Principy operačních systémů

KI/POS

Jiří Fišer

Ústí nad Labem 2020

Kurz: Principy operačních systémů

Obor: Aplikovaná informatika

Klíčová slova: operační systém, jádro OS, virtualizace paměti, multitasking, meziprocesová komunikace, synchronizace, souborový systém

Anotace: Kurs je zaměřen na základní principy a interní strukturu současných operačních systémů. Pozornost je také věnována tomu, jak se tato struktura projevuje v aplikačním rozhraní jednotlivých systémů (Win32, POSIX) a tedy i plnohodnotnému využití možností současných operačních systémů v uživatelských programech.

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Katedra informatiky, PřF, UJEP v Ústí nad Labem, 2016

Autor: Jiří Fišer

Obsah

Úvodní slovo **5**

1. Operační systém **6**
- 1.1. Co je „operační systém“? 6
  - 1.1.1. funkční pohled 7
  - 1.1.2. systémový pohled 10
- 1.2. Architektura operačního systému 11
  - 1.2.1. monolitické operační systémy 12
  - 1.2.2. hierarchické operační systémy 13
  - 1.2.3. operační systémy typu klient-server 14

2. Správa operační paměti **18**
- 2.1. Funkce správy paměti 18
- 2.2. Metody správy (fyzického) adresového prostoru 19
  - 2.2.1. monolitická aplikační paměť 19
  - 2.2.2. statické bloky 21
  - 2.2.3. dynamické bloky 22
  - 2.2.4. setřásání bloků 24
- 2.3. Virtualizace paměti 24
  - 2.3.1. stránkování 26
  - 2.3.2. logický adresový prostor procesu 29
  - 2.3.3. stránkování na žádost 31
  - 2.3.4. jak si ukrást stránku 33
  - 2.3.5. strategie kradení stránek 35
  - 2.3.6. sdílená paměť 38

3. Správa procesů **47**
- 3.1. Proces a jeho kontext 47
- 3.2. Multitasking 50
  - 3.2.1. vzájemné volání procesů 50
  - 3.2.2. kooperativní multitasking 51
  - 3.2.3. preemptivní multitasking 52
- 3.3. Stavový diagram procesů 54
  - 3.3.1. nový proces (stav NEW) 54
  - 3.3.2. čekající proces (stav WAITING) 55
  - 3.3.3. běžící proces (stav RUNNING) 57
  - 3.3.4. zablokovaný proces (stav SLEEPING) 57
  - 3.3.5. proces mátoha (stav ZOMBIE) 58
- 3.4. Vlákna (thready) 59

4. Synchronizace a meziprocesová komunikace 62
4.1. Kritický kód a vzájemné vyloučení 62
4.2. Synchronizační prostředky 66
4.2.1. binární semafor 66
4.2.2. mutex 69
4.2.3. událost (event) 70
4.2.4. obecný semafor 72
4.3. Uváznutí 72
4.3.1. eliminace uváznutí 75
4.4. Základní komunikační prostředky 77
4.4.1. klasifikace komunikačních prostředků: 77
4.4.2. roura 78
4.4.3. soket (schránka) 79
4.4.4. fronta zpráv 80

5. Souborový systém 82
5.1. Souborový systém — úložiště persistentních dat 82
5.2. Svazek 83
5.3. Vyrovnávací paměti 84
5.4. Vnitřní representace souborů 86
5.5. Paměťové i-uzly a VFS 88
5.6. Adresářová struktura 90
5.6.1. Odkazy (links) 92
5.6.2. Připojování svazků (mounting) 93
5.7. Otevřený soubor 94

6. Subsystém vstupu a výstupu 97
6.1. Bloková a znaková zařízení 97
6.2. Ovladače 98
6.2.1. Rozhraní ovladačů 98
6.2.2. Identifikace zařízení a ovladačů 99
6.2.3. Horní a dolní polovina ovladače 101

7. Bezpečnost na úrovni OS 104
7.1. Co je bezpečnost 104
7.2. Kvantifikace bezpečnosti 106
7.2.1. Trusted Computer System Evaluation Criteria (TCSEC) 107
7.2.2. Common Criteria 108
7.3. Kryptografie a PKI 109
7.3.1. Symetrická a asymetrická kryptografie 110
7.3.2. Certifikační autorita 113
7.3.3. Vytvoření bezpečného kanálu 117

Literatura 121

Úvodní slovo

Vstupní znalosti

Kurs *Principy operačních systémů* přímo navazuje na kurs *Operační systémy* (povinná závislost). V rámci navazujícího kursu se předpokládají znalosti základních pojmů (např. proces, objekty souborového systému) a znalost základních unixovských nástrojů (zobrazení stavu procesů, zobrazení souborového systému).

Výhodou je také znalost základů OOP jazyka C# nebo Java, neboť některé algoritmy a API rozhraní jsou diskutována v tomto jazyce.

Výstupní znalosti

Po absolvování kursu se budete orientovat v základních principech návrhu operačních systémů.

Pro aplikačního programátora je hlavní výhodou znalost principů rozhraní moderních OS včetně jejich postranních efektů. V rámci kursu není popsáno žádné konkrétní API, znalosti v oblasti návrhu OS však výrazně zjednodušují jeho percepci.

Pro systémového programátora je hlavním přínosem popis hlavních principů, algoritmů a objektů v rámci jádra. Absolventi, kteří by se chtěli orientovat tímto směrem musí prostudovat i doporučenou literaturu, a/resp. pokračovat ve studiu na magisterském stupni.

Na kurs nenavazuje žádný kurs bakalářského studia. Předpokládá se, že znalosti z tohoto kursu studenti využijí v dalším studiu a to jak v rámci praxe tak navazujícího studia.

Klíčové pojmy

klíčový pojem

Pojmy uvedené na levém okraji textu (a v textu zvýrazněné tučně) jsou tzv. **klíčové pojmy**. Jejich správné a plné pochopení je nezbytné pro další studium. Je samozřejmou součástí zkoušek (a to i zkoušek navazujících předmětů resp. státní závěrečné) a jejich znalost se předpokládá i v rámci obhajoby seminární práce.

Většinu z nich můžete najít v doporučené literatuře a jsou popsány i v anglické *Wikipedii* (anglický překlad je uveden, s výjimkou termínů, kde je zřejmý). V oblasti informatiky jsou články anglické Wikipedii (ve většině případů) velmi kvalitní, s rozsahem přesahujícím popis uvedený v opoře a tak mohou být využity k dalšímu zpřesnění prohloubení znalostí. Navíc obsahují odkazy na další hodnotné zdroje.

1 Operační systém

CÍLE KAPITOLY

Úvodní kapitola poskytuje dvě alternativní definice operačního systému a zavádí klíčové pojmy. Pochopení pojmu operační systém usnadňuje i popis základních typů architektur operačních systémů.

PŘEČTĚTE SI

Pokud nevíte, jak probíhá vykonávaní instrukcí a obsluha přerušení prostudujte z knihy *Operating Systems Internals and Design Principles* (Stallings) [12] následující kapitoly:

1.3. Instruction Execution

1.4. Interrupts

1.1 Co je „operační systém“?

Na tuto otázku, jež je pro tato skripta zcela zásadní, bohužel neexistuje jednoduchá a přitom zcela přesná odpověď. Nepopíratelná je pouze příslušnost operačních systémů k *programovému vybavení digitálních počítačů* (= ***softwaru***). Jinak řečeno, každý operační systém je programem *univerzálního počítače* (chápaného však velmi obecně — např. celá síť může být u distribuovaných operačních systémů chápána jako jediný počítač, naopak na jednom fyzickém počítači může být programovými prostředky vytvořeno několik virtuálních počítačů), tj. souborem *rutin* strojového kódu daného počítače (*rutina* je funkčně vymezená část kódu, nejčastěji ve formě podprogramu vyššího programovacího jazyka).

S velmi vysokou jistotou lze dále konstatovat, že operační systémy tvoří vlastní podmnožinu *softwaru*, neboť jinak by se jednalo o zcela nadbytečný termín (což by ale na druhou stranu nebyla žádná zvláštní výjimka, některé obory lidské činnosti se bez nadbytečných termínů neobejdou). Pokud však budeme ověřovat všechny rutiny schopné běhu na určitém počítači lze jen u některých s jistotou tvrdit, že patří resp. nepatří do *operačního systému*. Naštěstí i přes určitou mlhavost lze najít kritéria, která lze pro vymezení *operačního systému* (v konkrétním i obecném případě) použít. Mezi hlavní kritéria patří systémový pohled (*operační systém* je systémem) a především pohled funkční (tj. lze definovat obecnou funkci operačního systému), přičemž oba pohledy se doplňují i prolínají.

1.1.1 funkční pohled

Funkční pohled (a tudíž i funkční vymezení) operačního systému vychází z následující definice:

Operační systém je (softwarový) správce prostředků.

Tato definice popisuje operační systém relativně přesně, užívá však dvou dříve nezavedených pojmů — *správce* a *prostředek*. Jejich definici jsou proto věnovány dvě následující podkapitoly.

