraní*, což je sada rutin (funkcí) a objektů poskytovaná operačním systémem procesům pro přístup k periferiím, jiným způsobem obvykle běžné procesy s periferiemi komunikovat nemohou.



Obrázek 7.1: Struktura I/O systému

Řadič (controller) je integrovaný obvod (na desce tištěných spojů, případně samostatný čip), přesněji kód v tomto integrovaném obvodu, určený k řízení daného zařízení a komunikaci s ním. Řadič

může být jednoduchý, ale řadiče některých zařízení jsou poměrně složité obvody srovnatelné s klasickým procesorem (to se týká například SSD, jejich řadiče jsou ARM čipy). Software řadiče je součástí firmwaru.

Další úrovní jsou *ovladače*. Ovladač je software běžící zpravidla jako modul v jádře (ne nutně, existují i ovladače v uživatelském prostoru), který slouží jako rozhraní ke konkrétnímu zařízení, nebo je součástí složitější komunikační struktury vedoucí ke konkrétnímu zařízení.

✎ Pokud proces provede systémové volání vztahující se k určitému zařízení (například potřebuje otevřít soubor na disku, přidělit další paměť apod.), vytvoří *I/O request* (žádost o přístup k I/O zařízení). Tato žádost je zařazena do fronty, kterou zpracovává příslušný *I/O plánovač* (I/O scheduler).

7.1.3 I/O Buffer

✎ *Buffer* (vyrovnávací paměť) je paměť pro dočasné uložení dat přenášených mezi dvěma zařízeními. Není to úplně totéž jako cache – zatímco data v cache jsou určitým způsobem organizována a obvykle existují mapovací algoritmy, které určují, jak se s kterou oblastí cache zachází (viz cache v procesorech), s bufferem se obvykle zachází „průtokovým“ způsobem (organizace typu FIFO).

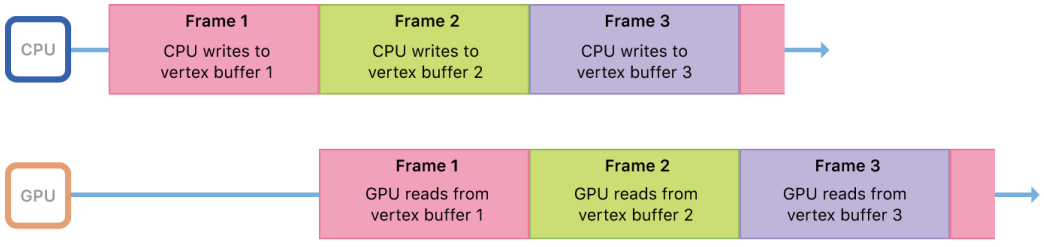
Nicméně jejich účel je podobný – vyrovnání komunikace mezi dvěma komponentami (zařízeními) s různou rychlostí. Například soubor určený k vytisknutí na tiskárně bude pravděpodobně čekat v tiskové frontě a pak v bufferu tiskárny, než ho tiskárna zvládne přenést na papír. Buffery máme také v síťových zařízeních, včetně síťového rozhraní koncového zařízení.

Dalším účelem bufferu může být vyrovnání rozdílu ve velikosti přenášených či zpracovávaných bloků, případně rozdílu v šířce sběrnic, popřípadě v cílovém zařízení či komponentě potřeba zkompletovat sekvenci bloků do jednoho celku.

✎ Někdy je nutné použít *několik bufferů*. Například pokud jeden buffer nestačí k vyrovnání rychlosti zařízení (odesílatel je příliš rychlý), použijeme druhý buffer a v komunikaci se střídavě do jednoho bufferu ukládají data od odesílatele a z druhého bufferu se čte (pak se prohodí). Je to rychlejší než jeden buffer vyhrazovat.

Grafické karty používají ve videopaměti alespoň dva buffery – snímek (rámec) je postupně zpracováván v zadním bufferu a poté přesunut do předního bufferu (k odeslání přes grafické rozhraní do displeje).

Pro synchronizaci komunikace mezi procesorem a grafikou ve videopaměti a I/O rozhraním jsou někdy potřeba tři buffery. Všechny tři strany (procesor, grafika a I/O rozhraní) pracují paralelně: procesor předzpracovává grafické instrukce, předává je grafice, ta je provádí (vyhodnocuje, vytváří obraz) a předává I/O rozhraní. Postup je naznačen na obrázku 7.2.



Obrázek 7.2: Využití tří bufferů při V-sync[1]

[1] Zdroj: https://developer.apple.com/documentation/metal/synchronization/cpu_and_gpu_synchronization

Další informace:

- https://www.displayninja.com/what-is-freesync/
- https://lifehacker.com/learn-the-basics-of-nvidias-g-sync-and-amds-freesync-mo-1831578473

7.2 Ovladače

7.2.1 Struktura ovladačů

Ovladač zařízení je software, který slouží jako rozhraní mezi zařízením a procesy, nebo jinými ovladači a moduly jádra.

Jednou z úloh ovladače je zprostředkovávat komunikaci mezi propojenými entitami tak, aby bylo možné 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.

Současné systémy se neomezují na ovladače (fyzických) zařízení, ale existují také ovladače virtuálních zařízení či souborových systémů, a také *filtry*, což jsou speciální „průtokové“ ovladače provádějící zpracování dat procházejících mezi komponentami uvnitř jádra (šifrování, komprimace, filtrování, monitoring, atd.).

Je obvyklé, že ovladač implementuje několik důležitých funkcí. Funkce pro komunikaci s procesy byly naznačeny výše, ale k důležitým funkcím patří také ty vztahující se k systému:

- *rutina obsluhy přerušení* – kód, který se má provést, pokud zařízení ovladače vygeneruje přerušení,
- *inicializační rutina* – kód, který se má provést při inicializaci ovladače (například rutina pro přidání zařízení, která je používána správcem Plug-and-Play).

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).

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.

V současných operačních systémech jsou ovladače často programovány jako moduly jádra. To znamená, že jádro jako takové je ve svém kódu neobsahuje, ale načítají se ze souborů, které jsou obdobou dynamicky linkovaných knihoven (linkují se do jádra).

Existují také projekty, jejichž účelem je přenést co nejvíc z funkcionality ovladačů z jádra do uživatelského prostoru. To je velmi užitečné, protože většina chyb při běhu jádra nebo dokonce jeho pádů je zaviněna právě špatně napsanými ovladači (vše, co běží v režimu jádra, se dostane opravdu kamkoliv, tedy poškození datových struktur jádra je docela dobře možné). Původně se tyto snahy objevily spíše v UNIXových systémech, ale v současné době se ovladače v uživatelském prostoru používají i ve Windows.

7.2.2 Ovladače ve Windows

Ve Windows rozlišujeme různé druhy ovladačů podle různých kritérií. Celková struktura je poměrně složitá, zde si ji trochu zjednodušíme.

Dělení podle modelu (tj. způsobu, jak je ovladač naprogramován, co vše může implementovat a jak komunikuje se svým okolím):

- ovladače podle modelu WDM (Windows Driver Model) – většina ovladačů běžných zařízení (klávesnice, zvuková karta apod.), používá se od Windows 2000,
- ovladače podle modelu WDDM (Windows Display Driver Model) – speciální model pro multimediální ovladače (grafické karty apod.), existuje od verze Vista,
- starší ovladače (předchůdci WDM) – například PMD (Protected Mode Driver), RMD (Real Mode Driver), pro starší nebo jednodušší zařízení.

U modelů WDM a WDDM existují různé verze, jejich specifikace se může mírně lišit.

Dělení podle umístění kódu (resp. formy komunikace se systémem):

- ovladače běžící v režimu jádra (Kernel-Mode Drivers) – tyto ovladače jsou vlastně moduly jádra, většinou se načítají ze souborů s příponou .sys,
- ovladače běžící v uživatelském prostoru (User-Mode Drivers) – obdoba služeb, obvykle běží v některém hostitelském procesu, často v `svchost`.

V režimu jádra běží také například ovladač souborového systému NTFS načítaný ze souboru `ntfs.sys`.

Pro každý z těchto dvou typů ovladačů existuje pomocný podsystém, který zajišťuje jejich běh:

- Kernel-Mode Driver Framework (KMDF) – podsystém pro ovladače běžící v režimu jádra,
- User-Mode Driver Framework (UMDF) – podsystém pro ovladače běžící v uživatelském režimu, jeho součástí je i modul *Driver Manager* zajišťující mimo jiné i komunikaci ovladačů s okolím (obdoba SCM od služeb).

Ovladače běžící v KMDF mají tyto výhody:

- snadněji mohou komunikovat se zařízením, protože jsou v jádře a nemusejí řešit přepínání mezi režimem jádra a uživatelským režimem,
- mohou používat mechanismus DMA a jiné způsoby přímé komunikace mezi komponentami,
- pokud potřebujeme Plug-and-Play, nemusíme to řešit přímo my (protože to dělá jádro za nás),
- mohou se napojit na jiné moduly jádra přímo přes jejich rozhraní, což se hodí hlavně u filtrů,...

Nevýhodou ovladačů běžících v režimu jádra je vyšší riziko pro jádro systému – cokoliv se pokazí v jádře, to ovlivní celý systém (modrá obrazovka apod.).

Ovladače běžící v UMDF oproti tomu mohou snadněji komunikovat s běžnými procesy (třebaže to není až tak jednoduché – obvykle běží pod účtem LocalService, tedy nevlastní objekt WinSta0, nemají okno pro příjem zpráv a nemají přístup ke schránce). UMDF ovladač také snadněji zjišťuje situace, kdy je zařízení ve stavu nízké spotřeby (protože baterie je blízko stavu vybití), tomu dokáže přizpůsobit zpracovávání fronty I/O požadavků. V neposlední řadě je provoz UMDF ovladačů bezpečnější, protože nemůže způsobit BSOD („modrou smrt“).

Například ovladač PCap (nebo Npcap, podle systému) je napojen na moduly síťového zásobníku v jádře a monitoruje veškerou síťovou komunikaci. Neblokuje ji, pakety přes něj pouze procházejí. Musí být v jádře, jinak by se k síťovému provozu nedostal.

Naproti tomu ovladače, které především komunikují s procesy, ale pro komunikaci s moduly jádra jim stačí prostředník, jsou obvykle typu UMDF. Týká se to například virtuálních zařízení nebo některých zařízení připojených přes USB.

Výše jsou uvedeny typy ovladačů WDM, WDDM a další. V současnosti se můžeme setkat ještě s jednou zkratkou: *WDF (Windows Driver Framework)*. Framework WDF v sobě ve skutečnosti zahrnuje funkcionalitu obou dříve jmenovaných frameworků KMDF a UMDF, tedy se dá použít pro programování obou typů ovladačů (postupy jsou podobné), a programátor si může vybrat, který typ vlastně bude chtít vytvořit. Framework obsahuje abstraktní vrstvu, která hotový ovladač zařadí do KMDF nebo UMDF.

Účelem WDF je usnadnit programování ovladačů, protože velká část kódu je předpřipravená v příslušných knihovnách. Je to podobné jako s frameworky pro programování webových aplikací.

Poznámka:

Abychom si to ujasnili: můžeme buď programovat přímo WDM ovladač a budeme muset spoustu kódu řešit „ručně“, přičemž jinak se programují ovladače pro režim jádra a jinak pro uživatelský režim. Nebo budeme programovat ovladač s využitím frameworku WDF, kde je část funkcionality už předpřipravená a jen rozhodneme, zda půjde o ovladač pro KMDF nebo UMDF.

Podle typu instalace rozlišujeme dvě skupiny ovladačů podle toho, zda podporují zjednodušenou instalaci zařízení (bez nutnosti hardwarové konfigurace, určování IRQ apod.):

- ovladače *Plug-and-Play* – souvisejí s konkrétním zařízením, u kterého má smysl uvažovat o této funkci (výměnné paměti, některé typy rozšiřujících karet, klávesnice, myši, tiskárny, apod.), předpokládá se také komunikace se správcem napájení,
- ovladače *non-Plug-and-Play* (rozšíření jádra) – obvykle nesouvisejí s konkrétním hardwarem, nebo sice ano, ale se zařízením komunikují ještě přes další ovladač (typicky ovladače komunikačních protokolů).

Podle konkrétní funkce, kterou v systému plní, dělíme ovladače typu WDM – toto dělení souvisí i se začleněním do komunikační struktury v jádře:

- *ovladače funkce* – tyto ovladače komunikují přímo s konkrétním zařízením, jejich úkolem je zajišťovat rozhraní k zařízení,
- *ovladače sběrnice* – spravují fyzické a logické sběrnice (například ovladač PCI nebo USB), tyto ovladače detekují zařízení připojované k dané sběrnici podle standardu Plug-and-Play, zajišťují správné napájení sběrnice apod.,
- *ovladače filtru* – ovlivňují komunikaci od nebo do ovladače funkce, tedy buď rozšiřují funkčnost navázaného ovladače funkce nebo ji nějakým způsobem mění; může jít například o šifrování, nejrůznější konverze, sledování, atd.

Hierarchie ovladačů. Jak bylo výše naznačeno, struktura ovladačů ve Windows je poměrně složitá, *v této struktuře* se rozlišují tyto *typy ovladačů* uspořádaných do vrstev nebo ještě složitěji:

1. *ovladače třídy* – vycházejí ze třídění zařízení do logických tříd, kde v rámci jedné třídy existují standardizované postupy, funkce a datové struktury (například existuje třída pro pevné disky),

tyto ovladače umožňují standardizovaným způsobem přistupovat k zařízením od různých výrobců tak, aby zařízení fungovalo, i když nebudou jeho funkce plně využity,

2. *ovladače portu* – jedná se o ovladače realizující rozhraní k některému I/O portu, například USB nebo SCSI, obvykle nejde o klasické ovladače, ale spíše o dynamicky linkované knihovny,
3. *ovladače miniportu* – propojují komunikační cestu mezi portem některého rozhraní a konkrétním adaptérem (rozšiřující kartou) na tomto rozhraní, jedná se o skutečné ovladače zařízení, které se napojují na funkce ovladače portu.

Navrstvené ovladače ve skutečnosti navzájem nekomunikují přímo, komunikace mezi dvěma ovladači vždy vede přes *správce I/O* – příslušný podsystém (v novějších Windows to je KMDF nebo UMDF). Vlastně je to podobné jako u služeb: tam vede komunikace zásadně přes modul SCM v jádře.