Prostředky operačního systému

Prostředky spravované operačním systémem lze rozdělit do dvou velkých skupin:

fyzický prostředek

hardwarové (fyzické) prostředky mohou tvořit všechny hardwarové komponenty počítače včetně periferií. Z pohledu operačního systému jsou nejdůležitější tyto skupiny fyzických prostředků (výčet není z pochopitelných důvodů úplný):

- **centrální prostředky**: procesor, operační paměť (tyto prostředky jsou bezpodmínečně nutné pro samu existenci OS)
- **vnější paměti**: pevný disk, CDROM, disketa apod.
- **vstupně výstupní prostředky**: terminál, tiskárna, myš, hardware sítě

logický prostředek

softwarové (logické) prostředky jsou vytvářeny rutinami operačního systému za využití jiných (nízkoúrovňovějších) prostředků, a to jak fyzických tak logických (rutiny přidávají určitou funkčnost nebo odchylné vlastnosti). Logické prostředky tvoří rozsáhlý hierarchický systém, který v několika vrstvách obklopuje fyzické prostředky počítače, čímž mimo jiné brání rutinám stojícím mimo operační systém přistupovat přímo k fyzickým prostředkům spravovaným daným operačním systémem (u jednodušších OS však mohou existovat fyzické prostředky stojící mimo správu OS).
Úplná klasifikace logických prostředků je při jejich různorodosti velmi obtížná (ne-li nemožná), mezi nejdůležitější typy softwarových prostředků současných OS však patří:

- **centrální prostředky**: proces, paměťový region (nejdůležitější logické prostředky vybudované přímo nad centrálními prostředky fyzickými)
- **nízkoúrovňové logické prostředky**: logický disk, logický terminál (displej), logické znakové zařízení (vytvořeny přímo nad fyzickými prostředky)
- **prostředky dílčí**: soubor, GUI okno (prostředky zapouzdřující dílčí část nízkoúrovňového prostředku)
- **virtuální prostředky**: virtuální paměťový nebo síťový disk, virtuální tiskárna (logické prostředky napodobující funkčnost fyzických prostředků, jež nemusí být fyzicky přítomny)
- **prostředky synchronizační a komunikační**: roura, semafor apod.
- **bezpečnostní prostředky**: účet, šifrovaný komunikační kanál
- **síťové prostředky**: TCP/IP sokety, HTTP server a klient, elektronická pošta

- **multimediální prostředky**: písmový stroj, přehrávač multimediálních souborů, zobrazovač bitmapové grafiky
- **programovací prostředky**: překladače a interpretry operačních systémů

Speciální pozornost si zaslouží především dva centrální softwarové prostředky, neboť bez jejich alespoň částečného pochopení nelze získat reálnou základní představu o operačním systému.

***Proces** je instancí programu (aplikace).*

Každý program (někdy se používá i pojem *aplikac*e, jenž si však vyhrazuji pro označení programů, které nejsou přímou součástí operačního systému) může být vykonán (tj. počítač může zpracovávat jednotlivé instrukce programu) a to v dané instanci operačního systému vykonán více než jednou (na druhou stranu nemusí být přirozeně vykonán ani jednou). Každé toto vykonání od okamžiku provedení první instrukce programu po provedení instrukce poslední, včetně případných volání rutin operačního systému označujeme termínem **proces** (operačního systému). Rutiny operačního systému vytvářejí proces jako identifikovatelnou entitu (proces je označen jednoznačným identifikátorem), poskytují mu prostředky nutné pro jeho běh (primárně procesor a paměť) a postarají se i jeho zánik po skončení běhu. Detailnější pohled na proces přináší kapitola 3, ale již nyní je vhodné uvést několik poznatků o procesech:

proces

- každá rutina výpočetního systému (resp. softwaru) může být vykonávána pouze v rámci procesu, může však být vykonána i několika procesy (a to dokonce současně)
- ve výpočetním systému musí v každém okamžiku existovat alespoň jeden proces
- operační systém (jako program či množina rutin) není vykonáván jako jediný proces, ale jeho jednotlivé rutiny jsou sdíleny všemi procesy výpočetního systému (tj. jsou vykonávány v rámci běžných aplikačních procesů), pouze menší část je součástí specializovaných tzv. *systémových procesů*
- procesy vznikají jako reakce na požadavek jiného procesu (rodičovského procesu). Jedinou výjimkou je proces vznikající při startu (bootování) instance operačního systému.

paměťový region

Paměťový region je souvisle adresovatelná oblast paměti, přidělená procesu (či výjimečně u tzv. sdílené paměti několika spolupracujícím procesům). Důležitým požadavkem je přímý přístup k této paměti, spočívající v možnosti jejího přímého čtení, či modifikaci, což umožňuje využít paměťový region (resp. jeho fyzickou realizaci) pro ukládání strojového kódu procesu (programu) nebo jeho stavů (mezivýsledků, návratových adres podprogramů apod.). *Paměťový region* je u nejjednodušších systémů realizován nad souvislou oblastí fyzické operační paměti, u složitějších pak nad souvislou oblastí *virtuální paměti*, jejíž fyzickou realizací je nesouvislá množina malých bloků operační paměti, *odkládacího (swapovacího) prostoru* resp. běžných souborů na diskovém zařízení, či dokonce bez skutečné fyzické realizace (nebyla-li tato paměť nikdy využita).

OS jako správce

Operační systém si však nelze představit pouze jako množinu prostředků a správa prostředků nespočívá pouze v jejich vytváření (u logických prostředků), či bezprostředního řízení (u prostředků fyzických). Hlavní účel správy spočívá ve vytváření přesně definovaného a bezpečného prostředí pro procesy (resp. programy), a to ve dvou následujících těsně provázených směrech:

vytvoření tzv. virtuálního počítače

— rutiny operačního systému vytvářejí pro aplikační programy jednotné rozhraní, jež skrývá jemné (ale mnohdy i výrazné) rozdíly na úrovni fyzických zařízení. Toto jednotné rozhraní má charakter virtuálního počítače, v němž roli instrukcí hrají volání přesně definovaných služeb systému, jež jsou identická pro všechny instance a verze operačního systému (nové verze rozhraní rozšiřují, ale zachovávají zpětnou kompatibilitu). Navíc operační systém nabízí oproti fyzickému počítači (hardwaru) rozhraní výrazně vyšší úrovně, tj. pomocí relativně jednoduchých volání lze provést i velmi složité akce (například jedním voláním lze přehrát audio soubor nebo navázat bezpečné spojení se vzdáleným počítačem). Hardwarová nezávislost, stabilita a relativně komfortní programátorské prostředí, to vše přináší operační systém a to vše výrazně usnadňuje tvorbu aplikačního programového vybavení.

Vytvoření komfortního a přitom zcela unifikovaného prostředí však může ve svém důsledku vést ke ztrátě pružnosti a efektivity celého softwarového systému, neboť některá hardwarová zařízení mohou být pro aplikace buď zcela nepřístupná (operační systém jimi vyžadovaný prostředek neposkytuje ve formě logického zařízení), nebo pro určitou skupinu aplikací nenabízí dostatečně efektivní (rychlý) přístup (vytváření unifikovaného vysokoúrovňového rozhraní snižuje efektivitu). Většina operačních systémů proto nabízí možnost využívání nižších úrovní (ne zcela unifikovatelných) logických zařízení nebo výjimečně i téměř přímý přístup k hardwaru (i ten je však alespoň minimálně řízen operačním systémem).

OZ: Jaké (běžné) typy aplikací vyžadují efektivní a nízkoúrovňový přístup k hardwaru?

nezávislost jednotlivých procesů

— pro každý proces běžící v rámci operačního systému musí rutiny OS vytvářet iluzi, že je jediným procesem, který kdy běžel, běží a bude běžet v rámci dané instance OS.

instance OS

Instance OS vzniká natažením [bootováním] systému a zaniká vypnutím počítače resp. rebootováním. Každý program (a tím i její programátor resp. uživatel) musí mít dojem, že může téměř bez časového či kvantitativního omezení využívat všech fyzických a logických prostředků počítače resp. operačního systému, což výrazně zjednodušuje tvorbu aplikací (aplikační programátor nemusí řešit interakce nebo kolize postupně či dokonce souběžně běžících procesů). Každý proces má tudíž k dispozici vlastní (vyhrazený) virtuální počítač, jenž je zcela nezávislý na zbytku systému (tj. na ostatních procesech a jejich virtuálních počítačích)

OT: Proč je nutné striktně oddělovat prostředky jednotlivých procesů.

Není to však zadarmo, neboť operační systém může této téměř absolutní nezávislosti dosáhnou pouze za cenu tvorby složitých logických (nejčastěji virtuálních) prostředků, jež jsou jediné schopny bránit kolizím procesů například při souběžném využívání fyzických zařízení (představte si například, k čemu by mohlo dojít při souběžném nekoordinovaném přístupu více procesů k vypalovačce CD-ROM). Je zřejmé, že nejsložitější je dosažení této iluze u operačních systémů s možností souběžného běhu více procesů, ať již je tento souběh skutečný (operační systémy na počítačích s více procesory) nebo zdánlivý (systémy s tzv. *preemptivním multitaskingem*), ale na jednodušší úrovni musí být řešeno u všech OS.

Vytváření striktně oddělených virtuálních počítačů je výhodné pro většinu aplikací, pokud však (spíše výjimečně) potřebuje skupina procesů interagovat (tj. vzájemně se ovliv-

ňovat), nabízí většina OS i prostředky pro jejich vzájemnou komunikaci, jež jsou však poskytovány pouze na vyžádání a jsou operačním systémem i kontrolovány.

1.1.2 systémový pohled

Operační systém je tvořen množinou rutin, které je možno na základě vzájemného volání organizovat do několika vrstev. Nejníže leží vrstva rutin, jež přímo přistupují k hardwaru (tj. k fyzickým prostředkům), dále skupina rutin přímo volajících rutiny nejnižší vrstvy, a tak dále až po nejvyšší vrstvu, přičemž zahrnuty mohou být všechny rutiny softwarového systému. Každá vrstva nabízí vyšším vrstvám přesně definované rozhraní (tj. vytváří různé úrovně logických prostředků resp. virtuální počítače na různých úrovních abstrakce).

Zatímco u nejnižší vrstvy je příslušnost rutin k operačnímu systému nezpochybnitelná, klesá u vyšších vrstev míra příslušnosti postupně k nule. Je tedy pouze otázkou dohody, kam hranici operačního systému položit. Nejčastěji se uvažuje o čtyřech vymezeních (bohužel opět trochu mlhavě definovaných):

jádro operačního systému

jádro OS

Do **jádra operačního systému** (anglicky *kernel*) patří ty rutiny, které bezprostředně přistupují k hardwaru počítače, resp. zajišťují virtualizaci nezbytnou k zajištění nezávislosti jednotlivých procesů. Navíc se k jádru operačního systému často počítají i rutiny zajišťující základní logické prostředky (např. prostředky pro komunikaci procesů včetně souborového systému) a vrstvy, jež nabízí přesně definované a přitom přehledné rozhraní pro jednotlivé aplikace (tzv. systémové API). Typickým vnějším znakem jádra operačního systému bylo jeho uložení v jediném souboru (tzv. obraz [*image*] jádra) a jeho neustálá přítomnost v paměti (včetně odkládací paměti [*swapu*]) po celou dobu běhu OS tj. téměř po celou dobu běhu počítače. Současná jádra však již bývají modularizována, tj. rozdělena do funkčních částí, jež jsouce uloženy v oddělených souborech mohou být do paměti zaváděny dle aktuální spotřeby.

úroveň systémových procesů

systémový proces

Některé funkce operačního systému se snadněji zajišťují, pokud jsou vykonávány zvláštním procesem (rutiny jádra většinou netvoří samostatný proces, ale jsou vykonávány jako části všech procesů a to jak systémových tak aplikačních). Základním kritériem pro vymezení **systémových procesů** je závislost jádra na správné funkci těchto procesů (tj. chybná funkce nebo dokonce nepřítomnost systémového procesu vede k tzv. pádu operačního systému, při němž OS není schopen dostát požadavkům aplikací, počítaje v to i úplné nebo téměř úplné zastavení výpočetního systému). Systémové procesy jsou buď řízeny běžnými programy a s jádrem komunikují pouze prostřednictvím standardních systémových volání (tj. stejně jako běžné aplikace), nebo běží pouze na úrovni rutin jádra (tzv. procesy jádra).

minimální uživatelský operační systém

Operační systém skládající se jen z jádra a případných systémových procesů je v praxi nepoužitelný, neboť neobsahuje prostředky pro komunikaci s uživateli operačního systému (a to jak pasívními tak aktivními [správci, programátory]). Proto musí být rozšířen o aplikace zajišťující tuto komunikaci, počínaje tzv. **shellem** (aplikace umožňující v textové či grafické podobě správu procesů resp. souborového systému), přes administrátorské nástroje (včetně např. editoru konfiguračních souborů nebo jejich ekvivalentu) a nástroje pro zobrazení či přehrávání multimediálních dat, až po nástroje programátorské (překladače jazyků). Přesné vymezení minimálního systému však silně závisí na cílové skupině, úzu, či spíše tradici daného operačního systému (např. pro Unix je typické zahrnutí programátorských nástrojů pro jazyk C a centrální postavení textového shellu [resp. shellů], pro operační systémy platformy Macintosh centrální postavení grafického shellu). Skutečně minimální systém však může být omezen na mechanismus spuštění jediné aplikace, je-li zájem uživatelů zaměřen jen a pouze na tuto aplikaci (např. u terminálů pro prodej jízdenek).

shell

Do minimálního uživatelského operačního systému patří i některé tzv. systémové knihovny, z nichž nejdůležitější jsou knihovny, jež umožňují snadné volání systémových služeb v programech různých programovacích jazyků (mnohdy také přidávají další vrstvu komfortnějších služeb včetně služeb mimosystémových). Například v Unixu je to knihovna *libc* (primárně pro jazyk C).

balíčkový operační systém

Jako balíčkový operační systém označuji soubor aplikací (včetně obrazu jádra), jenž zákazník obdrží při nákupu operačního systému. Jinak řečeno je to operační systém z obchodního či obecněji komerčního hlediska. Rozsah balíčkového operačního systému se může měnit od minimálního systému po komplexní distribuce, jež obsahují i všechny základní uživatelské aplikace (typické např. pro distribuce Linuxu). Někdy je vhodnější místo konkrétní verze balíčkového operačního systému uvažovat množinu aplikací společnou pro více verzí či distribucí (tj. aplikace typické či očekávané).

Tato skripta se nezabývají operačními systémy na úrovni balíčků (informace na této úrovni jsou součástí uživatelských příruček jednotlivých přibalených aplikací).

1.2 Architektura operačního systému

Základní architektura operačního systému (= statický a generalizující pohled na strukturu) s jádrem obklopeným dalšími vrstvami systémových aplikací byla popsána v předchozí kapitole. Tato architektura je společná valné většině OS (ne-li všem), avšak pokusíme-li se ji dále zpřesnit, narazíme na zásadní odlišnosti jednotlivých OS, jež jsou dány různými požadavky na ně kladenými i rozmanitými přístupy jejich autorů (tj. kolik operačních systémů, tolik architektur).

Vezmeme-li však za základní klasifikační kritérium úroveň spolupráce (obecněji interakce) jednotlivých *rutin* OS, lze rozlišovat systémy s těsným propojením rutin (označované nejčastěji jako *monolitické*), systémy u nichž je spolupráce rutin omezena hierarchickým principem (označované jako *hierarchické nebo vrstevnaté*) a nakonec plně distribuované

systémy, u nichž jednotlivé rutiny netvoří jediný celek, ale jsou soustřeďovány do modulů s pevně daným rozhraním a komunikací prostřednictvím zpráv (označované často termínem *klient-server*). Je nutné si uvědomit, že tyto kategorie architektur nejsou ostře oddělené, ale ve skutečnosti tvoří postupný přechod, u něhož *monolitické* a *distribuované* systémy tvoří extrémy a *hierarchické* pak střední stav (tj. systém může ležet kdekoliv mezi striktně *monolitickou* a striktně *distribuovanou* architekturou, i když oba extrémy jsou prakticky nedosažitelné).

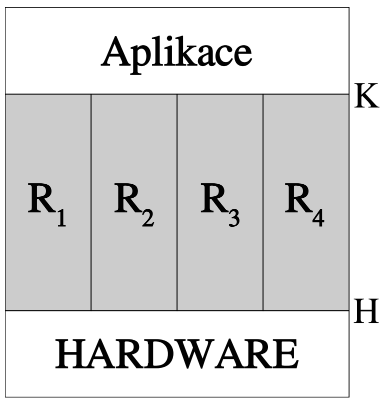
1.2.1 monolitické operační systémy

monolitická architektura

Jádra *monolitických systémů* (z řeckého `mon'o-lijos` = z *jediného* [kusu] *kamene*) jsou tvořena souborem pouze minimálně spolupracujících rutin (či lépe hierarchie rutin), které ve své spodní části spolupracují přímo s hardwarem, ve své části horní pak poskytují rozhraní jádra (tj. jsou přímo volány aplikačním kódem).

Monolitická architektura je možná pouze u těch nejjednodušších operačních systémů (a i zde ne zcela striktní), v omezené míře však lze stopy monolitické architektury nalézt u všech systémů, povětšinou z historických důvodů (i velmi složité systémy vznikly postupným vývojem z jednodušších), avšak vývoj vede k jejich úplné eliminaci (tj. nahrazení *hierarchickou* či dokonce *klient-server* architekturou).

Hlavním důvodem je skutečnost, že (relativní) výhody monolitické architektury, tj. jednoduchost (především ve fázi předběžné analýzy a modelování rutin jádra) a efektivita (kód rutin může být optimalizován pro daný hardware i požadavky aplikací), se zcela ztrácí u rozsáhlejších, a tudíž týmově a inkrementálně vyvíjených operačních systémů, neboť monolitický kód znesnadňuje týmovou spolupráci a snižuje znovupoužitelnost kódu (monolitický kód lze jen obtížně modifikovat).



Jádro operačního systému (označené tmavší šedou barvou) je tvořeno souborem paralelních rutin (zde R_1 až R_4). Jádro je omezeno dvěma ostrými rozhraními — H je rozhraní vůči hardwaru (definované například stavovými a řídícími registry elektronických obvodů), K je tzv. **rozhraní jádra**, jenž se navenek projevuje jako množina přípustných volání rutin operačního systému (tzv. služby jádra či systémové služby) či na vyšším stupni abstrakce jako dostupné rozhraní logických systémových prostředků. Rozhraní jádra využívají jak systémové procesy (pokud existují), tak běžné aplikace. Rozhraní jádra je často provázeno i změnou *režimu privilegovanosti procesoru*, neboť jedině tak lze jádro (a tím i celý OS) ochránit před případným destruktivním chováním aplikací.

rozhraní jádra

Pokud proces vykonává kód aplikace, běží v tzv. *uživatelském* či neprivilegovaném *režimu*. V tomto režimu může přistupovat pouze k části operační paměti a nesmí provést

uživatelský režim

tzv. privilegované instrukce (zastavení běhu procesoru, změna privilegovaných [řídících] registrů apod.). V **režimu jádra** či tzv. privilegovaném režimu vykonává pouze rutiny jádra a není nikterak omezen (může přistupovat k celé fyzické paměti i ke všem ostatním hardwarovým zařízením). Proces mění úroveň privilegovanosti pouze při vzniku *přerušení* (z neprivilegovaného na privilegovaný) a při návratu z něho (v opačném směru). *Přerušení* je vyvoláno buď asynchronním signálem od nějakého zařízení (např. při stisku klávesy, příchodu dat na sériový port, či při dokončení zápisu na disk) nebo programově použitím speciální instrukce (pomocí programového přerušení jsou volány služby operačního systému). Některé procesory volají přerušení také při některých speciálních (většinou chybových) stavech, tato přerušení se označují jako *procesorové výjimky*.

režim jádra

Příkladem operačního systému s monolitickým charakterem je (resp. spíše byl) MS DOS.