Další informace:

- https://learn.microsoft.com/cs-cz/windows-hardware/drivers/gettingstarted/choosing-a-driver-model
- https://msdn.microsoft.com/en-us/windows/hardware/drivers/wdf/overview-of-the-umdf
- https://msdn.microsoft.com/en-us/windows/hardware/drivers/wdf/user-mode-driver-framework-frequently-asked-questions
- http://technet.microsoft.com/en-us/library/cc778056%28WS.10%29.aspx

K ovladačům se můžeme dostat na několika místech:

- stejně jako u služeb, informace o ovladačích najdeme v registru, dále přes službu WMI, příkaz `sc` apod. (probírali jsme na cvičeních z Operačních systémů),
- *Process Explorer* (od Sysinternals) – Pokud ve spodním podokně zobrazíme seznam DLL a pak v horním podokně klepneme na proces System, získáme přehled o většině ovladačů v jádře.
- *WinObj* (od Sysinternals) – zde máme přehled o objektech ovladačů (především v kontejnerech *Driver* a *FileSystem*).

Jak můžeme vidět na obrázku 7.3, mnohé KMDF ovladače ve skutečnosti běží jako vlákna hlavního systémového procesu. Zde je například ovladač `ACPI.sys` běžící jako vlákno v systémovém procesu s prioritou 8 (normální).

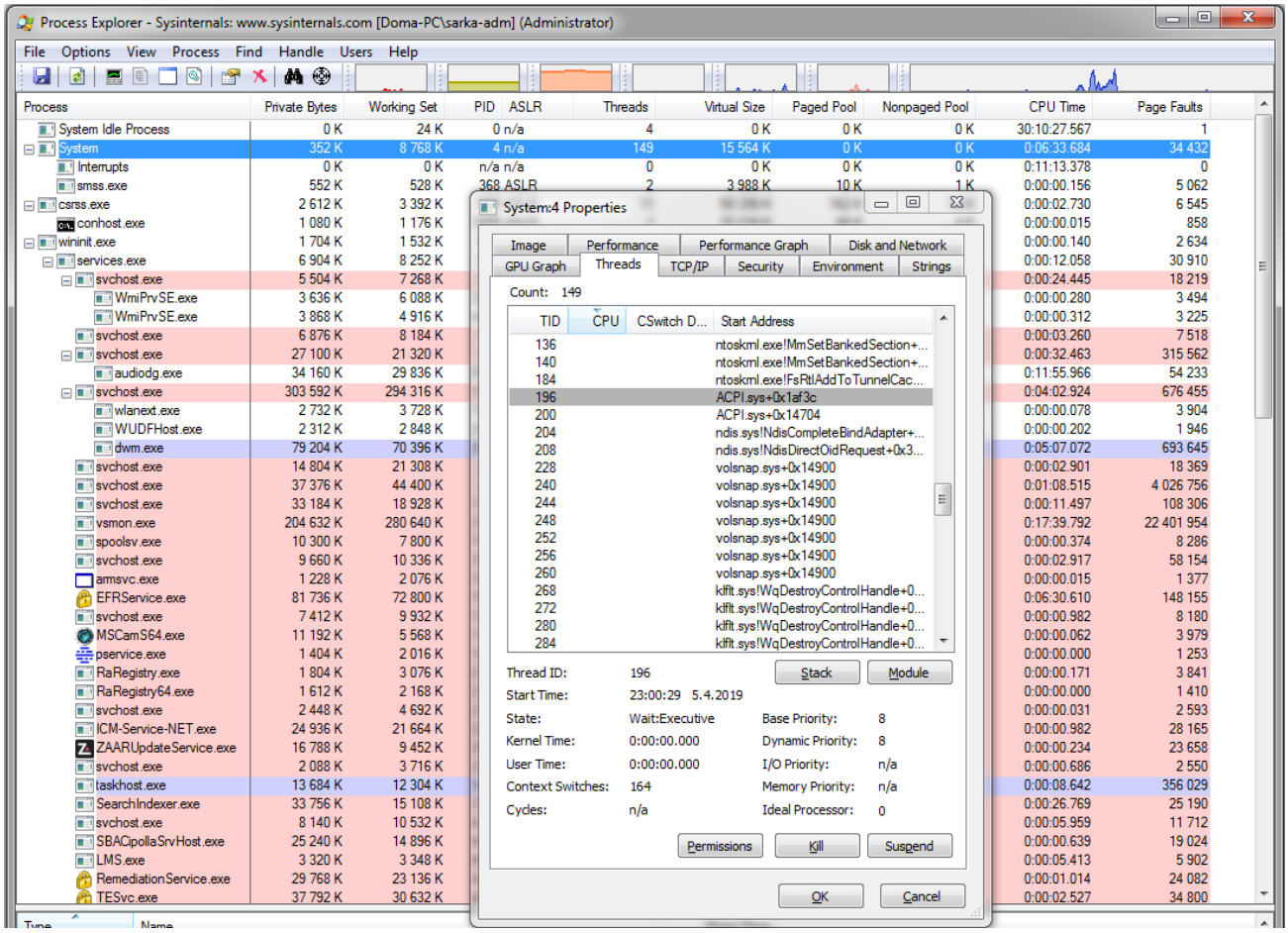
7.2.3 Ovladače v Linuxu

V Linuxu lze za ovladače považovat následující:

- ovladače zařízení, včetně virtuálních zařízení, různé filtry,
- ovladače souborových systémů,
- síťové protokoly, což jsou obvykle moduly jádra.

V Linuxu existují dva základní typy ovladačů, podle *umístění kódu*:

- ovladače běžící v režimu jádra (Kernel Drivers) – fungují jako moduly jádra,
- ovladače běžící v uživatelském prostoru (User-Space Drivers) – v jádře mají jen svého „agenta“, který jim zprostředkovává přístup k prostředkům jádra, ale samy běží v uživatelském prostoru, většinou jako služby/démoni.



Obrázek 7.3: Vlákna procesu System v Process Exploreru

První typ ovladačů má samozřejmě výhodu přímého přístupu ke strukturám jádra a různým možnostem komunikace s jinými moduly jádra, ale na druhou stranu je třeba jejich kód velmi důkladně odladit, protože jakákoliv chyba by mohla fatálně ovlivnit fungování jádra. Je třeba velmi dbát na používání mutexů, spinlocků a jiných synchronizačních mechanismů (na správném místě, ve správný čas), kromě zamykání taky odemykat, podrobovat důkladné analýze cokoliv, co přijde „zvenčí“ (třeba ze vstupního zařízení), protože by případně mohlo jít o hackerský útok.

Moduly pro načtení do jádra jsou uloženy v souborech s příponou .ko (kernel object), a to `/lib/modules/`*`číslo_jádra`*`/kernel/drivers/`*`kategorie_modulu`*`/`*`název_modulu`*`.ko`.

Oproti tomu ovladače běžící v uživatelském prostoru mají výhodu v menších problémech s bezpečností (což ale neznamená, že by se jejich programování mohlo odfláknout), ale na druhou stranu se „nějak“ musí do jádra dostávat. K tomu většinou slouží *modul jádra FUSE* (FileSystem in UserSpace), který právě plní roli „styčného důstojníka“ pro komunikaci s jádrem.

Poznámka:

Přes FUSE je dnes řešeno obrovské množství ovladačů, právě z důvodu bezpečnosti (a také se to snadněji programuje, jsou k dispozici knihovny s předpřipraveným kódem). Jedná se většinou o reálné nebo virtuální souborové systémy či cokoliv, co prostě funguje jako filtr dat (to vše je ve skutečnosti v UNIXových systémech bráno jako souborový systém), také pro šifrování, kompresi, logování, atd.

Z nejznámějších například ntfs-3g (ovladač souborového systému NTFS), EncFS (souborový systém nabízející šifrování), FuseCompress (komprese, kromě jiného také algoritmem gzip), ClamFS (antivirová kontrola při přístupu k souborům), sshfs (implementace SSH), atd.

https://github.com/libfuse/libfuse/wiki/Filesystems

Další nevýhody ovladačů v uživatelském prostoru jsou podobné jako u běžných procesů, například v případě nutnosti mohou být jejich paměťové stránky odloženy (swapovány). Jejich komunikace s čímkoliv v jádře je pomalejší (je třeba provádět přepínání mezi režimy) – to je pozorovatelné například u gigabitových ethernetových karet, pokud jsou jejich ovladače takto řešeny.

Moduly mohou mít také parametry, což se využívá hlavně u tzv. *watchdogů*, tedy modulů hlídajících některé funkce zařízení a systému.

Příklad

Příkaz `lsmod` vrací seznam modulů, které jsou aktuálně zavedeny v jádře. Například:

```
lsmod | grep wifi
```

```
iwlwifi      188416 1 iwldvm
cfg80211     524288 3 iwlwifi,mac80211,iwldvm
```

Získali jsme název modulu, jeho velikost a seznam modulů, které tento modul používají (i mezi moduly je totiž určitá komunikační struktura). Pokud budeme chtít podrobnější informace o modulu `iwlwifi`, použijeme následující příkaz:

```
modinfo iwlwifi
```

Zjistíme umístění souboru, ze kterého se modul načítá, licenci, popis, závislosti na jiných modulech, také digitální podpis (moduly jádra musejí být digitálně podepsané) a další informace.

Při načítání nebo provozu modulu mohou být nastaveny některé z tzv. *tained příznaků jádra*. Tyto příznaky jádra indikují, že něco v jádře není úplně v pořádku a existuje možnost narušení stability nebo výkonu jádra. Některé z tained příznaků jsou celkem nevinné a není důvod si jich více všímat, například příznak „P“ (načten modul s proprietální nebo neuvedenou, a tedy zřejmě také proprietální, licencí). Jiné příznaky však rozhodně zasluhují pozornost, například příznak „B“ (paměťová stránka některého procesu je poškozena) nebo „H“ (došlo k vážné hardwarové chybě, například přehřátí procesoru).

Úlohu hlavního správce I/O plní u starších systémů především *HAL* a *udev*. Práci se speciálními zařízeními má v kompetenci udev, zbytek je na vrstvě HAL. Tato vrstva například provádí samotné načítání modulů s ovladači, připojuje souborové systémy, plní roli správce Plug-and-Play (hlídá a zajišťuje připojování zařízení). V neposlední řadě vytváří jednotný virtuální pohled na strukturu zařízení (podobně jako existuje struktura souborů) a exportuje ho procesům. Jedná se o stromovou strukturu dostupnou přes souborový systém *sysfs* v adresáři `/sys`.

V novějších systémech již neexistuje vrstva HAL, vše včetně práce s adresářem `/sys` je v režii modulu udev.

Každé zařízení (včetně virtuálních) má svůj vlastní *objekt*. Tento objekt obsahuje veškeré informace o zařízení včetně kategorie, údajů o řízení přístupu, rodiče, názvu apod.

7.3 Přerušení

7.3.1 Mechanismus přerušení a výjimek

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.

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 (formou volání jádra), pak jde o *softwarové přerušení*. Softwarová přerušení také nazýváme *výjimky*.

Hardwarová přerušení jsou například generována I/O zařízeními jako je klávesnice (stisk klávesy) nebo myš (pohyb či stisknutí tlačítka), ale třeba také procesorem, přerušení generovaná časovačem (v pravidelných intervalech, předem nastavených), při hardwarové implementaci ochrany paměti je procesorem generováno přerušení při neoprávněném přístupu do chráněné paměti. Základní charakteristikou hardwarových přerušení je, že přichází „nečekaně“, bez přímé návaznosti na prováděný programový kód.

Softwarová přerušení (výjimky) na rozdíl od hardwarových přerušení vyplývají z prováděného kódu a při opakování téhož kódu za stejné situace (při běhu stejných procesů apod.) se stejnými daty by byla generována znovu. 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.

Rozlišujeme výjimky nechybové (trap), opravitelné (fault) a neopravitelné (abort).

Kanály přerušení jsou hardwarový systém pro signalizaci přerušení. Jednotlivé kanály jsou reprezentovány čísly IRQ (Interrupt Request). Na jednom IRQ kanálu může být napojeno více zařízení ⇒ sdílený kanál, sdílené IRQ.

Jejich provoz zajišťuje speciální obvod *řadič přerušení* (IC – Interrupt Controller, resp. PIC (Programmable IC), který je buď součástí procesoru nebo může jít o samostatný obvod, který například souvisí s konkrétní sběrnicí.

Řadič přerušení zajišťuje několik možných realizací IRQ (hlášení úrovní signálu, hlášení hranou, hybridní, hlášení zprávou), na použitém typu realizace závisí např. i to, zda může být kanál sdílen. Typicky se tyto odlišnosti dají pozorovat mezi sběrnicemi PCI a ISA, kdy na PCI je sdílení IRQ celkem bezproblémové, na sběrnici ISA je na některých architekturách dokonce nemožné.

V jádře operačního systému pak najdeme alespoň jeden *modul pro obsluhu přerušení* (Interrupt Handler), který zajišťuje vyřízení (obsluhu) přerušení z pohledu operačního systému (ve víceprocesorovém systému bývá těchto modulů více).

7.3.2 Obsluha přerušení

Aby zařízení mohlo procesoru posílat signály přerušení, musí zaregistrovat *obslužnou rutinu přerušení* (handler), a totéž platí i pro výjimky. Obslužná rutina obsahuje kód, který má být proveden, když zařízení vygeneruje toto přerušení.

Obslužná rutina nesmí dlouho blokovat procesor, a protože také často má právo přistupovat k synchronizovaným objektům jádra, jsou na ni kladeny přísné požadavky:

- musí být co nejkratší,
- musí používat pouze statické datové struktury, nesmí volat jiné funkce,
- používá pouze atomické operace.

Pokud je třeba provést kód, který je delší nebo jinak neodpovídá požadavkům, rozdělí se na části:

- horní polovina (musí se provést okamžitě, odpovídá požadavkům na obslužnou rutinu),
- dolní polovina (časově náročné, ale ne kritické operace apod.).

Dolní polovina (ta méně kritická) se může implementovat několika různými způsoby (záleží na konkrétním operačním systému), obvykle má na procesoru menší přednost než samotná obsluha přerušení, ale větší než běžné procesy.

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). To znamená, že přerušení můžeme rozdělit do dvou skupin:

- *maskovatelná přerušení* (maskable interrupt) – zde patří většina přerušení od různých zařízení,
- *nemaskovatelná přerušení* (nonmaskable interrupt, NMI) – nikdy nejsou maskována, například přerušení signalizující problémy hardwaru, v UNIXových systémech také přerušení způsobující restart po zamrznutí systému.

Kód, který je obslužnou rutinou přerušení nebo vyžaduje maskování přerušení, by měl být co nejkratší, protože při jeho vyhodnocování systém nereaguje na žádná IRQ.