1.2.2 hierarchické operační systémy

hierarchická architektura

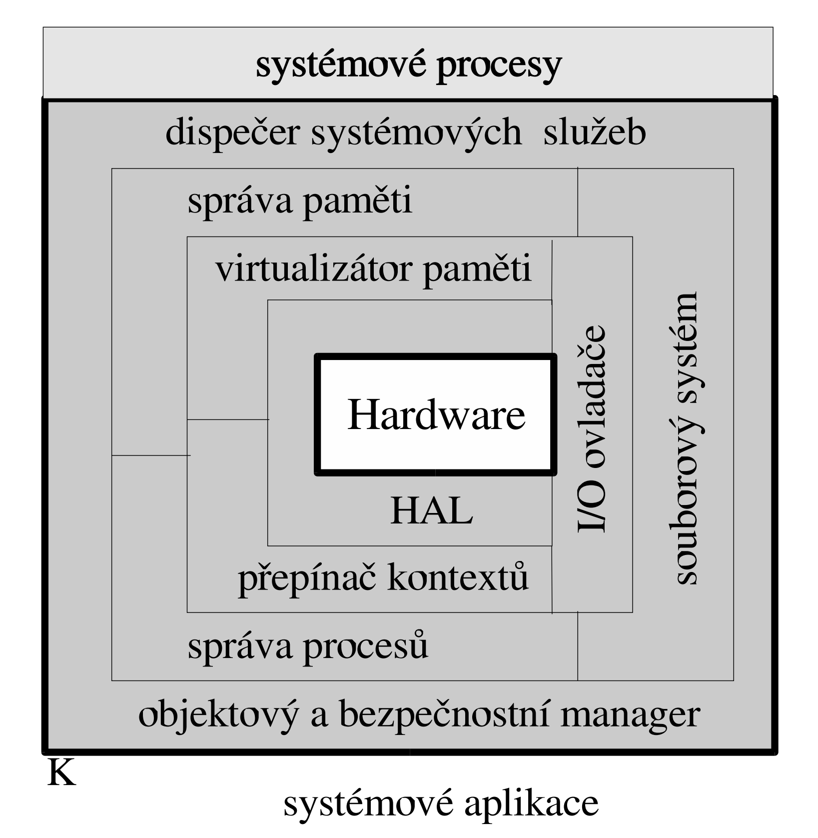
Hierarchická architektura (z řeckého <ier-arqeia , tj. svatá vláda) je pro operační systémy typická a lze dokonce říci, že neexistuje operační systém, jež by neobsahoval rysy hierarchické architektury. Nejtypičtějším příkladem hierarchické architektury je Unix (především ve starších verzích).

V hierarchickém systému jsou rutiny OS uspořádány do vrstev, jež postupně obalují hardware a nabízejí vyšším vrstvám pevně definovaná rozhraní s postupně rostoucí úrovní abstrakce. Mezi nejdůležitější (ale již ne jediná) rozhraní patří také rozhraní hardwaru a rozhraní jádra.

Čistě hierarchický (vrstevný) systém vyžaduje, aby rutiny každé vrstvy přímo volaly pouze veřejně definované rozhraní vrstvy bezprostředně nižší (nebo rutiny stejné vrstvy). Tento požadavek nelze u operačních systémů zcela dodržet, a tak se lze (spíše výjimečně) setkat i s přímým voláním rozhraní hlouběji zanořených vrstev (tj. některé vrstvy jsou přeskočeny) nebo dokonce (skutečně výjimečně) s voláním vrstev na vyšší úrovni abstrakce (nejčastěji při virtualizaci zařízení).

Kromě vertikálního členění je pro hierarchické operační systémy typické i členění horizontální, jehož výsledkem jsou funkčně vymezené subsystémy, jež sahají přes několik vertikálních vrstev. Nejčastěji se na úrovni jádra vyčleňují tyto subsystémy:

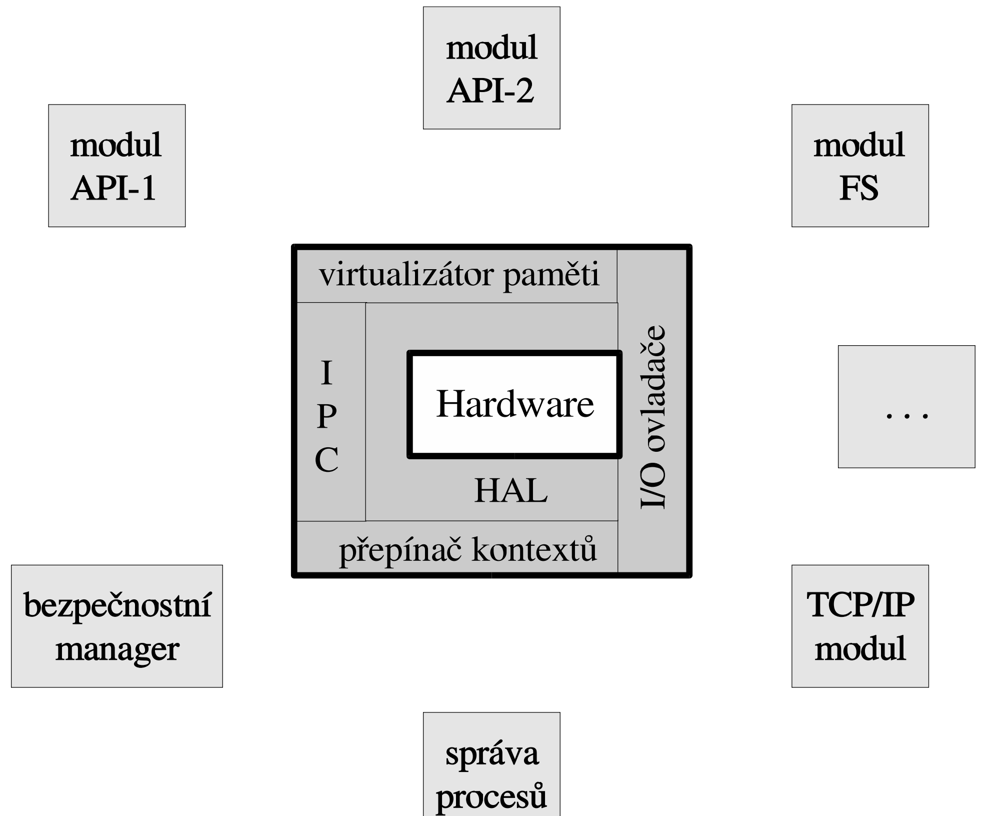
- subsystém správy procesů
- subsystém správy paměti
- souborový systém
- síťový a komunikační subsystém (někdy je zařazován pod vstupně výstupní subsystém)
- vstupně výstupní subsystém



1.2.3 operační systémy typu klient-server

klient-server architektura

Architektura klient-server přináší maximálně distribuovaný přístup k návrhu a implementaci operačních systému, což usnadňuje týmový a inkrementální návrh a implementaci rozsáhlých systémů, bohužel však poněkud snižuje výkonnost systému (systémy s čistě klient-server architekturou jsou relativně pomalé a tudíž jen omezeně konkurenceschopné).



Hlavní změnou oproti klasickému hierarchickému modelu je vyčlenění všech zbytných rutin z jádra do specializovaných systémových procesů — *serverů*. V jádře zůstávají pouze zcela nezbytné rutiny pro virtualizaci paměti, přepínání a komunikaci procesů, resp. pro přístup k dalším hardwarovým prostředkům. Výsledné redukované jádro (označované jako **mikrojádro**) nabízí ostatním procesům (systémovým i aplikačním) pouze základní funkce (tj. centrální logické prostředky), vše ostatní zajišťují servery prostřednictvím *meziprocesorové komunikace* (rychlá a efektivní meziprocesorová komunikace zajišťovaná mikrojádrem je klíčovým mechanismem architektury klient-server).

mikrojádro

Počet serverů je formálně neomezený (dokonce může existovat i více serverů pro jednu službu), mezi ty nejdůležitější patří například servery souborových systémů (pro každý podporovaný formát souborového systému existuje specializovaný server), servery bezpečnostní (zajištění autentifikace, důvěrnosti dat apod.), a servery nabízející API na úrovni rozhraní jádra (může existovat více serverů emulujících API různých operačních systémů). API servery jsou z hlediska aplikačního programátora nejdůležitější, neboť jen s nimi komunikuje při volání služeb operačního systému (např. při otevření souboru, posunu GUI okna či ukončení procesu). Toto volání, i když se z pohledu aplikačního programátora prakticky neliší od volání služby klasického (hierarchického) systému, používá zcela odlišný vnitřní mechanismus.

Například pokud uživatelská aplikace zavolá systémovou službu pro otevření souboru (to se děje voláním funkce z uživatelsky přístupné knihovny systémových služeb), je místo přímé komunikace s API serverem (procesy jsou odděleny a jejich přímá komunikace je nemožná) zavolána služba mikrojádra pro meziprocesorovou komunikaci a procesu zajišťujícímu API server je poslána zpráva, jejíž formát je učen dohodou. Mikrojádro zajistí předání zprávy serveru (viz popis mechanismu zpráv v kapitole 4.4.4), jenž zprávu přečte a v reakci na ni zajistí otevření souboru, a to ve spolupráci se serverem souborového systému a serverem bezpečnostním (ne všichni mají právo soubor otevřít), což si vyžádá předání několika dalších zpráv (žádostí a odpovědí). Po splnění (resp. dosažení

chybového stavu) pošle API server původní aplikaci zprávu s identifikátorem (handlerem) nově otevřeného souboru (resp. chybovým kódem). Původní aplikace pak může dokončit volání služby návratem do aplikace.

PŘEČTĚTE SI

Alternativní pohled na základní charakteristiku operačních systémů nabízí druhá kapitola knihy *Operating Systems Internals and Design Principles* (Stallings) [12] a to především tyto podkapitoly:

2.1 Operating System Objectives and Functions

2.2 The Evolution of Operating Systems

2.3 Major Achievements

OTÁZKY

1. Jaký je rozdíl mezi procesem a aplikací?
2. Proč je nutné striktně oddělovat prostředky jednotlivých procesů.
3. Co nemůže proces přímo provádět v uživatelském režimu?
4. Jak lze překonat rozhraní jádra
5. Jaká je hlavní nevýhoda architektury klient-server?

OTÁZKY K ZAMYŠLENÍ

1. Promyslete rozpor mezi požadavky úplné virtualizace prostředků a maximální efektivity u počítačových her?
2. Patří webový prohlížeč do minimálního uživatelského OS? Argumentujte pro i proti. Jaké to má komerční konsekvence?
3. Jaké jsou výhody modularizace jader operačních systémů?

2 Správa operační paměti

CÍLE KAPITOLY

Hlavním cílem této kapitoly je hodnocení některých metod správy operační paměti (rozšířené o odkládací) prostor, ze tří základních hledisek:

efektivita — přesněji řečeno dodatečné režie při provádění přístupu k paměti

podpora multitaskingu — přesněji řečeno rozsahu závislostí, které vnucuje procesům, které by měly být zcela nezávislé

bezpečnost — jak chrání paměť před neoprávněnými přístupy aplikací

Hlavní pozornost je věnována mechanismu tzv. virtualizace paměti. Programátor musí znát nejen API jádra pro správu paměti, ale i důsledky jednotlivých volání, včetně principů implementace v hardwaru.

I když se například může zdát směr procházení matic (po řádcích nebo po sloupcích na první úrovni) jako bezvýznamný detail, může výrazně ovlivnit výkonnost systému.

Dalším cílem je pochopení významu řízeného sdílení paměti včetně lenivého přístupu ke kopírování (i lenivost může být ctností).

PŘEČTĚTE SI

Pokud neznáte organizaci paměti v moderních počítačích prostudujte z knihy *Operating Systems Internals and Design Principles* (Stallings) [12] následující kapitoly:

1.5 The Memory Hierarchy

1.6 Cache Memory

1.7 Direct Memory Access

2.1 Funkce správy paměti

Operační paměť, tj. paměť přímo přístupná procesoru, je zcela nezbytným fyzickým prostředkem a její správa jednu z hlavních částí operačního systému. Operační paměť musí uchovávat kód programů-procesů, mezivýsledky jejich činnosti, ale i stav ostatních prostředků a základní datové struktury jádra.

Z hlediska správy paměti lze operační paměť pro zjednodušení chápat jako souvislý prostor paměťových buněk o velikosti jednoho bytu, jež jsou lineárně adresovány adresami pevné délky. Jedná se opravdu o zjednodušení, neboť paměť nemusí být souvislá, ani

organizována po bytech a adresa nemusí být vždy lineární. Pokud je tento adresový prostor přímo representován fyzickou pamětí s přímým přístupem (tj. dnes některým z typů paměti RAM), označujeme jej jako **fyzický adresový prostor** (FAP). Velikost tohoto prostoru je dána buď velikostí fyzické paměti (tj. osazením paměťových modulů), nebo velikostí adresy (adresa o velikosti n bitů může adresovat nejvýše 2^n paměťových míst, tj. zde bytů), podle toho, která hodnota je nižší.

fyzický adresový prostor

U jednodušších procesorů je fyzický adresový prostor jediný možný, většina současných procesorů však nabízí i tzv. virtualizaci paměti, a tedy i prakticky neomezený počet tzv. logických adresových prostorů. Těmto prostorům, jejich tvorbě a správě je věnována kapitola 2.3 na straně 24. Nejdříve však zaměříme pozornost na správu fyzických adresových prostorů, neboť jsou jednodušší a především mnohé jejich prvky jsou užívány i v případě správy založené na virtualizaci paměti.

Správa fyzického adresového prostoru musí plnit následujících pět funkcí:

1. přidělování paměťových regionů na požádání procesů — proces specifikuje požadovanou velikost regionu a je mu (pokud možno) přidělena souvislá oblast adresového prostoru (v případě FAP je mu tak přidělena i odpovídající část operační paměti)
2. uvolňování paměťových regionů na požádání procesů — uvolnění paměti musí vždy skončit s úspěchem
3. udržování informací o obsazení adresového prostoru — správce musí udržovat informace o obsazených i volných oblastech paměťového prostoru
4. zabezpečení ochrany paměti — správce paměti musí, pokud to technické prostředky dovolují, bránit procesům přistupovat k paměti mimo regiony, jež vlastní (tj. k volné paměti, k regionům ostatních procesů a především k paměti vyhrazené jádru).
5. u víceúlohových systémů musí správa paměti podporovat střídavý běh více procesů nebo mu alespoň nesmí bránit

2.2 Metody správy (fyzického) adresového prostoru

2.2.1 monolitická aplikační paměť

paměťový blok

Tato nejjednodušší zpráva operační paměti rozděluje adresový prostor na dva **paměťové bloky** (= souvislá oblast adresového prostoru určená počáteční adresou — bází a velikostí). První blok je přidělen rutinám jádra a jeho datovým strukturám (včetně dat vlastního správce paměti), druhý může být jako celek přidělován na požádání procesům. První blok tedy tvoří tzv. paměť jádra (*kernel memory* – KM), druhý je paměť aplikační (*application memory* – AM).

Paměť jádra je sdílena všemi procesy, neboť rutiny jádra i jeho datové struktury jsou užívány všemi procesy při vykonávání služeb operačního systému (nikoliv však mimo tyto služby). Paměť aplikační je naproti tomu soukromá a přístup by k ní měl mít pouze proces-vlastník (ať již vykonává službu jádra či nikoliv).

Algoritmus přidělení regionu je jednoduchý. Pokud je aplikační paměť volná (= není alokována), je přidělena procesu celá bez ohledu na jím požadovanou velikost (samozřejmě nesmí převýšit velikost bloku). Jestliže není volná, je požadavek odmítnut (většinou s fatálními důsledky pro proces). Alokace se tudíž děje jen jednou při spuštění procesu a proces pak využívá takto alokovanou paměť po celou dobu svého života. Paměť je uvolněna při ukončení procesu.

Informace o obsazení paměti je triviální, stačí pouze registrovat aktuálního vlastníka aplikační paměti, jímž je aktuální (= právě běžící) proces.

Určitým problémem při alokaci regionu (bloku) je případná proměnnost jeho fyzického umístění (tj. báze aplikační paměti). Umístění se sice za dobu běhu dané instance operačního systému nemění, ale může se měnit například při změně verze (*upgrade*) operačního systému. Tato situace nastává, pokud paměť jádra předchází v adresovém prostoru paměť aplikační (velmi častý případ) a jádro se v nových verzích postupně zvětšuje (tj. báze bloku aplikační paměti se posunuje k vyšším adresám). Na druhou stranu překladač z vyšších programovacích jazyků produkuje kód, jenž užívá neměnných adres dat i návěští kódu, tj. příchod nové verze systému by si vyžadoval nový překlad všech v systému užívaných aplikací, aby odpovídaly nové bázi aplikační paměti, což je značně nepohodlné.

bázový registr

Naštěstí mnohé procesory podporují tzv. **bázový registr**, jehož obsah je automaticky přičítán k adrese použité ve strojovém kódu, a teprve tento součet je užit k fyzické adresaci paměťového místa. Překladač tak může generovat kód počínající od nulové adresy (tj. lokální adresy uvnitř regionu), jehož adresy jsou teprve za běhu systému převáděny na skutečné fyzické adresy dané aktuálním umístění bloku. Při startu procesu a alokaci paměti stačí uložit bázovou adresu přiděleného regionu (bloku) do bázového registru.

relokací

Pokud procesor postrádá podporu bázového registru, lze stejného efektu dosáhnout tzv. **relokace** (= znovuumístěním). Při relokaci během kopírování programu do alokovaného regionu je procházen kód programu a ke všem zde použitým adresám je přičtena bázová adresa aplikační paměti. Teprve takto modifikovaný program je spuštěn skokem na jeho první instrukci. Bohužel u většiny procesorů lze jen velmi obtížně identifikovat adresy ve strojovém (binárním) kódu, neboť nejsou nijak explicitně vyznačeny (instrukce, data, adresy apod. jsou rozlišeny pouze celkovým kontextem programu) ani nejsou na pevných místech (instrukce jsou různě dlouhé). Proto překladač doplňuje program o speciální datovou tabulku, která identifikuje pozici adres ve strojovém kódu a tato tzv. *relokační tabulka* je užita při každém spuštění (zavedení) procesu.

Ochrana paměti se v případě strategie monolitické aplikační paměti omezuje pouze na ochranu paměti jádra, jelikož v paměti nejsou (nepočítáme-li přirozeně region aktuálního procesu) žádné další regiony. Nejjednodušší metodou je použití bázového registru, jenž fyzicky znemožňuje použití fyzických adres nižších než je báze. Pokud je tedy v tomto registru uložena báze bloku aplikační paměti a paměť jádra leží před tímto blokem, nemůže proces paměť jádra tuto paměť ani adresovat (tj. leží mimo aktuální adresový prostor). V *režimu jádra* obsahuje bázový registr nulovou adresu a proces tak může přistupovat k celému adresovému prostoru (tj. nejen k paměti jádra, ale i k paměti aplikační). Nastavení registru je přirozeně privilegovanou instrukcí, jejíž volání je přípustné pouze v režimu jádra (v *uživatelském režimu* způsobí volání programového přerušení [procesorové výjimky], v rámci jehož obslužné rutiny může být proces-záškodník zlikvidován).

Strategie jediného bloku aplikační paměti je použitelná i ve víceúlohových systémech (a byla v nich skutečně užívána). Stačí pouze výše uvedený mechanismus správy paměti

rozšířit o možnost dočasného odkládání obsahu regionu do tzv. sekundární či odkládací paměti (např. na pevný disk). Při střídání procesů na procesoru, jež je pro víceúlohové systémy typické, se vyměňují i bloky paměti mezi primární (operační) pamětí a odkládacím zařízením, tj. při odsunutí procesu z procesoru je na disk ukládán obsah jeho aplikační paměti a následně je z disku obnoven datový region nově naplánovaného procesu. Je to sice funkční, ale velmi pomalé. V interaktivních systémech je proto tato strategie téměř nepoužitelná, neboť ukládání bloků paměti by výrazně zvýšilo reakční dobu procesů (proces, jenž je pozastaven, může na vnější událost reagovat až po obnovení svého běhu).