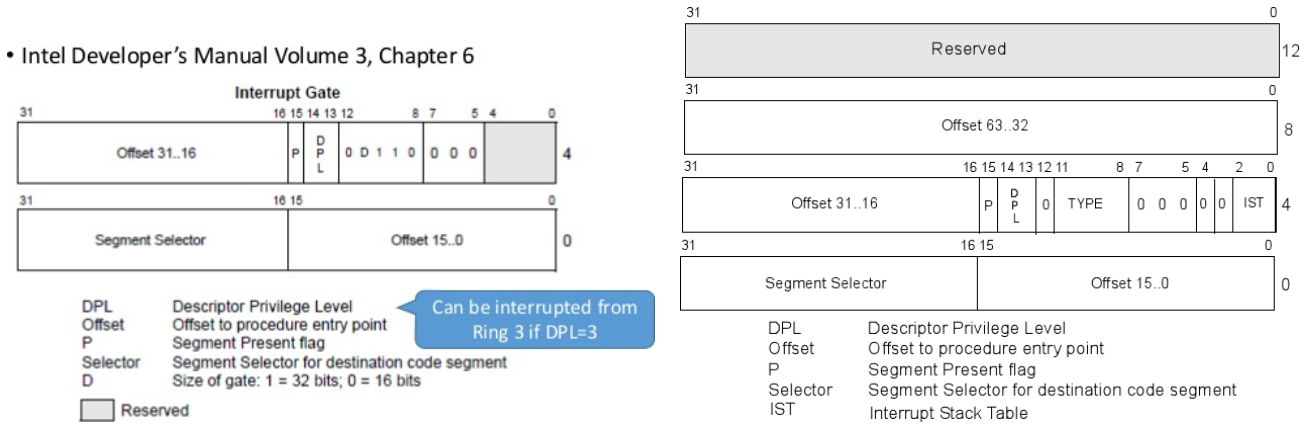
V UNIXových systémech platí, že sdílená přerušení nelze maskovat. Maskování se obvykle používá na jedno konkrétní přerušení, i když lze lokálně (na jednom procesoru či jádře) zakázat všechna přerušení, u která je to možné, najednou. Doporučuje se zakazovat jedno přerušení jen v naprosto nevyhnutelných případech, a o to více se doporučuje pokud možno se vyhýbat plošnému maskování přerušení.

Tabulka deskriptorů přerušení. Informace k jednotlivým IRQ jsou uloženy v poli, které se nazývá *Interrupt Descriptor Table* (IDT). Tato tabulka se používá na všech hardwarových architekturách, ale může mít různou podobu. Ve skutečnosti ji vede operační systém, ale adresa jejího začátku je v jednom z registrů. Položky („řádky“) této tabulky jsou *vektory přerušení*, tedy jakési záznamy o probíhajícím přerušení s daným číslem (pro každé číslo IRQ je zde jeden vektor přerušení).

Formát vektorů přerušení pro architektury x86 a x64 je na obrázku 7.4. Nejdůležitější části jsou:

- DPL (Descriptor Privilege Level) – číslo od 0 do 3 určující úroveň oprávnění subjektu, který způsobil přerušení (0 pro zařízení nebo jádro – Ring0, 3 pro běžné procesy – Ring3), v současných operačních systémech se typicky povoluje jen 0,
- P (Present) – 1bitová informace, zda se tento konkrétní vektor používá nebo jestli je prázdný,
- Segment – určení segmentu (bloku v paměti), ve kterém se nachází obslužná rutina přerušení,
- Offset – offset (relativní adresa v segmentu) obslužné rutiny.

[2] Zdroje: https://www.slideshare.net/inaz2/abusing-interrupts-for-reliable-windows-kernel-exploitation-en, http://ethv.net/workshops/osdev/notes/notes-3



Obrázek 7.4: Formát položky v IDT pro architektury x86 a x64[2]

Identifikace obslužné rutiny (segment a offset) vede buď přímo k obslužné rutině, nebo k poli adres (či jiných identifikací) obslužných rutin pro všechna zařízení sdílející toto IRQ.

Co se děje, když je vyvoláno přerušení (například po stisku klávesy):

- IRQ je po vygenerování komponentou nejdřív posláno do *řadiče přerušení* (PIC), který tato přerušení eviduje a informuje procesor, že došlo k přerušení (u Intelu to je signál INTR, sděluje „pozor, přišlo přerušení“),
- procesor zkontroluje, zda nemají být přerušení maskována, pokud ano, ignoruje je, pokud ne, postupujeme dále,
- procesor odpoví řadiči přerušení (u Intelu signálem INTA), tedy se ptá „o co jde?“,
- PIC najde a pošle procesoru záznam o přerušení z IDT,
- zbytek postupu je v režii operačního systému: uloží se kontext předchozího běžícího procesu, určí se obslužná rutina, spustí se…

Řadič přerušení má omezenou kapacitu paměti, do které ukládá záznamy o přerušení. Proto pokud procesor dostatečně rychle „neodebírá“ tyto záznamy, novější přerušení přepisují ta dříve uložená. Důsledkem je ztrácení přerušení (uživatel má pak dojem, že je „ignorován“). Ovšem ne vždy se přerušení opravdu ztrácejí, jejich odebírání může být pouze zpomaleno.

Problematika IRQ je poněkud složitější. Existují různé možnosti hlášení IRQ, řešení souběhu žádostí v čase, priorit, paralelní obsluhy přerušení apod.

7.3.3 Správa přerušení v různých systémech

MS-DOS. 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 (v *tabulce vektorů přerušení*, jejíž správu má na starosti BIOS), 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).

Linux. Základem evidence přerušení je *tabulka deskriptorů přerušení* (IDT), o které již byla řeč dříve. Ke každému přerušení zde máme všechny potřebné informace, včetně pole adres obslužných rutin napojených na dané IRQ. Pole, protože totéž IRQ může být sdíleno více zařízeními, a ke každému potřebujeme evidovat jeho obslužnou rutinu.

Předpokládejme, že proběhly kroky diskutované v postupu na straně 137. Řízení přebírá operační systém (zde Linux), resp. jeho modul pro obsluhu přerušení, a děje se následující:

1. Modul pro obsluhu přerušení zjistí, o jaké přerušení jde, a vytvoří datovou strukturu s údaji týkajícími se přerušení (typ, jak bylo vyvoláno, související data, ...).
2. Pokud kanál přerušení pro dané IRQ není sdílen, přímo se zavolá obslužná rutina, pokud však je kanál sdílen, volají se *postupně všechny obslužné rutiny* registrované na tento kanál, dokud procesor nedostane oznámení, že bylo přerušení obslouženo

⇒ součástí obslužné rutiny by mělo být zjištění, zda opravdu přerušení pochází od zařízení příslušejícího danému ovladači.

3. Po provedení obslužné rutiny je procesor přidělen některému z připravených procesů (může to být tentýž, který byl přerušen).

Jak bylo výše napsáno, (složitější) ovladač může být rozdělen na dvě poloviny – horní kritickou, která se musí provést hned, a dolní méně kritickou, která „může chvíli počkat“. Dolní polovina se dá implementovat několika různými způsoby. Nejběžnější jsou tyto:

- *tasklet* (časová kritičnost někde mezi obslužnou rutinou a běžným procesem, priorita nižší než obslužná rutina, ale vyšší než běžné procesy), spouští se jako *softwarové přerušení* na stejném procesoru jako původní přerušení,
- *pracovní fronta* (priorita nižší, obdobná jako u běžných procesů), běží v kontextu vlákna jádra, je běžným způsobem plánována na kterémkoliv procesoru,
- provedení *v rámci systémového volání*, to znamená v kontextu běžného procesu (nezáleží na rychlosti).

Na *víceprocesorovém* (vícejádrovém) systému existuje hlavní řadič přerušení a pak každý procesor (jádro) má svůj vlastní lokální řadič přerušení. Hlavní řadič přerušení rozděluje přerušení na jednotlivé procesory. Distribuce přerušení je zajištěna *tabulkou přerozdělování přerušení* (Interrupt Redirection Table, IRT), ve které je stanoveno pro každé přerušení, které procesory mají toto přerušení ošetřit. Přerušení může být přeposláno jednomu konkrétnímu procesoru, několika vybraným procesorům, všem anebo tomu procesoru, na kterém právě běží úloha s nejnižší prioritou.

Kromě běžných přerušení najdeme ve víceprocesorových systémech navíc přerušení, která slouží k zajištění komunikace mezi procesory (meziprocesorová přerušení).

✎ **Windows.** Ve Windows obecně platí o přerušeních velká část toho, co bylo napsáno výše o Linuxu, ale s určitými odlišnostmi.

Taktéž se používá tabulka přerušení IDT (ostatně, je to hardwarově závislý mechanismus), jen se trochu liší to, co se stane po zjištění příchozího IRQ. Zatímco v UNIXových systémech jsou postupně spouštěny obslužné rutiny registrované na dané IRQ a každá z nich musí testovat, zda přerušení přišlo či nepřišlo od ní, ve Windows je místo toho po sběrnici posílán dotaz na zjištění, které zařízení ve skutečnosti signál poslalo, a spustí se jen ta jediná obslužná rutina.

Další odlišnost je ve způsobu využívání priorit v souvislosti s přerušeními. Ve Windows se setkáme s úrovněmi IRQL (jsou podrobněji popsány v kapitole o synchronizaci, na straně 107). Oproti tomu v Linuxu (a obecně v UNIXových systémech) tento mechanismus nenajdeme, protože není přenositelný na různé hardwarové platformy. Windows totiž běží jen na několika málo různých hardwarových platformách, kdežto UNIXové systémy existují pro velké množství platforem.

Z výčtu možností komunikace ve Windows již víme, že například volání DPC (Deferred Procedure Call) se používá v podobné situaci jako v Linuxu tasklet: pokud v rámci obsluhy přerušení potřebujeme provést něco, co nesmí být součástí obslužné rutiny, dáme do do DPC, které je přerušitelné (na rozdíl od obslužné rutiny), ale má vyšší prioritu v rámci IRQL než procesy.

7.4 Spouštění nenativních aplikací

✎ Nenativní aplikace jsou aplikace psané pro jiný operační systém. Programy, které můžeme pro spouštění nenativních aplikací využít, lze rozdělit do několika skupin:

- *virtuální počítač* – provádí simulaci konkrétního 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, na něž vlastníme licenci,
- *emulátor operačního systému* – simuluje konkrétní operační systém včetně jeho různých součástí,
- *podsystém* pro spouštění aplikací jiného operačního systému, tedy neobsahuje nic „navíc“, je to jen přídavné rozhraní.

✎ Některé produkty mohou fungovat v tzv. *bezešvém módu*. To znamená, že aplikace spouštěná virtualizovaně „zapadá“ do prostředí reálného operačního systému, tedy má vlastní okno a samotný virtualizační produkt je pro uživatele prakticky neviditelný, s aplikací je možné zacházet stejně jako kdyby byla instalována přímo v hostitelském systému.

7.4.1 Virtuální počítač

✎ Tento typ pomocného softwaru je nejsložitější a často i nejpomalejší. Po instalaci obvykle můžeme nakonfigurovat simulovaný hardware (kterou HW platformu chceme používat, který hardware bude k dispozici), dále nastavíme BIOS, a pak můžeme instalovat operační systémy.

Každý z těchto programů má své typické vlastnosti, například existují emulátory sloužící čistě k emulaci konkrétní hardwarové platformy (pro Amigu, ZX Spectrum, PowerPC, herních konzolí, robotů – využívají především programátoři těchto zařízení) 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).

 U některých produktů se setkáváme s podporou tzv. *paravirtualizace*. Emulátor nemusí virtualizovat hardware, ale pouze vytvoří komunikační rozhraní uvnitř hostitelského systému, které předává požadavky na hardware od hostovaného (vnitřního) operačního systému do Ring0. Zatímco u plné virtualizace musí virtualizační software překládat binární kód do formy, které rozumí hardware, při paravirtualizaci již byl kód virtualizovaného systému předem upraven.

Podpora pravirtualizace musí být jak na straně virtualizačního softwaru (v současné době ji „umí“ například produkty od VMWaru a Citrixu), tak i na straně virtualizovaného operačního systému.

 Současné *procesory* obsahují *hardwarovou podporu virtualizace* (jde o rozšiřující instrukční sady). Virtualizované 64bitové systémy obvykle tuto podporu přímo vyžadují. Pokud virtualizační řešení běží nad procesorem podporujícím virtualizaci, je rozdíl v odezvě skutečného (hostitelského) a virtualizovaného operačního systému menší.

Pokud máme nainstalováno více než jedno virtualizační řešení, pak obvykle pouze jedno z toho může používat hardwarovou podporu virtualizace.

 Z nejznámějších programů:

VMWare Workstation běží pod Windows i Linuxem (hostitelské systémy), jako hostované systémy mohou být dovnitř nainstalovány prakticky kterékoliv. Je to jeden z nejlepších a nejoblíbenějších univerzálních simulátorů, komerční (existuje volně šiřitelná varianta pro osobní nekomerční použití, která umí pouze spouštět předpřipravené obrazy systémů). Podporuje bezešvý mód.

Produkty společnosti VMWare bývají tradičně v čele vývoje v této oblasti – zde se totiž komerčně úspěšná virtualizace začala vyvíjet.

VirtualBox je volně šiřitelný produkt společnosti Sun, běží na Windows, v Linuxu a MacOS. Uvnitř mohou běžet jak Windows, tak i Linux a různé UNIXové systémy. Podporuje i bezešvý mód.

Xen je volně šiřitelný, je běžnou součástí Linuxu a některých dalších UNIXových systémů. Dokáže pracovat jako nativní hypervizor (viz dále) a je základem pro další odvozené produkty (včetně komerčních).

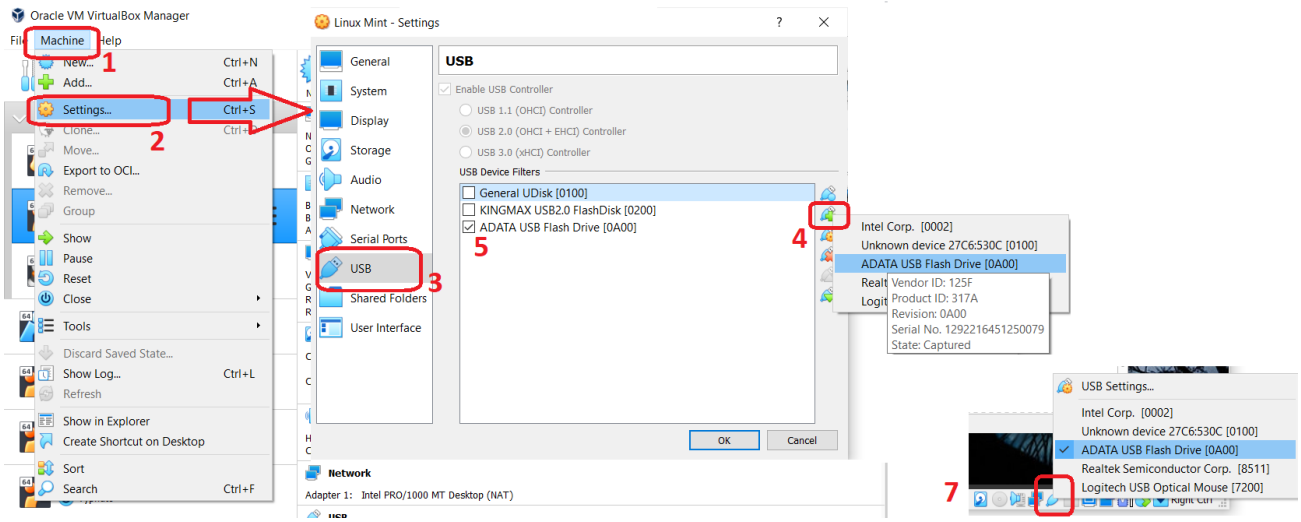
MS Virtual PC je distribuován Microsoftem (původně byl vyvíjen jinou firmou, Microsoft tuto firmu odkoupil), běží pouze pod Windows. V nejnovějších verzích je oficiálně podporován pouze běh různých verzí Windows coby hostovaných (vnitřních systémů), neoficiálně lze do tohoto produktu nainstalovat i Linux, ale na vlastní nebezpečí. Možnosti nastavení jsou spíše podprůměrné, a dokonce překvapivě nepodporuje MS Direct3D.

Qemu je svižný nástroj pro emulování různých hardwarových architektur; běží pod Linuxem, Windows i MacOS, je volně dostupný.

Parallels Desktop je komerční virtualizační nástroj běžící na MacOS, včetně variant s novějšími procesory typu ARM.

 **Poznámka:**

Pokud používáme VirtualBox pro běh virtuálních systémů a chceme uvnitř virtuálního stroje připojit (fyzický) USB flash disk (nebo obecně externí disk), jde to. Jen potřebujeme mít instalováno rozšíření Virtual Box Extensions (dá se stáhnout na stejné stránce jako samotný VirtualBox, pozor – musí odpovídat verze), a další postup je naznačen na obrázku 7.5.



Obrázek 7.5: USB flash drive inside VirtualBox

7.4.2 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 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).

Když si nainstalujeme emulátor operačního systému nebo podsystém, pak samozřejmě nemusíme instalovat žádný „vnitřní“ (hostovaný) operační systém a ani na něj nepotřebujeme vlastnit licenci. Instalujeme pouze aplikace, které chceme virtualizovaně spouštět, a to pomocí nástrojů poskytovaných emulátorem (podsystémem). Na aplikace už musíme licenci vlastnit, pokud to licenční podmínky vyžadují (EULA apod.).

Z nejznámějších emulátorů a podsystémů:

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. Jde o vlastní implementaci Win API (rozhraní, překladové vrstvy mezi aplikací a jádrem skutečného operačního systému).

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 výjimky). Programy, ale i jednotlivé jejich verze, jsou řazeny do skupin podle náročnosti zprovoznění ve Wine – platinum, gold, silver, bronze a ostatní.

Wine najdeme prakticky ve všech distribucích Linuxu a také v mnoha dalších UNIXových systémech. 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 společnosti Microsoft. Je určen ke spouštění různých, i náročnějších programů pro Windows, je využíván především pro spouštění her.

Proton je nástavba Wine pro běh her, používá se zejména pro provoz her ze Steamu.

PlayOnLinux, PlayOnMac je grafická nástavba Wine zjednodušující spouštění her určených pro Windows v Linuxu nebo v MacOS.

CrossOver je také komerční varianta pro Wine vyvíjená společností CodeWeavers, původně byl určen především do kanceláří, kde se využíval pro provoz kancelářských balíků, účetních a jiných ekonomických aplikací psaných pro Windows. V současné době je již univerzálněji používán a v seznamu podporovaných Win aplikací najdeme i mnohé známé hry, dále Adobe Photoshop nebo rozhraní .NET Framework. Snadnost zprovoznění je také hodnocena „medailemi“, podobně jako ve Wine.

CygWin je obdoba předchozích, ale funguje jako podsystém ve Windows pro spouštění linuxových aplikací (podobným způsobem jako Wine v Linuxu – sada knihoven plus mechanismus překladu API). Je volně dostupný a je často používán i tehdy, když chceme využívat nástroje Linuxu pod Windows (včetně shellu).

CygWin zahrnuje spoustu různých nástrojů (práce s textem, programovací nástroje, práce se sítí, dokonce i grafické prostředí a spoustu dalších) a během instalace rozhodujeme, které z těchto nástrojů si nainstalujeme.

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).

WSL (Windows Subsystem for Linux) je modul v jádře Windows (využívající Hyper-V), emuluje Linux.

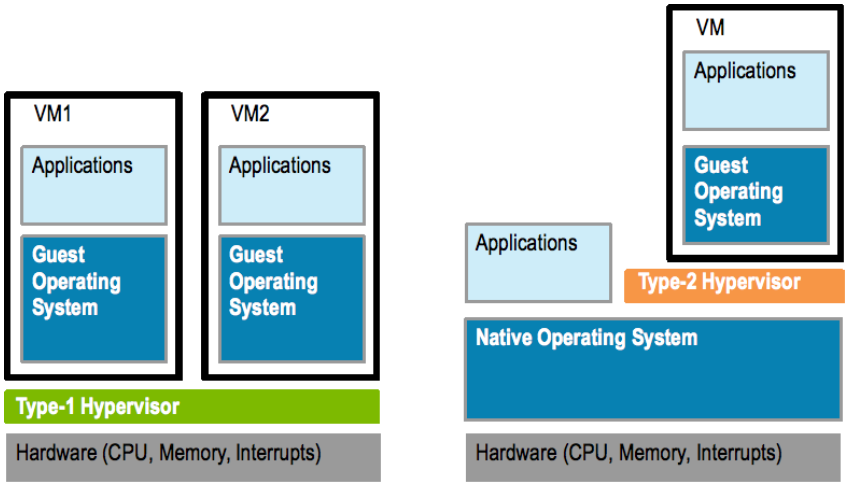
Další informace:

- https://appdb.winehq.org/objectManager.php?sClass=application (seznam podporovaných aplikací pro Wine, více než 10 tisíc)
- https://www.codeweavers.com/compatibility/ (seznam podporovaných aplikací pro CrossOver)
- https://cygwin.com/cygwin-ug-net/cygwin-ug-net.pdf (CygWin User's Guide)
- https://alternativeto.net/software/wine/ (několik alternativ k Wine)

7.4.3 Serverová a desktopová virtualizace

Serverová virtualizace. Ve firemním prostředí, především v datových centrech, se setkáváme s plnou virtualizací (nativní, serverovou). To znamená, že nejnižší vrstva celého systému (nejblíže hardwaru) není jádro některého operačního systému, ale je zde *nativní hypervizor* (také hypervizor typu 1) – tenká vrstva, která je vpodstatě obdobou jádra. Žádný z nainstalovaných operačních systémů není upřednostňován (alespoň u většiny těchto řešení).

Nad nativním hypervizorem pak běží virtuální stroje a v nich konkrétní operační systémy. Hypervizor je vlastně rozhraní, které zajišťuje transparentní provoz operačních systémů. Veškeré požadavky operačních systémů na hardware jsou směrovány hypervizorovi a systémy uzavřené ve virtuálních počítačích se navzájem nevidí, pouze hypervizor má přímý přístup k hardwaru.



Obrázek 7.6: Hypervizor typu 1 a typu 2[3]

Na obrázku 7.6 vidíme srovnání hypervizora typu 1 a typu 2. Hypervizor typu 2 je vlastně běžný virtualizační software, který instalujeme „do“ nativního operačního systému, třeba VMWare Workstation nebo VirtualBox. Hypervizor typu 1 umí serverovou virtualizaci, tedy se podsune pod operační systém, v rámci něhož se instaluje, a jakýkoliv další operační systém také „odřízne“ od přímého přístupu k hardwaru.

Kromě toho, že rozděluje prostředky mezi operační systémy, má i jiné role: například zprostředkovává komunikaci (může fungovat třeba jako virtuální switch propojující instalované systémy).

Produkty využívající serverovou virtualizaci (s nativním hypervizorem) jsou například

- VMWare ESXi Server,
- Xen,
- KVM (Kernel-based Virtual Machine) v Linuxu,
- Citrix XenServer (založen na projektu Xen),
- Microsoft Hyper-V ve Windows.

Xen a KVM jsou volně dostupné s otevřeným zdrojovým kódem, ostatní jsou komerční. U všech komerčních existuje zdarma dostupná varianta nebo alespoň demonstrační časově omezená verze.

Desktopová virtualizace. V datovém úložišti jsou uloženy obecné nebo personalizované obrazy desktopů (plných instalací OS a aplikací). Uživatel má buď tenkého klienta nebo jakýkoliv počítač s příslušným softwarem (specializovaný SW nebo stanovený webový klient, podle řešení). Uživatel se „přihlásí“ do firemní sítě, vzdáleně pracuje se „svým“ desktopem.

Existují dvě varianty – centralizovaná a distribuovaná. U centralizované varianty pracuje především procesor datového centra (princip hypervizora), na straně klienta je jen rozhraní. V druhém případě klient pracuje na plnohodnotném počítači, kde spustí obraz svého desktopu, přes cloud se řeší synchronizace rozdílů při změnách.

[3] Zdroj: https://microkerneldude.files.wordpress.com/2012/01/type1-vs-2.png

Účelem desktopové virtualizace je především možnost jednoduché hromadné správy desktopů, možnost provozovat tentýž desktop na různých zařízeních podle momentální pozice, a při vzdáleném přístupu také možnost práce z domova na tomtéž desktopu.

Opět se setkáváme především s produkty společností VMWare, Citrix, Microsoft.

Další informace:

- http://www.vmware.com
- http://www.vmware.com/support/ (zde je možné stáhnout si volně šiřitelné varianty produktů)
- http://www.citrix.cz/
- http://www.citrix.cz/downloads.html (stažení volně šiřitelných variant)
- http://www.microsoft.com/en-us/server-cloud/windows-server/server-virtualization.aspx
- http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx (stažení volně šiřitelné varianty)

Kapitola 8

Paměťová média

Rychlý náhled: 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 či SSD, což je paměťové médium). V této kapitole se zaměříme na správu paměťových médií, soustředíme se zejména na operace související se souborovými systémy a adresářovou strukturou.

Klíčová slova: paměťové médium, MBR, GPT, svazek, složka, adresář, adresářová struktura, souborový systém, žurnálovací systém, FAT, NTFS, exFAT, VFS, extXfs, Btrfs, Squashfs, procfs, sysfs.

Cíle studia: Po prostudování této kapitoly porozumíte problematice správy paměťových médií a struktuře dat v běžných souborových systémech.

8.1 Základní pojmy

Paměťová média mohou být buď napevno připojena datovým kabelem nebo přes jiné rozhraní (třeba M.2) k základní desce počítače, často přímo ve skříni počítače (například běžný pevný disk) nebo mohou být vyměnitelná a připojují se přes některé rozhraní vně skříně (například přes USB).

Poznámka:

Dále budeme předpokládat, že laskavý čtenář již má určité znalosti o hardwarové stránce paměťových médií (pevný disk vs. SSD vs. další typy paměťových médií, formátování, stopa, sektor, sekvenční vs. přímý přístup k médiu, LBA adresace,...).

Většina souborových systémů ukládá soubor do jedné nebo více oblastí, kterým říkáme *cluster* (terminologie z Windows) nebo *blok* (terminologie z ostatních operačních systémů). Jeden cluster či blok zabírá vždy určitý počet sektorů, obvykle 2, 4, 8, 16 nebo 32 sektorů. Konkrétní hodnota závisí jak na typu souborového systému, tak i obvykle na velikosti disku. Disky s velkou kapacitou obvykle mívají delší clustery/bloky, především kvůli adresaci (čím větší disk, tím víc clusterů/bloků bude potřeba, a každý musí mít LBA adresu).

Aby mohlo být paměťové médium používáno, musí na něm být předpřipravena určitá struktura (formát), 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 souborový systém FAT je vytvořena FAT tabulka a další potřebné struktury). 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 *oddíly* (svazky, oblasti, partitions podle terminologie v různých operačních systémech), na jednom fyzickém disku je vždy alespoň jeden oddíl.

Rozdělení se provádí pomocí k tomu určených nástrojů, v každém operačním systému máme takový. 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 nástroj z Windows a teprve pro zbytek disku nástroj z Linuxu).

8.2 Správa blokových zařízení

8.2.1 Vlastnosti blokových zařízení

Bloková zařízení se liší od znakových v mnoha ohledech. Kromě běžného množství přenášených dat také odlišným způsobem přístupu k těmto datům, například je možné se v proudu dat posouvat také zpět nebo obecně na zadanou adresu (*seek*).

Ještě výraznější odlišnost je ve způsobu přístupu k datům. Zatímco přístup ke znakovým zařízením bývá obecně jednodušší a snadnější, při přístupu k blokovým zařízením obvykle (ne vždy) využíváme rozhraní nabízené souborovými systémy.

Paměťová média jsou typickým zástupcem blokových zařízení, proto se zde budeme věnovat především tomuto typu zařízení. K blokovým zařízením můžeme také řadit některá virtuální zařízení, například šifrovací modul jádra.

Přístup k blokovým zařízením musí být řádně plánován a synchronizován. K tomu účelu obvykle souží modul jádra zvaný I/O Scheduler (I/O plánovač). Tento modul spravuje fronty požadavků na bloková zařízení a s použitím těchto front posílá požadavky ovladačům zařízení.

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).

Při adresování LBA má každý sektor na disku svou adresu, což je jedno číslo; kapacita disku a oddílu je omezena jak počtem bitů, do kterých lze adresu sektoru uložit, tak i velikostí tohoto sektoru. Takže pro zvýšení maximální kapacity disku a oddílu máme dvě možnosti – zvýšit počet bitů adresy (což může být problém) nebo zvětšit velikost sektoru.

Sektor na běžném disku typicky obsahuje 512 B (1/2 KiB) dat. To však neplatí pro disky označované jako *Advanced Format* – tyto disky používají 8× větší sektory, do jednoho sektoru se vejdou 4 KiB dat. Takže poslední sektor, který je nutné na nejnižší úrovni adresovat pomocí LBA, může být na disku „dál“ než při použití původních kratších sektorů.