Správa paměti s monolitickou pamětí aplikací je velmi jednoduchá, má však i podstatné nedostatky. Hlavním nedostatkem je **nedostatečné využití operační paměti**, neboť proces ve většině případů obdrží více paměti než požadoval. Tato paměť není nikterak využívána, ale přesto nemůže být přidělena jinému procesu (a může být dokonce zbytečně odkládána na disk). Navíc existuje i **skrytá nevyužívaná paměť**, již tvoří ty úseky, jež si sice proces nárokoval, ale které nakonec nevyužije (například probíhá-li jinou větví algoritmu).

Mezi operační systémy, jež tento model užívaly, patří CPM (operační systém pro malé 8-bitové počítače z konce 70. let minulého století) a některé jeho rysy se objevují i v MS–DOSu (u tzv. COM spustitelných souborů, jež byly součástí režimu kompatibility s CPM).

2.2.2 statické bloky

statický blok

Jedním z možných řešení problémů s nedostatečným využitím operační paměti je rozdělení *aplikační paměti* na několik alokačních bloků. Pokud je velikost i uspořádání bloků během běhu systému neměnné, označujeme tyto ***bloky*** jako ***statické***. Velikost bloků se stanovuje při překladu OS nebo jeho zavádění, obecně jich je větší počet (ne tedy dva nebo tři) a mají různou velikost (počet a velikosti jsou dány požadavky klíčových aplikací provozovaných na daném OS.

OT: Jak bývají stanoveny velikosti statických paměťových bloků?

Žádá-li proces o přidělení regionu paměti je mu přidělen nejmenší z volný bloků dostatečné velikosti (pokud ovšem existuje alespoň jeden takový blok). Pro udržování informací o obsazení bloků stačí pole předem určené velikosti (= počtu statických bloků).

limitní registr

Mírně se komplikuje i ochrana paměti, neboť použití *bázového registru* chrání jen bloky na nižších adresách (a paměť jádra, pokud je na počátku FAP). Pro ochranu bloků na vyšších adresách je možno použít například tzv. ***limitní registr*** procesoru, jenž obsahuje velikost aktuálního regionu. Hodnota lokální adresy je ještě před přičtením k bázovému registru porovnávána s hodnotou limitního registru. Pokud je větší, je vyvolána výjimka, neboť proces se pokouší o zápis mimo region.

Největší změnou, jež přináší rozdělení na bloky, je možnost alokace více bloků jedním procesem. Adresový podprostor procesu tak přestává být jednolitý a může být dokonce nesouvislý (to nastane, jestliže proces získá dva regiony oddělené volnou pamětí nebo regionem jiného procesu). Adresový prostor procesu je většinou rozdělen na tři hlavní regiony:

kódový region

kódový region[*code region*, CR] (v Unixu se používá i historický a poněkud zavádějící termín *textový region*) — kódový region obsahuje (strojový) kód programu tj. posloupnost binárně representovaných instrukcí. Obsah regionu je tedy vykonáván jako program a až na výjimky do něj není zapisováno (samomodifikující se programy jsou ve většině operačních systémů zakázány)

datový region

datový region [*data region*, DR] — obsahuje statická data programu (tj. statické proměnné) resp. dynamicky alokovaná data (jen v některých OS). Na rozdíl od kódového regionu vyžaduje zápis a čtení a v některých systémech může měnit svou velikost.

zásobníkový region

zásobníkový region[stack region, SR] — obsahuje lokální (automatické) proměnné a návratové adresy funkcí organizované jako zásobník (tj. data se odebírají a přidávají pouze na jednom konci podle principu LIFO – *last-in first-out*). Ve většině hardwarových architektur roste zásobník směrem k nižším adresám tj. zásobník je umístěn na konci alokovaného regionu).

Strategie statických bloků umožňuje souběžnou existenci více procesů i bez odkládání do sekundární paměti. Počet souběžně existujících procesů je však omezen počtem bloků a omezená nabídka bloků odpovídající velikosti vnáší silné závislosti mezi procesy (např. proces nemůže být dokončen resp. ani spuštěn, dokud jiný proces neuvolní použitelný blok aplikační paměti). Tuto situaci lze sice zmírnit možností odkládání překážejících regionů neaktuálních procesů, to však vede k dalším problémům (např. k fragmentaci odkládacího prostoru, neboť odkládané bloky mají různou velikost).

Správy paměti založené na statických blocích se však stále používají pro správu paměti jádra, kde lze požadavky jednotlivých rutin jádra kvalifikovaně odhadnout a lze předvídat i jejich časovou následnost. Jedná se však o poněkud vyspělejší strategie, jako je např. tzv. *buddy systém*, který umožňuje efektivní (především velmi rychlou) alokaci a správu malých regionů o velikosti, jež je mocninou dvou.

2.2.3 dynamické bloky

dynamické bloky

Aplikační paměť lze rozdělit i na bloky, jejichž velikost se dynamicky přizpůsobuje požadavků procesů (**dynamický bloky**). V počátečním stavu (před alokací prvního regionu) tvoří aplikační adresový prostor jediný volný blok.

Při alokaci se vyhledá první přípustný blok (tj. volný blok větší požadavku) tzv. *first fit* resp. nejmenší z vhodných (*best fit*). Pokud je jeho velikost rovna požadavku (či přesněji je velmi blízká, neboť požadavek je ve většině případů zaokrouhlen na vhodnou mocninu dvou), je blok přidělen celý, v ostatních případech (blok je zbytečně velký) je rozdělen na dva bloky, první o požadované velikosti je přidělen procesu, druhý zůstává volný.

Jaká strategie je lepší *best fist* nebo *first fit*? Zdůvodněte.

Při uvolňování se musí provádět tzv. zcelování volných bloků. Pokud po uvolnění bloku vznikne souvislá řada dvou nebo tří bloků, jsou tyto bloky spojeny do jediného bloku.

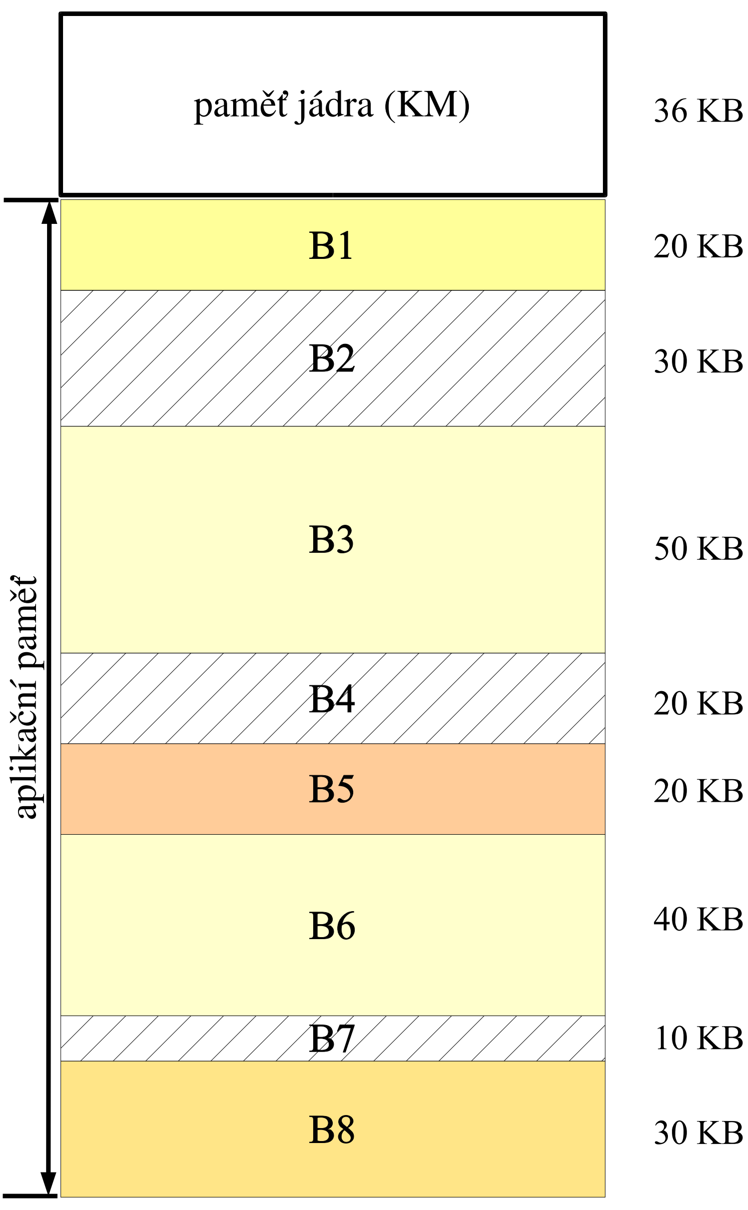
Pro representaci obsazení paměti není příliš vhodné statické pole, neboť počet bloků se může výrazně měnit (od jednoho bloku po desítky bloků). Z tohoto důvodu se u dynamických bloků dává přednost umístění základních informací o bloku přímo do tohoto bloku resp. bezprostředně před něj (tj. informace tvoří jakousi hlavičku bloku). Hlavička bloku obsahuje údaj o velikosti bloku, údaj zda jde o poslední blok v aplikační paměti, a samozřejmě i identifikaci procesu-vlastníka (pokud existuje). Jádro si tak udržuje pouze

informaci o bázi prvního bloku, ostatní bloky (a jejich hlavičky) lze nalézt procházením zřetězeného seznamu hlaviček.

Ochrana paměti je u dynamických bloků obdobná blokům statickým, tj. nejjednodušší je použití bázového a mezního registru. Zvláštní pozornost je nutno věnovat ochraně hlaviček dynamických bloků, jejichž narušení může vést k degradaci funkce celé správy paměti (hlavičky bloků by měly z hlediska ochrany ležet vně svých bloků).

fragmentace paměti

Hlavním problémem užití dynamických bloků je sklon k **fragmentaci operační paměti**, tj. ke vzniku mnoha malých nesouvislých bloků volné paměti. Důsledkem je situace, kdy je sice relativně velké množství volné paměti, ale nikoliv souvislé bloky rozumné velikosti (tj. např. bloky dostatečně vhodné pro kódové nebo datové bloky typických aplikací). Na obrázku je např. 60 KB volné paměti, ale největší volný blok má velikost pouze 30 KB (volné bloky jsou vyšrafovány pošikem).