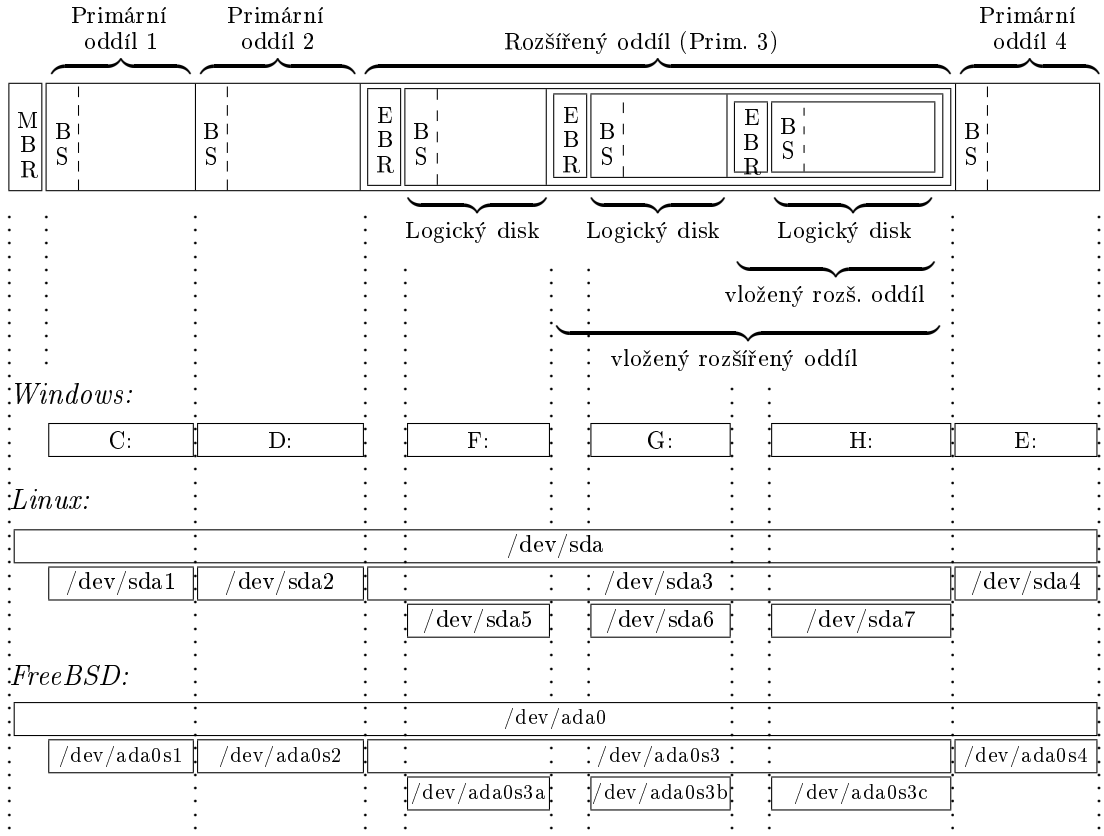
Dnes existují dva základní formáty disků (platí také pro SSD a externí disky):

- starší formát *MBR* (Master Boot Record),
- novější formát *GPT* (GUID Partition Table).

8.2.2 Struktura MBR disku

MBR disky mohou mít maximálně 4 *primární oddíly*, z nichž jeden může být označen jako *rozšířený*, v něm lze vytvořit jakýkoliv počet *logických disků*.

V některých operačních systémech (jmenovitě FreeBSD) se v případě MBR hovoří o slices místo o oddílech (a až pokud jde o rozšířený oddíl, pak vnitřní logické disky jsou označovány jako oddíly), ale na GPT disku to už jsou rozhodně oddíly.



Obrázek 8.1: Struktura MBR disku a označení v různých operačních systémech

Na obrázku 8.1 je naznačena struktura MBR 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. Popíšeme si některé části této struktury.

Zkratka MBR znamená *Master Boot Record* – hlavní zaváděcí záznam disku. Zde najdeme hlavní zaváděcí záznam (instrukce pro BIOS, které říkají, co se má stát, když je počítač spuštěn 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.

Další informace:

Terminologie FreeBSD je k nalezení například zde:

- https://docs.freebsd.org/en/books/handbook/basics/#disk-organization
- https://docs.freebsd.org/en/books/handbook/disks/

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*).

Zkratka EBR znamená *Extended Boot Record* – zaváděcí záznam rozšířeného oddílu. Je to obdoba MBR, obsahuje podobné informace. Také je tu tabulka rozdělení, tentokrát rozšířeného oddílu, a je zde místo pro dva záznamy: pro jeden logický disk (logický oddíl) a pak buď pro druhý logický disk nebo pro vložený rozšířený oddíl. Tento vložený rozšířený oddíl může být opět rozdělen na dvě části (logický disk a buď další logický disk nebo vnitřní vložený rozšířený oddíl), atd.

To znamená, že 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.

Do adresových polí ve strukturách na MBR disku je možné uložit jen čísla, která se vejdou do 32 bitů – pokud používáme adresaci LBA. Z toho důvodu je maximální velikost oddílu jen cca 2 TiB a adresa začátku oddílu se také musí vejít do 32 bitů (tj. poslední oddíl na disku musí začínat na adresách přibližně kolem 2 TiB a jeho velikost je maximálně kolem 2 TiB, tj. celková velikost disku je maximálně cca 4 TiB).

8.2.3 Struktura GPT disku

GPT je součást standardu UEFI od společnosti Intel. Jedná se již o 64bitový koncept, což dává více možností při adresaci (tj. oddíly mohou být i hodně velké). Na GPT discích se používá výhradně LBA adresace (CHS není vůbec podporována).

Můžeme mít až 128 oddílů na jednom GPT disku. Jeden oddíl může podle standardu zabrat až 9.4×10^{21} B = 8 ZiB, ale tak velké oddíly nejsou podporovány operačními systémy (tj. operační systém by měl problém s využíváním a adresováním tohoto oddílu), například Windows zvládnou max. oddíl o velikosti 18 EB.

Protective MBR (1 sektor)
zbytek GPT tabulky (33 sektorů)
oddíly
kopie GPT tabulky (34 sektorů)

Tabulka 8.1: Základní struktura GPT disku

Základní struktura GPT disku je naznačena v tabulce 8.1. GPT tabulka (tabulka rozdělení GPT disku) zabírá celkem 34 sektorů, z toho jeden sektor nazýváme *Protective MBR* a jeho účelem je zajištění kompatibility se staršími systémy. Operační systém, který nepodporuje GPT, považuje takový disk za běžný MBR disk s jedním oddílem, v němž jsou pro něj nečitelná data (ale pozná, že jde o disk, a to stačí). Zbytek GPT záhlaví je primární GPT tabulka.

V primární GPT tabulce najdeme tyto informace:

- verze GPT, velikost záhlaví,
- kontrolní součet záhlaví (počítá se s tímto polem nulovým),
- LBA adresa tohoto a záložního GPT záhlaví (to záložní je na konci disku),
- LBA adresa prvního oddílu, poslední adresa LBA použitelná pro oddíly (tj. poslední sektor těsně před záložní GPT tabulkou),
- GUID disku, v UNIXových systémech se označuje UUID,
- údaje o oddílech (pole záznamů o oddílech, před tímto polem jsou údaje o samotném poli – počet položek apod.).

V poli záznamů o oddílech (na konci GPT tabulky a vlastně celého GPT záhlaví) najdeme položky ke každému oddílu na disku, položka obsahuje tyto informace:

- GUID typu oddílu, pak GUID samotného oddílu (každý 16 B),
- adresa začátku a konce oddílu (každá 8 B),
- atributy (8 B), například read-only, systémový, skrytý,
- název oddílu (72 B).

Následují oddíly a na konci disku je záložní GPT záhlaví (kopie primárního záhlaví).

Poznámka:

MBR disk může být konvertován na GPT disk. Potřebný nástroj najdeme v různých operačních systémech, nicméně není dobrý nápad to provádět na používaném disku obsahujícím data nebo systém (sice to je s některými nástroji možné, ale nemáme zaručeno, že obsah disku neutrpí).

V Linuxu (klidně nabootovaném z USB flash disku či externího disku) můžeme zadat:

```
sgdisk -g /dev/sdc    #pokud je speciálním souborem /dev/sdc
```

Yrovna tento příkaz není nedestruktivní. Ve Windows můžeme použít Správu disků (`diskmgmt.msc`).

8.3 Svazky

Svazek (volume) je seskupení jednoho nebo více diskových oddílů, které mohou být i na různých paměťových médiích. Uživatel vidí vždy svazek jako celek, nemusí se zabývat hranicemi mezi jednotlivými oddíly ve svazku (a také je možné svazek libovolně rozšiřovat o další oddíly), což je hlavní výhoda používání svazků – transparentnost přístupu k oddílům.

Pokud ve svazku pracujeme s celými disky, pak obvykle bývají v některém typu pole RAID. Používá se buď zrcadlení (pro zajištění bezpečnosti dat) nebo prokládání (pro zajištění transparentnosti využívání oddílů ve svazku).

Používání svazků je vlastně forma virtualizace úložiště (storage virtualization). Níže jmenovaná řešení pro Windows a Linux potřebují především mapovací funkci, která určuje, pod jakou (logickou) adresou vidí daný operační systém konkrétní adresu v konkrétním (fyzickém) oddílu.

Implementace:

- *dynamické svazky* ve Windows, dají se konfigurovat například ve Správci disků (`diskmgmt.msc`): několik oddílů/disků můžeme shrnout do jednoho svazku a kdykoliv přidat další,
- *LVM* (Logical Volume Manager) v Linuxu, umožňuje jít „oběma směry“: buď shrneme několik fyzických oddílů/disků do jednoho logického, nebo naopak jeden fyzický disk rozdělíme na několik logických, které se dynamickým způsobem dělí o místo na skutečném disku (na rozdíl od běžných oddílů, u kterých je změna velikosti o něco složitější),
- souborové systémy ZFS, APFS, BtrFS a další mají už tuto funkcionalitu přímo integrovanou.

Další informace:

- https://medium.com/@yhakimi/lvm-how-to-create-and-extend-a-logical-volume-in-linux-9744f27eacfe
- https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-lvm.html

8.4 Organizace dat na paměťových médiích

8.4.1 Soubory a adresáře

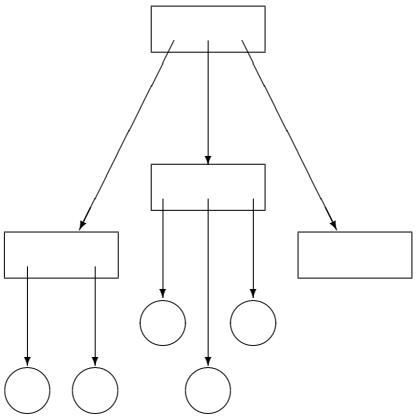
Paměťová média mohou obsahovat velmi mnoho dat, a aby bylo vůbec možné se v těchto datech vyznat, vyhledávat, používat je, přidávat další nebo některá odstraňovat, musí být vhodně organizována.

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). Rozeznáváme tyto základní typy souborů:

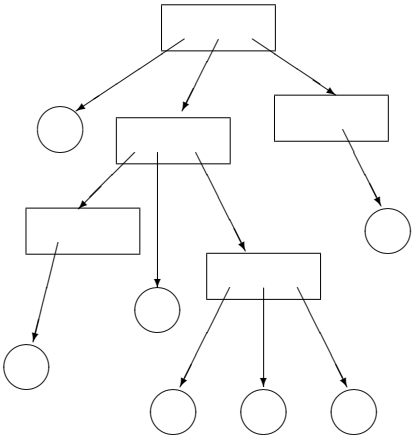
- standardní (dokumenty, spustitelné soubory),
- adresáře coby kontejnery souborů a jiných adresářů,
- simulované (pro přístup k I/O zařízení nebo pro mechanismus pipes a socketů),
- odkládací soubory pro virtuální paměť.

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 (především UNIXových) je transparentně chápán také jako soubor, třebaže se speciálním významem (obsahuje záznamy o obsažených podadresářích a souborech).

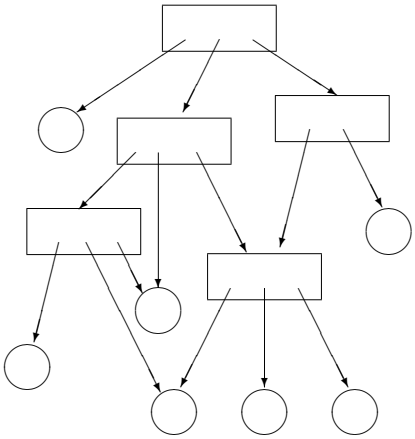
(a) dvouúrovňová struktura:



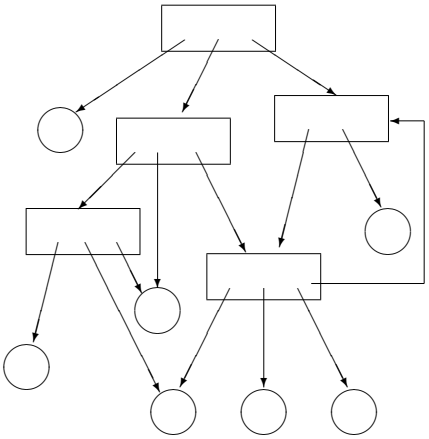
(b) stromová struktura:



(c) acyklická struktura:



(d) cyklická struktura:



Obrázek 8.2: Různé typy struktur adresářů

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.

Dvoúrovňová 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 jako základní pro své souborové systémy systém 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 jako základní struktura v UNIXových souborových systémech.

Cyklická struktura – k položkám může existovat více než jedna cesta, na rozdíl od předchozího řešení jsou dovoleny i cykly. Používá se pouze jako virtuální nástavba nad jinou strukturou.

Může jít například o systém symbolických odkazů v UNIXových souborových systémech nebo zástupců ve Windows. Tyto odkazy jsou krátké soubory s informací o skutečné adrese položky a případně dalšími informacemi.