Fragmentace je **fatálním** problémem především u víceúlohových systémů, kde žádosti o alokaci a dealokaci regionů mohou přicházet v libovolném a zcela nezávislém pořadí. U těchto systémů vede fragmentace v kratším či delším čase k degradaci celého systému (procesy stále čekají na paměť, nové nelze spouštět atd.). U procesů se vzájemným

voláním procesů (proces musí počkat na dokončení svého potomka) je situace o něco lepší, neboť požadavky na alokaci jsou na sobě závislé (naposledy alokovaná paměť je jako první dealokována). Proto byl tento systém použit mimo jiné např. v MS-DOSu pro správu standardní aplikační paměti.

2.2.4 setřásání bloků

setřásání bloků

Problém s fragmentací lze odstranit tzv. **setřásáním bloků**, tj. přesunem bloků vedoucím k soustředění volné paměti v jediném bloku (jenž je většinou na konci paměti). Bohužel ani tato strategie není bez problémů.

Prvním je nutná existence alespoň minimální technické podpory na úrovni procesoru, neboť veškeré přesuny by měly být na aplikační úrovni zcela transparentní. Většinu programů by totiž překvapilo, kdyby například během kopírování řetězce (v C funkcí *strcpy*) došlo k přesunu cílové paměti (část znaků by se kopírovala do jiného regionu či volné paměti). Řešením je přirozeně povinné použití bázového registru, neboť zde stačí po provedeném posunu zaměnit hodnotu bázového registru na bázi nové pozice regionu (lokální vnitroregionové adresy používané se nemění). Bohužel implicitní bázový registr neposkytují všechny procesory. I pro ně se však našlo řešení, i když velmi problematické — nucené využívání handlerů (popisovačů) paměti.

Při alokaci je procesu vrácena nikoliv bázová adresa regionu, ale tzv. handler paměti což je *de facto* ukazatel na paměťové místo v jádře, jež samo obsahuje bázovou adresu bloku (tj. ukazatel na počátek přiděleného bloku). Programátor musí pro každý přístup k regionu použít handler, nikoliv přímou adresu k paměti, neboť ta se stane po každém přesunu při setřesení neplatnou (handler nikoliv, správce změní pouze adresu uloženou v handleru). Pokud chce provést přímý přístup, musí po nezbytně nutnou dobu zabránit přesunu bloku v paměti (tzv. region uzamknout). Na toto uzamknutí nesmí zapomenout, neboť k přesunu může dojít téměř kdykoliv (nelze tudíž otestovat polohu regionu a následně provést přístupovou operaci, neboť k posunu může dojít i mezi těmito dvěma operacemi). Nesmí však také nechat paměť uzamknutu příliš dlouho, neboť uzamčený blok může bránit dokonalému setřesení a tj. i odstranění případné fragmentace. I přes veškerou snahu programátora však i nejkratší možné zamknutí vnáší ve svém důsledku nepříjemnou závislost mezi procesy, vedoucí nakonec k degradaci systému.

2.3 Virtualizace paměti

Přehled základních metod správy operační paměti uvedený v předchozí kapitole, není příliš optimistický, neboť žádná z jednoduchých správ není bez chyby a jejich použití je tudíž vždy limitované (žádná z nich se například příliš nehodí pro moderní interaktivní víceúlohové systémy).

Řešením na vyšší úrovni je úplná virtualizace paměti. Při úplné virtualizaci paměti může proces používat libovolné adresy z určitého (konečného) intervalu a adresovat tak souvislý adresový prostor (s počátkem v nulové adrese), jež však nemusí být fyzicky representován souvislým blokem operační pamětí. Velikost tohoto tzv. *virtuálního* či lépe **logického adresového prostoru** není omezena velikostí skutečné operační paměti, ale pouze velikostí adresy (například u 32bitových adres má logický adresový prostor rozsah 2^{32}bytů = 4 GB, u 64bitových dokonce 2^{64} = 16 exabytů [EB]).

logický adresový prostor

Tento logický adresový prostor lze z hlediska fyzické representace rozdělit do dvou základních částí. První část je tvořena adresami, jež nejsou součástí alokovaných regionů (i v logické paměti lze přirozeně alokovat regiony) resp. adresami, jež nebyly použity (tj. nebylo z nich čteno či do nich zapisováno). Tato část nemusí mít a většinou ani nemá žádnou fyzickou representaci (tj. data z této části nejsou nikde uložena). Tato část tvoří v mnoha případech větší část logického adresového prostoru (u 64 bitových adres vždy).

Druhá část obsahuje adresy, na nichž jsou uložena skutečná data a musí mít tudíž i skutečnou fyzickou representaci. Tuto část lze opět rozdělit na několik podčástí podle místa uložení dat. V současnosti jsou data ukládána buď v operační (primární) paměti [RAM] nebo na vnějších (sekundárních) paměťových zařízeních (tj. dnes převážně pevných discích). Důvodem tohoto rozdělení jsou rozdílné charakteristiky obou paměťových technologií: operační paměť je rychlá, ale relativně malá, disk má vyšší kapacitu (řádově 100× při srovnatelné ceně), ale je relativně pomalý (řádově 100 000×). Tj. z hlediska ceny a kapacitních možností by data z logického adresového měla být umístěna na disku, z hlediska rychlého běhu aplikací v operační paměti. **Virtualizátorirtualizátor**(= systémové rutiny zajišťující virtualizaci paměti, součást správy paměti) řeší tento rozpor tím, že právě užívaná data jsou umístěna v operační paměti, data delší dobu neužívaná jsou odložena na disku a mezi oběma úložišti probíhá obousměrná výměna dat podle aktuálních požadavků aplikace/aplikací.

virtualizátor

Hlavní podstatou virtualizace (a hlavní náplní práce virtualizátorů) je převod požadavků na přístup k logickému adresovému prostoru (= paměťovému místu určenému logickou adresou) na přístup k fyzickým datům na skutečném paměťovém zařízení. Pokud existuje podpora na úrovni procesoru, může se tento překlad dít zcela transparentně, tj. aplikace nemusí vědět, kde a jak jsou fyzicky uložena, tj. postačí jí pouze znalost logické adresy (nebo intervalu adres). Výhodou úplné transparentnosti je nezměněný tvar programů na úrovni strojového kódu, neboť z hlediska aplikace je nevýznamné, zda je adresa paměťového místa chápána jako fyzická (lze ji užít k přímé adresaci paměti) nebo logická (adresa musí být procesorem přeložena resp. musí být načtena z pevného disku).

V neposlední řadě je nutno zdůraznit, že virtualizace (resp. virtualizátor) není koncovým správcem paměti, a musí být doplněn klasickým správcem paměti, jenž však již nespravuje fyzický adresový prostor (jenž je u většiny počítačů jediný a prostorově značně omezený), ale prostor logický (těch může jich existovat více a může být téměř neomezený). Pro správu logické paměti lze použít libovolnou dříve uvedenou strategii (ve většině případů vystačíme dokonce s tou nejjednodušší). Díky charakteru logické paměti se však většina nevýhod těchto typů správy minimalizuje či dokonce eliminuje.

Například při použití strategie dynamických bloků pro logickou paměť může sice stále docházet k fragmentaci logické paměti, jež se však při vhodně zvolené metodě virtualizace (např. u stránkování) neprojevuje na fyzické paměti (fyzickou paměť lze přidělovat i v nesouvislých blocích). Fragmentace logického adresového prostoru je sice nepříjemná, avšak méně nebezpečná (rozsah logického prostoru je řádově větší a navíc je užíván jen jedním procesem).

Nejužívanější metodou virtualizace paměti je tzv. stránkování, jež je podporováno většinou současných procesorů včetně rodiny procesorů Intel počínaje procesorem 80386 (dále jen Intel386+).

2.3.1 stránkování

stránkování

Stránkování (angl. *paging*) je hardwarový mechanismus umožňující plnou virtualizaci paměti. Stránkování je dnes prováděno nejčastěji procesorem (dříve však bylo poskytováno speciálním obvodem — MMU [*memory management unit*], jenž však s procesorem úzce spolupracoval). Mechanismus stránkování má dvě základní části:

- *překlad adres* (logická adresa je překládána na fyzickou)
- *výpadek stránky* (přerušení při přístupu na neplatnou stránku)