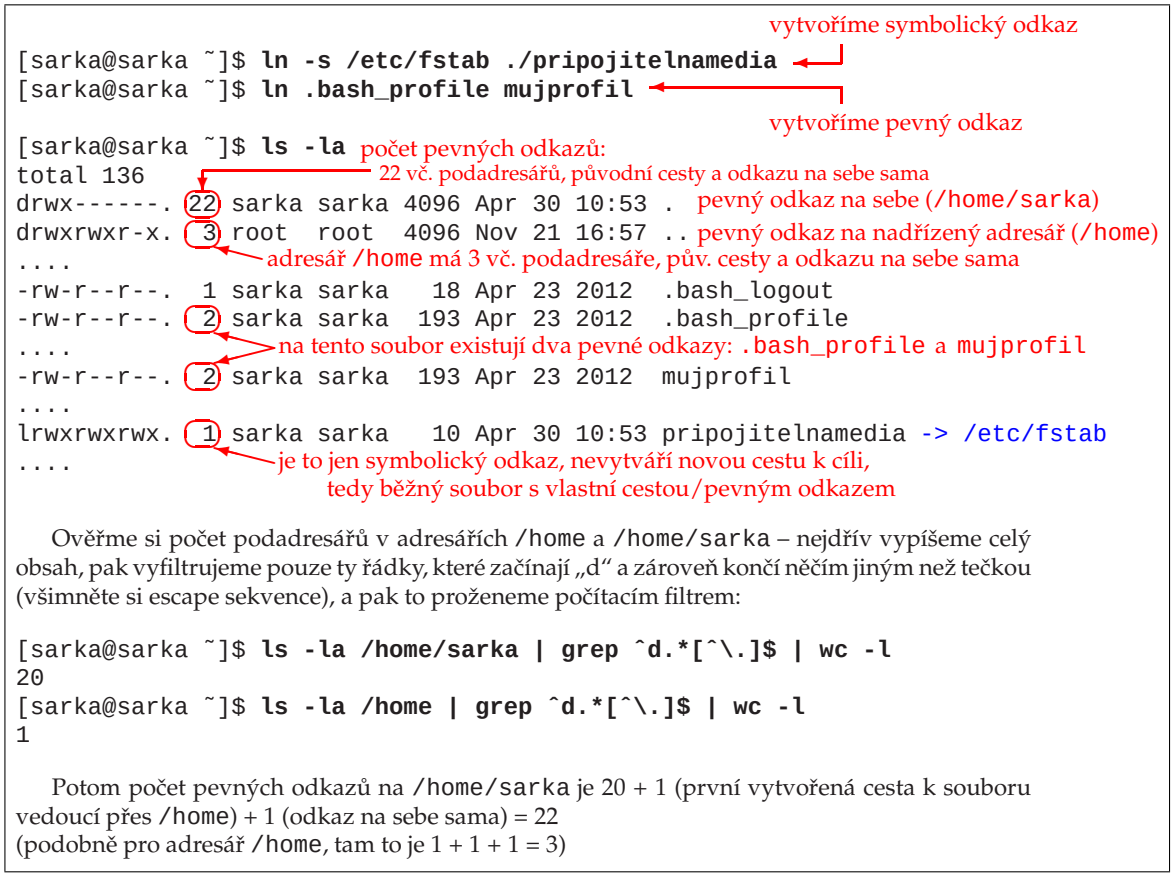
 U acyklické struktury může být problémem *zachování acykličnosti grafu* při přidávání nových adres do adresářů. To lze řešit více způsoby. Nejjednodušším způsobem je omezení týkající se vícenásobných adres – ve více adresářích může být jen soubor, nikoliv adresář (tj. když vytváříme alternativní cesty, mohou vést jen na běžné soubory, ne na adresáře), 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 `..`, vyhledávací algoritmus si pak nevšímá adresářů takto pojmenovaných, protože jde pouze o další alternativní cesty k těmto adresářům.

 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.

V UNIXových systémech, kde jsou běžné acyklické souborové systémy, se používá první způsob (tj. čítač, jde o čítač pevných odkazů na soubor).



Obrázek 8.3: Ukázka zjištění počtu pevných odkazů na soubor v Linuxu

8.4.2 Systémy souborů

Souborový systém (systém souborů) jsou metody a struktury dat, pomocí kterých operační systém udržuje záznamy o souborech. 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 vlastníka, přístupová práva k tomuto souboru, atributy a další vlastnosti. To je realizováno různými způsoby, z nichž některé budou podrobněji popsány dále.

Některé možnosti evidovaných položek:

a) Souborové systémy typu FAT (Windows s DOS jádrem a jiné systémy) – 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) Multics – každý soubor obsahuje jako metadata kompletní seznam uživatelů s přístupovými právy.

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

d) UNIXové souborové systémy – práva `r` (číst), `w` (zapisovat), `x` (spouštět). Každé položce se přiřazují tato práva pro vlastníka, přidruženou skupinu 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. Navíc jsou k dispozici ACL a atributy.

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

1. *Souborové systémy s okamžitým zápisem* (FAT, FAT32) – pokud proces chce zapisovat na disk a zároveň probíhá jiná disková operace, musí počkat. Výhodou je bezpečnost (data se nemohou neočekávaně ztratit bez toho, aby to proces „nevěděl"), nevýhodou snížení propustnosti (čekání při práci s diskovým oddílem).
2. *Souborové systémy s opatrným zápisem* (HPFS) – rozdělí zápis do posloupnosti dílčích operací, u kterých není pravděpodobné, že by mohly být přerušeny (trvají krátkou dobu, kondenzátory chvíli energii udrží). Když dojde s selhání při zápisu, data zůstanou konzistentní (žádná dílčí operace nezůstane „viset"). Vlastně se jedná o jednoduchý databázový systém s transakcemi.
3. *Souborové systémy s opožděným zápisem* – to znamená použití vyrovnávací paměti (cache), tedy data se nejdřív zapisují do cache paměti, zapisující proces může dále pracovat. Z cache paměti se data zapíšou na disk až tehdy, když disk dokončí předchozí probíhající operaci. Výhodou je zvýšení propustnosti systému (procesy nejsou zdržovány zápisem na disk), nevýhodou je možnost ztráty dat při havárii. Tento typ se sám o sobě nepoužívá, je základem pro následující.
4. *Žurnálovací souborové systémy* (journalized, zotavitelné – NTFS a většina UNIXových souborových systémů) si uchovávají informace o probíhajících operacích (tak jako v systémech s opatrným zápisem, plus soubor s evidencí), aby bylo možné v případě výpadku dostat data zpět do konzistentního stavu.

V žurnálovacím systému jsou změny evidovány podobně jako v databázích jako transakce. *Transakce* se skládá z jednoduchých (atomických) operací, navzájem oddělitelných, tyto operace se postupně evidují v *žurnálu* (log souboru). 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, tedy do stavu před provedením změn. Sice požadovaná transakce nebyla provedena, ale data máme v konzistentním stavu.

Jinou možností by bylo dokončení posloupnosti operací dané transakce (podle žurnálu bychom zjistili místo přerušení), ale obvykle není možné dodatečně zjistit, jaké operace je ještě třeba provést (ta informace byla ztracena při zmíněném výpadku proudu). Tedy tato možnost se nepoužívá.

Příklad

V případě NTFS žurnálování probíhá takto:

- během každé operace na disku jsou dílčí operace zaznamenávány do žurnálu (logu), po ukončení operace včetně vymazání z cache jsou všechny tyto dílčí operace z logu vymazány,
- po startu systému se prochází tento LOG soubor a *opakují se všechny dokončené transakce*, které nestihly být odstraněny z logu (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).

Cílem žurnálování je zabránit ztrátě dat při poruchách, a to zejména metadat o souborech. Některé souborové systémy tuto ochranu provádějí jiným způsobem: například ZFS (v BSD systémech) a APFS (Apple) místo přepisování metadat (během změn a přidávání souborů) vytvářejí nové struktury, při zotavení tedy stačí přesunout se k předchozím metadatům.

✎ **Virtuální souborový systém** je takový souborový systém, který nemá přímou vazbu ke konkrétnímu paměťovému médiu. Virtuální souborové systémy se používají k abstrakci přístupu k ostatním souborovým systémům (především v UNIXových systémech), jako filtr pro průchozí transformace dat 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 komunikační rozhraní.

✎ **Souborové systémy pro vyměnitelná média:** Na vyměnitelných médiích 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), jehož jiné označení je ISO 9660. Pro DVD a Blu-Ray, ale i pro CD, je to UDF (Universal Disk Format), USB flash disky mívají souborový systém FAT32 (VFAT), exFAT nebo ext2fs.

8.5 Souborové systémy ve Windows

8.5.1 Starší verze souborového systému 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 diskového oddílu.

Nejdřív se podíváme na strukturu jednodušší varianty (FAT16) a pak si ukážeme, co navíc funguje v novějším FAT32 (FAT64 neexistuje).

✎ **FAT16.** Délka clusteru je pro velmi malé oddíly obvykle 2 sektory (1 KB), se zvyšující se kapacitou je tato hodnota výrazně vyšší, určuje se napevno podle velikosti oddílu. Struktura oddílu 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 oddílu
- její kopie (použitelná v případě, že se první FAT poškodí)
- *root* (hlavní adresář oddílu) – zvláštní struktura s pevnou délkou, proto v hlavním adresáři oddílu 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). Ve skutečnosti se pro čísla clusterů nepoužívají všechny možné hodnoty, některé jsou vyhrazeny a vytvářejí speciální kódy například pro vadný cluster nebo poslední cluster souboru.

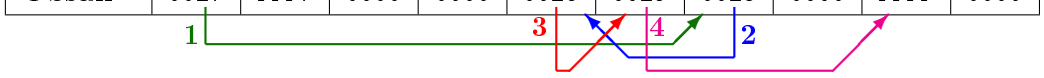
Obsah záznamů v tabulce určuje, co v příslušném clusteru najdeme. Jestliže je cluster volný, je zde číslo `0x0000`, vadný – číslo 0xFFF7 (toto číslo zde zapisují programy pro kontrolu povrchu disku).

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

Soubor začíná na clusteru s číslem `0x0021` (to jsme zjistili v informační položce pro daný soubor, viz dále), 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 8.2: 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 oddílu), 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 oddílu.

Zatímco běžný soubor obsahuje jakákoliv data, adresář se skládá z *položek* (záznamů) o délce 32 B popisujících soubory a podadresáře pro daný adresář. V položce jsou evidovány následující informace o konkrétním souboru, adresáři či labelu:

- název souboru či podadresáře (8 B),
- přípona souboru (3 B),
- pokud položka představuje label, tedy název oddílu, 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 oddílu, 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 oddíly velikosti od 512 MB do 1 GB stanovena na 16 KB, pro větší (do 2 GB) na 32 KB. Udává se, že systém FAT16 není použitelný pro oddíly větší než 4 GB, a není prakticky použitelný pro oddíly 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).

8.5.2 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, bez potřeby nástavby.

Název souboru nebo adresáře ve VFAT je 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 čtyři typy:

- položka pro soubor,
- položka pro adresář (složku),
- položka pro label (jmenovku) oddílu,
- 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 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 některých dalších znaků, případně jejich nahrazení, konec je odříznut a nahrazen identifikací rozlišující soubory se stejným zkráceným názvem.

Příklad

Někdy se může hodit i krátký název typu 8.3. Bohužel se čím dál častěji setkáváme s uživateli, kteří pojmenovávají soubory s použitím diakritiky (čárky a háčky nad písmeny). Pokud takový soubor zůstává na strojích se stejným operačním systémem, až tak to nevadí. Problém může nastat při doručení na stroj s jiným operačním systémem, různé souborové systémy totiž mají odlišná pravidla pro používání tohoto typu znaků.

Pokud je do Windows doručen (třeba e-mailem) soubor z MacOS a v jeho názvu je diakritika, takový soubor můžeme bez problémů uložit. Ale ve chvíli, kdy chceme tento soubor otevřít, objeví se chybové hlášení. Pokus o přejmenování nebo alespoň smazání souboru taky nedopadne dobře, opět chybové hlášení. Jak postupovat?

Nejdřív zjistíme krátký název souboru (předpokládejme, že soubor je v našem profilu):

```
C:\Users\sarka>dir /x
...
24.09.2024  13:24          52 412   njakob~1.jpg   nějaký obrázek s~divným názvem.jpg
...
```

V posledním sloupci je plný název souboru, před ním je zkrácený název ve formátu 8.3. Zůstávají pouze znaky ze spodní poloviny ASCII tabulky (anglická abeceda, číslice apod.), ostatní jsou odstraněny, pak je název zkrácen (v případě potřeby i přípona). Pokud by pro několik souborů vycházel stejný řetězec, k odlišení slouží číslo za vlnkou.

Krátký název můžeme použít například pro přejmenování:

```
C:\Users\sarka>ren njakob~1.jpg "nejaky obrazek s~divnym nazvem.jpg"
```

Nový název (dokonce i v případě, že v něm použijeme diakritiku) se už dá běžně používat v textovém i grafickém rozhraní.

Microsoft uvádí, že dlouhé jméno lze použít i pro label oddílu, ovšem reálně mohou nastat problémy s kompatibilitou 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 rozmezí MIN–16 (nebo 32) sektorů, podle verze Windows, kde hodnota MIN se řídí velikostí oddílu:

Velikost oddílu	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 8.3: 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.

Ve FAT32 je oproti FAT16 i řada dalších změn:

- je použitelná pro oddíly větší než 2 GB (ale pro oddíly 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.

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.

8.5.3 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), ale při přechodu na verzi 4 byl značně přepracován, proto mnohé nástroje, které nějakým způsobem závisejí na NTFS, často vyžadují alespoň verzi 4 (například nástroje pro změnu velikosti oddílu). Hlavním požadavkem při jeho vyvíjení bylo zajištění větší bezpečnosti dat, především možnost definování přístupových práv pro různé uživatele. Je určen pro velké oddíly, lze ho použít i na malé oddíly.

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, přesněji DACL) 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, přehled nástrojů a jejich používání jsme měli na cvičeních předmětu Operační systémy).

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.

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

`$MFT` (Master File Table) je obdoba FAT tabulky ve FAT systémech, obsahuje záznamy o všech souborech na oddílu (MFT je taky soubor, proto zde najdeme i záznam o MFT). Záznam v této

tabulce má obvykle 1 KB, ale může být delší. V každém záznamu je především odkaz na umístění začátku souboru, bezpečnostní nastavení, atributy, ...

`$MFTMIRR` je zrcadlo MFT (kopie), ale ne celá; jsou tu jen první čtyři záznamy.

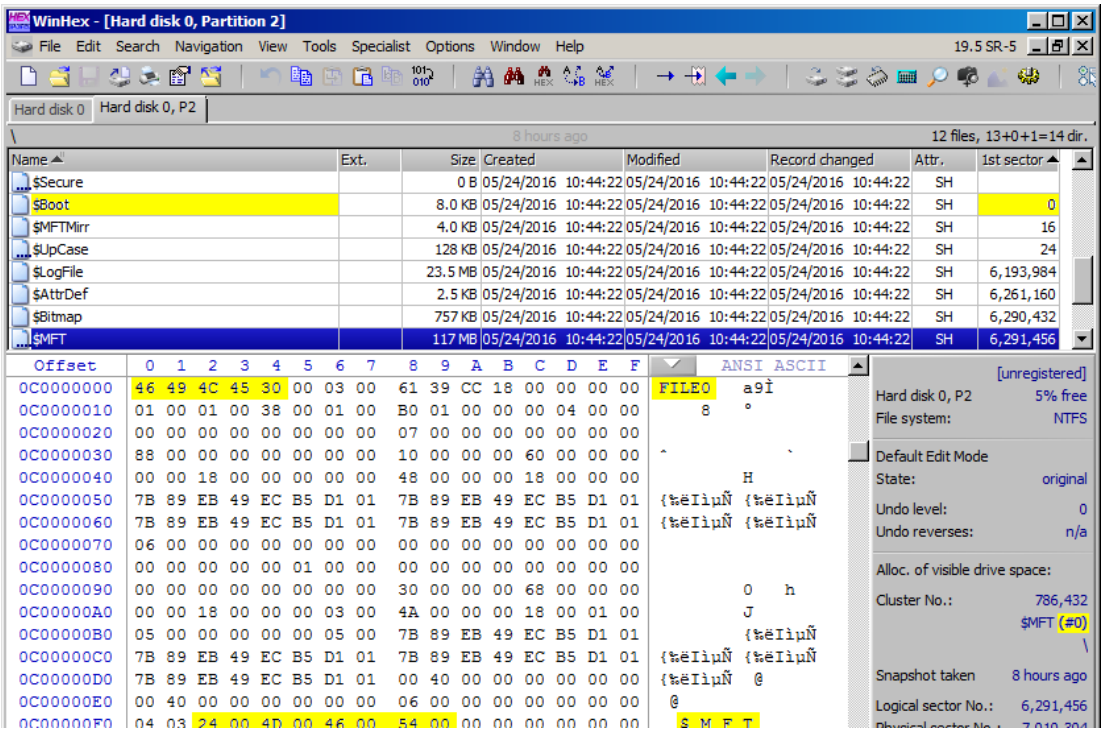
`$LOGFILE` je log soubor (žurnál), do kterého se ukládají transakční informace.

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

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

`$QUOTA` obsahuje informace o uživatelských kvótách. Atd.

V běžných souborových manažerech, ve kterých pracujeme se soubory, jsou tyto speciální soubory neviditelné, i když existuje způsob, jak je zviditelnit – viz obrázek 8.4.



Obrázek 8.4: Metasoubory na oddílu se souborovým systémem NTFS, náhled v programu WinHex[1]

Alternativní datové streamy (ADS). Neviditelné jsou také všechny datové streamy souboru kromě hlavního (neviditelné pro ty, kteří neznají `dir /r`). Zobrazovaná délka souboru se také týká hlavního streamu, takže po smazání jednoho malého souboru by se teoreticky mohlo stát, že na oddílu je najednou o několik KB více volného místa.

Hlavní stream souboru není pojmenován, jde vlastně přímo o data souboru, ostatní proudy jsou pojmenované. Ve streamech může být cokoliv, v sekundárních streamech se například v Office dokumentech ukládá autor a informace o obsahu souboru, celkově ale záleží na programátorovi aplikace vytvářející soubor.

Pokud zkopírujeme soubor s alternativními streamy na oddíl nepodporující ADS (například na USB flash disk se souborovým systémem FAT32), zkopíruje se pouze hlavní stream. Můžeme to poznat podle varování, že některá data nebude možné zkopírovat.

[1] Zdroj: https://thestarman.pcministry.com/asm/mbr/IntNTFSfs.htm

Příklad

Krátká ukázka vytvoření a zjištění alternativního streamu v souboru:

```
C:\Users\sarka>echo ahoj > souborADS.txt:prvni

C:\Users\sarka>echo hello > souborADS.txt:druhy

C:\Users\sarka>dir souborA*
 Volume in drive C is OS
 Volume Serial Number is 88C1-C862

 Directory of C:\Users\sarka

04.12.2024  12:37                 0 souborADS.txt
               1 File(s)              0 bytes
               0 Dir(s)  12 315 029 504 bytes free

C:\Users\sarka>dir /r souborA*
 Volume in drive C is OS
 Volume Serial Number is 88C1-C862

 Directory of C:\Users\sarka

04.12.2024  12:37                 0 souborADS.txt
                                  8 souborADS.txt:druhy:$DATA
                                  7 souborADS.txt:prvni:$DATA
               1 File(s)              0 bytes
               0 Dir(s)  12 314 836 992 bytes free
```

Jak vidíme, při běžném způsobu zobrazení se soubor jeví jako prázdný (má délku 0) a jeho další streamy nevidíme. Pokud však použijeme přepínač `/r`, zjistíme, že soubor má další streamy a jejich délka není 0.

Fragmentace. NTFS se brání fragmentaci tak, že pro uložení souboru hledá vždy ne nejbližší, ale *nejbližší vhodnou* posloupnost navazujících clusterů (samozřejmě takovou, ve které je tolik místa, že se tam soubor vejde, ale nevyužitý zbytek je co nejmenší), takže fragmentace vzniká pouze tehdy, když je na oddílu 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.

Další vlastnosti NTFS:

- Všechno je soubor (tedy také všechny implicitní struktury, metadata, na oddílu jsou implementovány jako speciální soubory).
- Názvy souborů mohou být v UNICODE (sice zaberou více místa na oddílu, ale nebývají tak velké problémy se znaky nepatřícími do anglické národní znakové sady).
- Možnost *indexace* podle různých vlastností souborů. 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 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). 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 oddílu nezabírají žádné místo.
- *Přípojné body* pro oddíly a výměnná média mohou být i složky, nemusí to být „písmeno a dvojtečka“.
- Používání *kvót*.

Velikost (implicitní) clusterů je stejně jako v FAT systémech odvozena od velikosti svazku podle tabulky (tab. 8.4), 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 8.4: Velikost clusteru pro souborový systém NTFS

Poznámka:

NTFS ve své implicitní podobě snižuje propustnost systému. Na rychlejších počítačích to nevadí, ale jinak existují způsoby, jak jeho práci zrychlit. Užitečný a celkem logický je tento způsob: NTFS dokonce 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. Tato úprava je velmi vhodná také u SSD.

8.5.4 exFAT

exFAT (Extended FAT) trochu vybočuje z řady jiných souborových systémů od Microsoftu. Je optimalizován pro USB flash disky a SD karty. Dá se říct, že svými vlastnostmi stojí někde mezi FAT32 a NTFS.

Je výrazně jednodušší než NTFS (také rychlejší) a zapisuje méně metadat na médium, tedy méně opotřebovává flash čip (víme, že flash paměti mají omezenou životnost co se týče maximálního počtu zápisu, paměťové buňky se opotřebovávají).

Oproti FAT32 je exFAT schopen ukládat větší soubory, velikost svazku (oddílu) může být větší, taktéž velikost clusteru (což souvisí). Ve specifikaci najdeme i podporu ACL, ale netýká se všech podporovaných operačních systémů. Podporuje sice transakce (žurnálování), ale jen tehdy, když je tato vlastnost podporována výrobcem dotyčného zařízení.

Stejně jako u FAT32, i zde se setkáme s FAT tabulkami (také v podobném významu – zřetězení clusterů), ale existují i další struktury. Podobně jako NTFS, i zde existuje struktura evidující volné clustery (ta v FAT32 není).

Jedná se o proprietární souborový systém, jeho specifikace není veřejně přístupná. Od toho se odvíjí omezenější podpora v některých operačních systémech. Obecně platí, že exFAT je podporován ve Windows od verze Vista SP1 a novějších, pro starší verze existuje záplata přidávající ovladač pro exFAT. MacOS podporuje exFAT od verze 10.6.5. Pro Linux existuje ovladač využívající modul FUSE.

8.5.5 Srovnání souborových systémů pro Windows

Pro velikost oddílu a maximální možnou velikost souboru platí tabulka 8.5.

	Max. velikost oddílu	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	256 TB bez 64 KB (pro 64KB cluster) 16 TB bez 4 KB (pro 4KB cluster)	$2^{64}-1$ (XP: $2^{32}-1$)	nedef.	2^{64} B bez 1 KB (XP: 2^{44} B bez 64 KB)	$2^{32}-1$
exFAT	128 PB	cca 2^{32}	nedef.	16 EB	nedef.

Tabulka 8.5: Srovnání souborových systémů pro Windows

Další informace:

http://www.ntfs.com/ntfs_vs_fat.htm

8.6 Souborové systémy pro Linux

8.6.1 VFS

Linux pracuje s virtuálním souborovým systémem *VFS* (Virtual File System), přes který jsou přístupné všechny „reálné“ souborové systémy na počítači. Jde o modul jádra, přes který jdou všechna volání diskových služeb, zastřešuje souborové systémy na všech svazcích a discích přítomných v systému (včetně výměnných médií) 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 diskový oddíl používat, musíme ho připojit (mount) do VFS buď v grafickém rozhraní nebo v konzole příkazem `mount`. Systémový oddíl 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, v grafickém prostředí je to opět většinou řešeno automaticky.

V Linuxu se na oddílech pevných disků nejčastěji používá souborový systém `ext4fs`, můžeme používat také souborové systémy Windows a jiných operačních systémů, jsou mapovány pod těmito názvy:

`msdos` kompatibilní s FAT12 nebo FAT16 bez VFAT,

`vfat` pro FAT32 nebo FAT16 s nástavbou VFAT,

`ntfs` kompatibilní s NTFS Windows NT, často bývá implicitně nastavena pouze možnost čtení, obvykle ovladač `ntfs-3g` nebo jiný podobný,

`iso9660` CD-ROM, totéž co pod Windows CDFS,

`procfs, sysfs, tmpfs, ramfs, devfs, udev` další virtuální souborové systémy připojené do VFS,

`FUSE` pro podporu běhu ovladačů v uživatelském prostoru,

`NFS` síťový souborový systém,

`swap` určený pro odkládací oddíl.

V UNIXu a Linuxu platí, že „všechno je soubor“, se zařízeními se také pracuje jako se soubory.

8.6.2 Souborové systémy typu ext*x*fs

Strukturu těchto souborových systémů si vysvětlíme na tom nejjednodušším, u dalších se zaměříme na rozdíly.

Oddíl na disku je rozdělen na *bloky*, jejichž velikost je možné předem stanovit (obvykle 1024, 2048 nebo 4096 B – několik sektorů). Je to obdoba toho, co ve Windows nazýváme cluster.

ext2fs. První blok oddílu, *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 je zachyceno, jak může vypadat struktura od s ext2, která je dlouhá 20 MB s délkou bloku 1024 B. 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 (i-node) je struktura obsahující důležité informace o souboru (ID vlastníka, skupiny, typ souboru, oprávnění, časová razítka, počet pevných odkazů, ...) 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).

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 jednotlivé i-uzly, tedy i-uzel je jednoznačně 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 oddílu se souborovým systémem ext2fs

Soubor použije bloky jen po tu úroveň, která mu stačí.

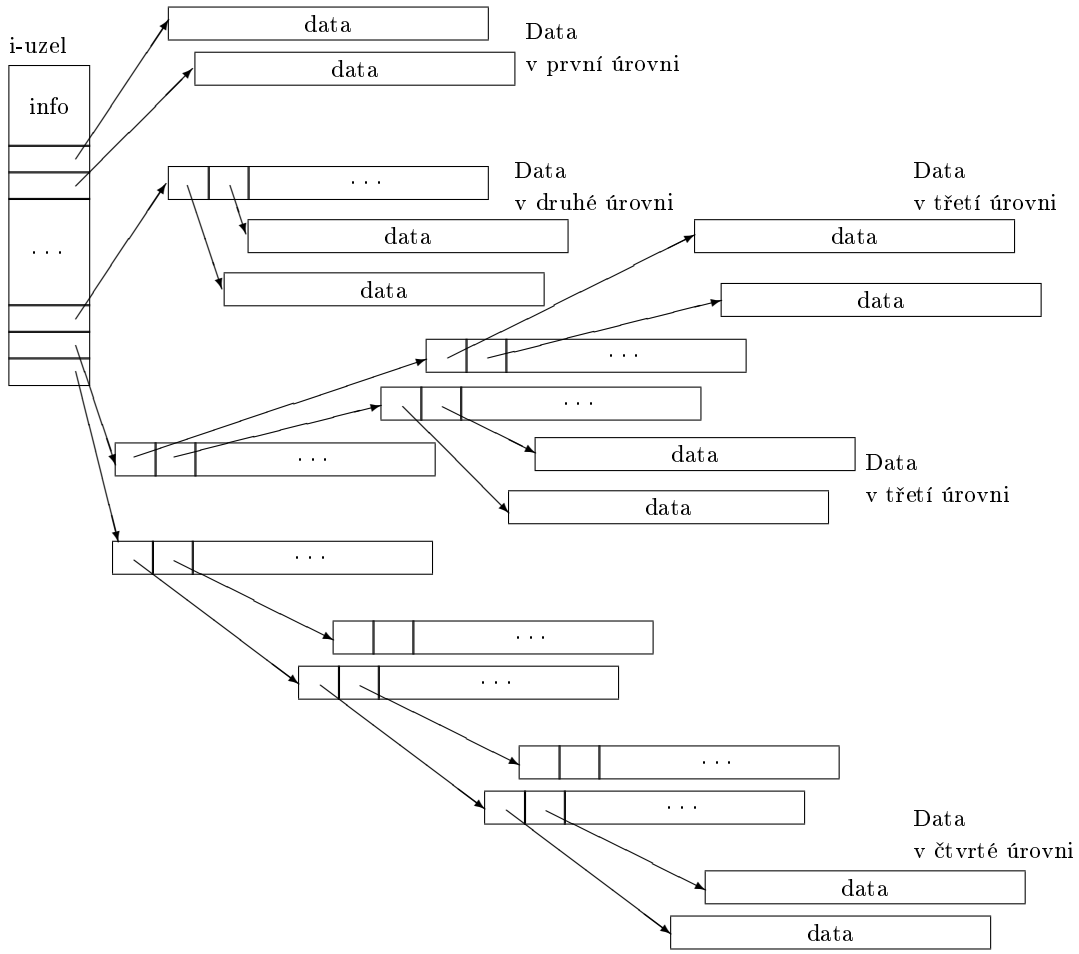
Obrázek 8.5 je zkrácenou ukázkou struktury souboru v souborovém systému ext2.

Příklad

Předpokládejme, že pro adresy se používá 32 bitů, tedy 4B, a délka bloku je 1024 B (1 KB). Podíváme se na možné limity.

- 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 (a taky provděpodobně bude) jiná velikost bloků než 1024 B.



Obrázek 8.5: Struktura souboru v souborovém systému ext2fs

Další informace:

https://wiki.osdev.org/Ext2

Každý *adresář* může obsahovat soubory nebo další adresáře. Adresáře jsou soubory obsahující posloupnost záznamů o vnořených položkách, kterým říkáme *položka adresáře* (directory entry).

Položka adresáře je tedy struktura informací o konkrétním souboru (ať už běžném, adresáři či speciálním). Každá položka adresáře obsahuje číslo i-uzlu (přes které se dají dohledat další informace o souboru), délku položky (pozor, ne souboru), délku názvu, typ a název souboru:

```
#define EXT2_NAME_LEN 255

struct ext2_dir_entry_2 {
  __u32  inode;              /* Inode number */
  __u16  rec_len;            /* Directory entry length */
  __u8   name_len;           /* Name length */
  __u8   file_type;          /* File type */
  char   name [EXT2_NAME_LEN];    /* File name */
};
```

Pokud si vypisujeme obsah adresáře (například příkazem `ls -l`), získáme údaje, ke kterým se dá dostat právě v těchto položkách nebo přes číslo i-node.

Adresář, stejně jako každá jiná struktura na oddílu, je také chápán jako soubor, proto má svůj i-uzel a může být rozprostřen ve více blocích stejně jako jiné soubory.

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 bloku.

Můžeme používat také odkazy (links). U *pevného odkazu* (hard link) je několik názvů souboru 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 oddíly).

Symbolické odkazy (soft link) obsahují 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.

Pro ext2fs se udává, že je použitelný pro oddíly 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, používají se jeho nástupci ext3fs, ext4fs. 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`). Důvodem větší rychlosti je, že se nepoužívá žurnál.

ext3fs. Tento souborový systém je zpětně kompatibilní s ext2fs (přesněji kompatibilní v obou směrech), zachovává všechny struktury ext2, ale navíc jde o žurnálovací souborový systém . Pokud máme na oddílu 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.

Ext3fs přišel s některými vylepšeními, například u velmi objemných adresářů lze jejich obsah (tedy položky pro vnořené soubory a podadresáře) evidovat jako hash-tree (HTree), tedy místo posloupnosti položek máme vyvážený strom, ve kterém jsou položky umístěny podle daného klíče a lze se mezi nimi tedy rychleji pohybovat.

ext4fs. Tento souborový systém je další verze souborových systémů ext. Je také zpětně kompatibilní s určitými omezeními. Oproti ext3 má kromě jiného tyto vlastnosti:

- limity udávané pro ext3 navyšuje pro použití na 64bitových systémech (je to 64bitový systém), včetně maximální velikosti oddílu a souboru,
- časová razítka v žurnálu jsou přesnější (1 ns),
- je šetrnější k flash pamětem (včetně SSD),
- může používat *extenty* – extent je souhrn více bloků za sebou následujících, místo ukazatele na blok dat lze použít ukazatel na extent (důsledkem je možnost uložení rozsáhlejších souborů s menší fragmentací).

Souborový systém ext4 lze připojit jako ext3, pokud nejsou používány extenty.

8.6.3 Další žurnálovací souborové systémy

ReiserFS. Tento souborový systém byl původně implicitně nabízen při instalaci některých distribucí, například SUSE (RedHat a Mandrake zase prosazují spíše ext systémy), v současné době je už na ústupu. Je to žurnálovací souborový systém, tedy při výpadku je větší pravděpodobnost, že data na oddílu zůstanou konzistentní.

ReiserFS je založen na *rychlém vyváženém stromu* (ballanced tree), 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 ext, FAT, NTFS každý blok vyhrazují pro určitý soubor, soubor může mít více bloků, ale ne naopak), takže na oddílu 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.

XFS. Jde o žurnálovací souborový systém, který svými přístupovými algoritmy snižuje zatížení systému způsobené používaním žurnálování. V reálu je toho dosaženo tak, že se žurnálují pouze metadata, nikoliv běžná data. To zvyšuje propustnost souborového systému, ale také je to důvodem nevhodnosti tohoto souborového systému pro nasazení na strojích s často modifikovanými daty.

Je to 64bitový souborový systém optimalizovaný pro práci s velkými soubory, kdežto práce s malými soubory už tak optimální není.

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. Z toho vyplývá, že XFS může být dobrá volba i pro menší server sloužící jako úložiště multimediálních dat.

Celkově je tento souborový systém díky omezením v žurnálování považován za méně bezpečný než ext4fs. Je ale velmi vhodný na ty servery, na kterých jsou data především čtena a méně modifikována.

BtrFS (B-tree File System). BtrFS je jeden z novějších souborových systémů (od společnosti Oracle) určených původně pro servery. Evidence souborů je založena na vyvážených B-stromech, jak název napovídá, a také v dalších vlastnostech je hodně inspirován souborovým systémem ZFS (ten se používá například v Solarisu a také ho můžeme najít v některých BSD systémech.

Oproti běžným linuxovým souborovým systémům je přidána podpora vlastností, které jsou ceněny hlavně na serverech – správa bez nutnosti odpojení (včetně defragmentace, vyvažování, kvót, apod.), vytváření obrazu svazku (snapshot) bez nutnosti odpojení (využívá možnost vytváření redundantních kopií souborů), což se dá využít při vytváření záloh včetně rozdílových, nativní podpora RAID 0, 1 a 10, používání kontrolních součtů, transparentní komprese, atd. Automaticky se zapíná TRIM, pokud je tento souborový systém použit na SSD. V plánu jsou i další vlastnosti včetně šifrování.

BtrFS není žurnálovací, ale podobně jako ZFS používá pro tento účel mechanismus CoW (Copy on Write): pokud jsou přepisována data na disku, jejich původní verze existuje až do chvíle, kdy je nová verze dokončena, aby byla zachována konzistentnost dat.

Jak bylo výše uvedeno, BtrFS vychází ze ZFS. Nicméně licence ZFS není kompatibilní s GNU GPL, tedy není možné implementovat podporu ZFS přímo do jádra Linuxu – zřejmě i proto se postupně zvyšuje popularita BtrFS, třebaže je tento systém ještě pořád ve vývoji.

SquashFS. Jde o read-only souborový systém nativně nabízející komprimaci obsahu. Dokáže velmi účinně komprimovat jak data, tak i metadata při zachování slušné přístupové doby. Další vlastnosti jsou celkem podobné souborovým systémům ext, včetně podpory pevných odkazů, používání i-uzlů, atributů, některých časových razítek, ale například nepodporuje POSIX ACL.

Používá se především pro Live distribuce na výměnných médiích. Protože USB flash disky, SD karty nebo optická média bývají obvykle pomalejší než běžný pevný disk či dokonce SSD, hodí se velmi nízká režie.

8.6.4 Srovnání některých 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	ext4fs	ReiserFS	XFS
Max. velikost oddílu	4 TB	4 TB	1 EB	16 TB *)	18∗210 PB
Velikost bloku	1–4 KB	1–4 KB	4 KB	až 64 KB	512 B – 64 KB
Max. velikost souboru	2 GB	2 GB	16 TB	až 210 PB *)	9∗210 PB

*) Záleží na verzi souborového systému.

Tabulka 8.7: Srovnání vlastností souborových systémů pro Linux

Udané hodnoty je však nutné brát s rezervou, na tom, jak velké soubory může souborový systém ukládat, záleží také na VFS.

8.6.5 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).

Následuje stručný výčet některých virtuálních souborových systémů, se všemi jsme již obeznámeni ze cvičení.

procfs zpřístupňuje běhové informace o systému a procesech (používá se v Linuxu). Do některých souborů se dá i zapisovat, čímž měníme chování systému za běhu. 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).

sysfs je založen na podobném principu jako procfs, slouží ke zpřístupnění údajů o zařízeních (v Linuxu). Data zpřístupňuje v adresáři `/sys`.

devfs a udev jsou virtuální souborové systémy spravující speciální soubory zařízení uložené v adresáři `/dev`. V novějších verzích jádra Linuxu modul *udev* kromě toho spravuje souborový systém

sysfs a obecně zařízení, ovladače. V MacOS se dosud používá statický *devfs*.

✎ **ramfs, tmpfs:** souborový systém *ramfs* se používá pro implementaci RAMdisku (tj. část operační paměti se bude používat jako diskový oddíl; výhodou je velká rychlost, nevýhodou je, že se obsah po vypnutí či restartu nezachová. Souborový systém *tmpfs* je jedna z možností použití ramfs, jen pro konkrétní účel – v operační paměti vytváří simulovaný diskový oddíl pro dočasná data (výhodou je, že u dočasných dat rozhodně nevadí, když se po vypnutí nebo restartu systému ztratí), přičemž staví na souborovém systému *ramfs* (je to pro něj prostředek).

✎ **Další:** Nejdůležitější virtuální souborový systém už známe, je to VFS. Je to 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.

8.6.6 Výměnná optická média

✎ Na USB flash discích a SD kartách se používá buď FAT32 nebo ext2, můžeme se také setkat se souborovým systémem exFAT. Souborový systém NTFS není pro tato média vhodný, protože zapisuje mnoho metadat a tím zbytečně snižuje životnost média. Používá se jen tehdy, když jsme k tomu nuceni (například musíme ukládat velmi velké soubory, na které FAT32 nestačí – i když je otázkou, jestli by pak nebyl vhodnější systém exFAT).

✎ **CDFS** je souborový systém pro média CD. Maximální velikost souboru je 4 GB, maximální počet adresářů je 65 535. Jiný název je ISO 9660.

✎ **UDF (Universal Disk Format)** je určen pro DVD a Blu-Ray, ale také CD. Podporuje dlouhé názvy adresářů a souborů, také v UNICODE, soubory mohou být i řídké. Ne všechny operační systémy obsahují ovladač s podporou všech vlastností UDF (podpora zápisu, pojmenované proudy, ACL, apod.).

Literatura

Základní:

[1] Dráb, M. *Jádro systému Windows: Kompletní průvodce programátora.* Brno, Computer Press, 2011.

[2] Jelínek, L. *Jádro systému Linux: Kompletní průvodce programátora.* Brno, Computer Press, 2008.

[3] Russinovich, M. E. – Solomon, D. A. *Vnitřní architektura Microsoft Windows.* Z anglického originálu Windows Internals. Brno, Computer Press, 2007.

Další:

[4] Aitken, P. G. *Windows Script Host 2.0.* Praha, Grada Publishing, 2001.

[5] Allen, R. – Lowe-Norris, A. G. *Active Directory.* Praha, Grada Publishing, 2005.

[6] Bělka, J. *Začínáme bezpečně s FreeBSD* [online]. Root.cz.
Dostupné na: http://www.root.cz/serialy/zaciname-bezpecne-s-freebsd/

[7] Bernáthová, A. *Linuxové souborové systémy* [online]. Linux Express.
Dostupné na: http://www.linuxexpres.cz/praxe/linuxove-souborove-systemy

[8] Born, G. *Skriptujeme operace na PC pomocí Microsoft Windows Script Host 2.0.* Brno, Computer Press, 2001.

[9] Caletka, O. *Partition Magic, Symantec Ghost a další utility pro práci s pevným diskem.* Brno, Computer Press, 2002.

[10] Čada, O. *Mac OS X Shell krok za krokem.* Praha, Grafika Publishing.

[11] Čada, O. *Operační systémy.* Praha, Grada, 1993.

[12] Dvořák, V. *WIN95 + RH6.2 = 10GB HDD* [online]. Linuxové noviny, 2001.
Dostupné na: http://www.linux.cz/noviny/2001-08/clanek05.html

[13] Horák, J. *BIOS a Setup.* Brno, Computer Press, 2004.

[14] Kačmář, D. *Programujeme v COM a COM+.* Brno, Computer Press, 2000.

[15] Kadlec, Z. *Průvodce nitrem BIOSu.* Praha, Grada Publishing, 1996.

[16] KOKOREVA, O. *Registr Microsoft Windows XP*. Brno, Computer Press, 2002.

[17] Kolektiv autorů. *Linux Dokumentační projekt* [online]. 3. aktualizované vydání. Brno, Computer Press, 2003. Soubory ke stažení na adrese http://knihy.cpress.cz/DataFiles/Book/00000675/Download/K0819.pdf

[18] Kolektiv autorů. *The Linux Documentation Project* [online]. Poslední aktualizace 2006. Dostupné na: http://www.tldp.org

[19] KRAVAL, I. – IVACHIV, P. *Základy komponentní technologie COM*. Brno, Computer Press, 2001.

[20] KWOLEK, J. *Problematika IRQ – sdílení, konflikty, PCI* [online]. Živě.cz. Dostupné na: http://www.zive.cz/clanky/problematika-irq—sdileni-konflikty-pci/irq—what-the-hell/sc-3-a-1399-ch-16552/default.aspx

[21] LASSER, J. *Rozumíme UNIXu*. Praha, Computer Press, 2002.

[22] LEE, J. – WARE, B. *OpenSource – vývoj webových aplikací*. Brno, Computer Press, 2000.

[23] LUCAS, M. *FreeBSD*. Brno, Computer Press, 2003.

[24] MACHEJ, P. *GRUB – zavaděč systému*. Časopis Linux+ 2/05, str. 52–57.

[25] Maturita.cz. *Konfigurace počítače* [online]. Dostupné na: http://www.maturita.cz/prv/konfigurace_pocitace.htm

[26] MICHL, V. *Linux a vlákna* [online]. Linuxové noviny 08-09/98. Dostupné na: http://www.linux.cz/noviny/1998-0809/clanek11.html

[27] Microsoft Corporation. *Microsoft Windows XP Professional Resource Kit*. Brno, Computer Press, 2004.

[28] MOSKOWITZ, J. *Zásady skupiny, profily a IntelliMirror ve Windows 2003, 2000 a XP*. Brno, Computer Press, 2006.

[29] MRÁZ, O. *Autoexec.bat a Config.sys* [online]. Dostupné na: http://www.volny.cz/otakarmraz/swPomoc/autocnfg.html

[30] PARK, E. B. – GALÍČEK, R. *Dual-Boot Linux a Windows 2000/XP s Grub HOWTO* [online]. Dostupné na: http://www.volny.cz/galicek/linux%20ver%20XP/grub-w2k-HOWTO-cz.html

[31] PATOČKA, M. *Porovnání systémů Linux a FreeBSD* [online]. Root.cz. Dostupné na: http://www.root.cz/serialy/porovnani-systemu-linux-a-freebsd/

[32] PLÁŠIL, F. *Operační systémy*. Praha, ČVUT, 1983.

[33] PLÁŠIL, F. – STAUDEK, J.: *Operační systémy*. Praha, SNTL, 1991.

[34] POKORNÝ, J. *Úvod do .NET Framework*. Brno, Computer Press, 2000. Soubory ke stažení na knihy.cpress.cz/DataFiles/Book/00000896/Download/frameworknet.zip

[35] PRICE, B. *Active Directory*. Brno, Computer Press, 2005.

[36] Raymond, E. S. *Umění programování v UNIXu.* Brno, Computer Press, 2004.

[37] Řehák, M. *Operační systémy.*
Dostupné na: http://www.volny.cz/rayer/os/os.htm

[38] Shulyupin, Constantine. Linux Technology Reference [online]. *MakeLinux.Net.*
Dostupné na: http://www.makelinux.net/reference

[39] Solomon, D. A. *Windows NT pro administrátory a vývojáře.* Brno, Computer Press, 1999.

[40] Stanek, W. R. *Příkazový řádek Microsoft Windows.* Brno, Computer Press, 2005.

[41] Toxen, B. *Bezpečnost v Linuxu.* Brno, Computer Press, 2003.

[42] Vondra, T. *Gentoo Handbook, konfigurace zavaděče* [online].
Dostupné na: http://www.fuzzy.cz/gentoo/files/html/ch09.html

[43] Walnum, C. *Programujeme grafiku v Microsoft Direct3D.* Brno, Computer Press, 2004.

[44] Walnum, C. *VMWare.* Brno, Computer Press, 2004.