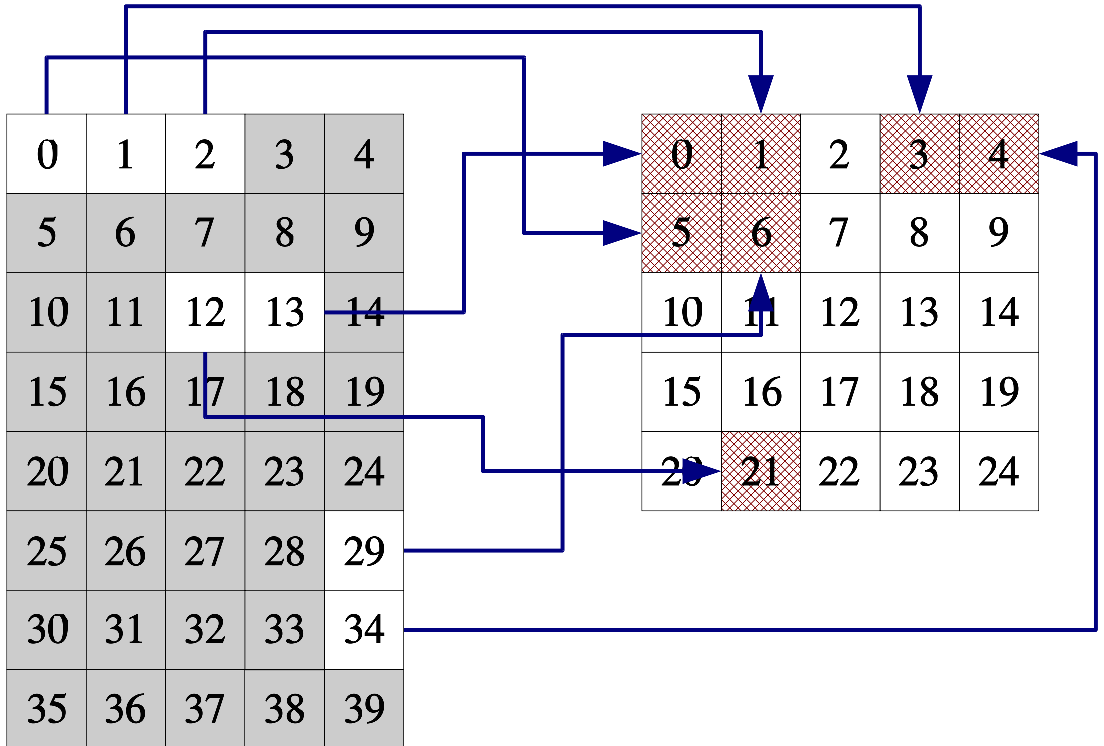
překlad adres

logická stránka

rámec

neplatná stránka

Překlad adresspojený se stránkováním vychází z představy rozdělení logického adresového prostoru (= souvislý prostor adresovaný logickými adresami) na ***(logické) stránky*** stejné velikosti, jež se vzájemně nepřekrývají. Velikost stránek se u různý architektur může lišit, typicky je v řádu jednotek kilobytů (u procesorů Intel 386+ jsou to 4 KB). Na stejné stránky se myšleně rozdělí i fyzický adresový prostor (tyto stránky se označují jako ***stránky fyzické*** či spíše jako ***(stránkové)*** **rámce** [*page frame*]). Mezi jednotlivými logickými a fyzickými stránkami existuje zobrazení, jež však nemůže být úplné (tj. množiny na množinu). Důvodem je skutečnost, že velikost logického adresového prostoru (je dán velikostí logické adresy) je výrazně větší než velikost fyzického adresového prostoru (tím spíše, že ve skutečnosti se do fyzického adresového prostoru mapuje hned několik logických adresových prostorů). Proto vždy existují logické stránky, jež nemají přiřazen rámec (fyzickou stránku), tzv. **neplatné stránky** [*invalid page*]. Mohou však existovat i rámce, na něž není mapována žádná logická stránka (tzv. *volné rámce*). Prozatím navíc předpokládejme, že zobrazení je prosté (tj. na rámec je mapována nejvýše jedna logická stránka).



Logický adresový prostor

platná stránka

neplatná stránka

Fyzický adresový prostor

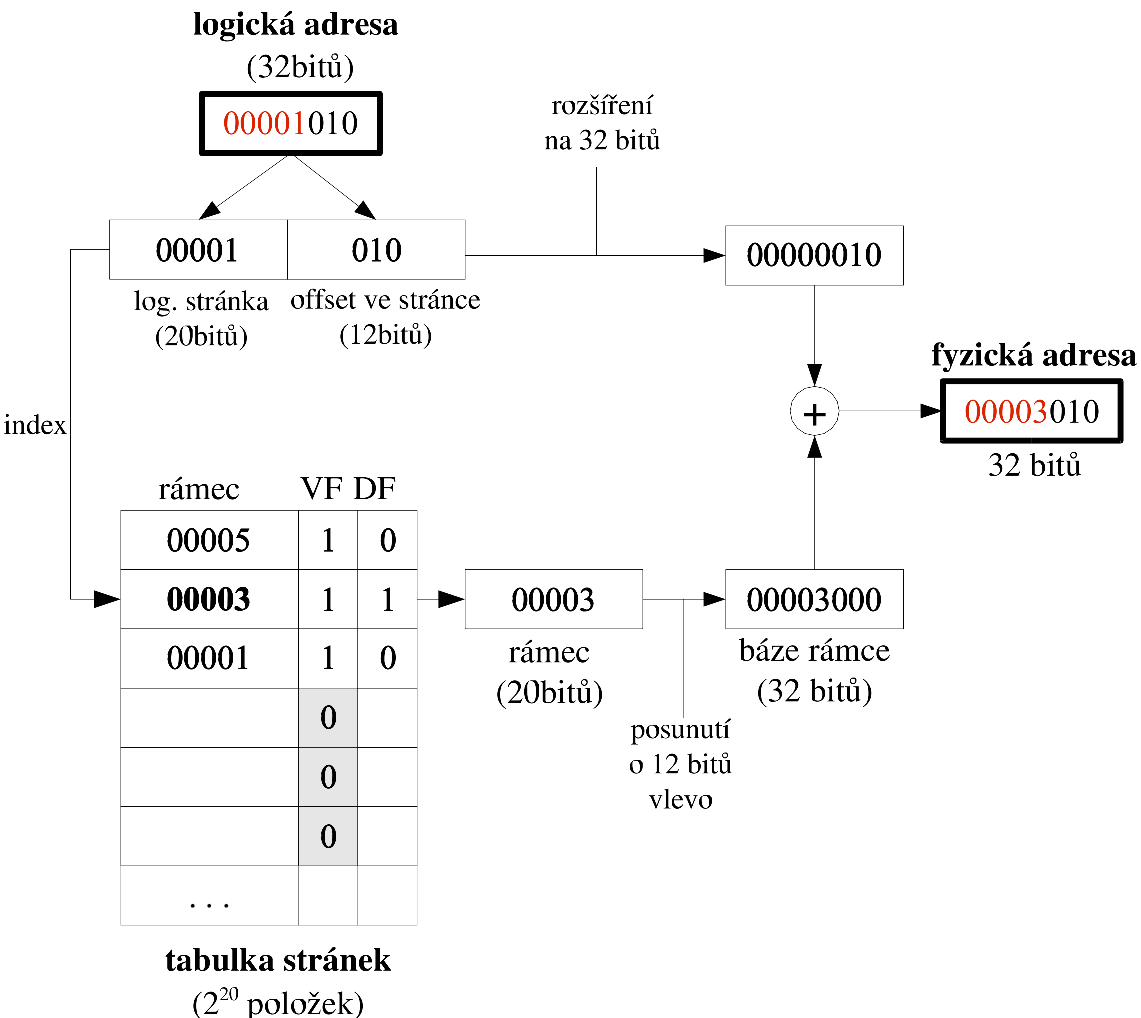
volný rámec

obsazený rámec

Za povšimnutí stojí především zcela libovolný charakter zobrazení (např. zobrazení nemusí zachovávat uspořádání ani jiné relace mezi logickými stránkami), které vede ke zdánlivě chaotickému obrazu souvislých regionů logické paměti (zde např. stránky 0-2). Navíc je nutno zdůraznit, že ve skutečnosti je počet logických stránek i rámců o mnoho řádů vyšší (např. u platformy Intel 386 je logický adresový prostor rozdělen na $2^{20} \doteq 1,05 \cdot 10^6$ logických stránek (počet rámců závisí na velikosti instalované fyzické paměti, u mého počítače je jich $16\times$ méně než logických).

tabulka stránek

Nyní již můžeme přistoupit k popisu vlastního mechanismu překladu adres při stránkování. Rozdělení logického adresového prostoru na stránky umožňuje každou logickou adresu rozdělit na dvě části – první (s vyššími bity) označuje logickou stránku (tj. její pořadí), druhá nižší obsahuje adresu paměťového místa vztaženou k dané stránce (tzv. stránkový offset). Přímým procesem překladu projde pouze číslo logické stránky. Překlad se děje převodem čísla (pořadí) logické stránky na číslo (pořadí) rámce prostřednictvím tzv. **tabulky stránek**. Tabulka obsahuje položky, jež jsou indexovány číslem logické stránky a obsahují číslo rámce a skupinu příznaků. Získané číslo rámce fyzické stránky je bitovým posunutím vlevo převedeno na bázovou (počáteční) adresu rámce ve fyzické paměti. Posledním krokem je přičtení stránkového offsetu k bázové adrese rámce, čímž se již získá fyzická adresa.



Mírně zjednodušený mechanismus překladu adres v 32bitovém systému procesorů Intel 386+ (logická i fyzická adresa je 32bitová) na obrázku Obrázek předpokládá stránky o velikosti 4 KB. Adresy jsou hexadecimální, v tabulce stránek jsou znázorněny i nejdůležitější příznaky (VF = příznak platnosti [Validity Flag], DF = příznak zápisu do stránky [Dirty Flag]).

Detailnější informace o procesu stránkování (včetně stručného popisu MMU u procesorů ARM) najdete na např. v poznámkách:

Paul Krzyzanowski. *Memory Management: Paging.*
Dostupné na `https://www.cs.rutgers.edu/~pxk/416/notes/10-paging.html`.

I když výše uvedený popis i příklad postihují hlavní podstatu mechanismu překladu, je jeho reálná implementace složitější, neboť výše uvedený mechanismus má příliš velké paměťové nároky (například jen tabulka stránek má rozměr 4 MB) a je příliš pomalý (při každém přístupu do paměti se musí provést dvě aritmetické operace). Řešením jsou velmi rychlé asociativní paměti stránek (obsahují mapování pro naposledy užité logické stránky a a jsou indexovatelné v konstantním čase) resp. použitím víceúrovňového překladu logické stránky na fyzickou pomocí několika úrovní tabulek (např. procesory Intel386+ užívají dvě úrovně, kde prvních 10 bitů čísla logické stránky indexuje tzv. adresář stránek, jehož položky ukazují na vlastní tabulky stránek, jež jsou pak indexovány druhými deseti bity čísla).

výpadek stránky

Co však nastane pokud proces přistoupí k neplatné stránce? Za této situace se uplatní druhý mechanismus spojený se stránkováním — **výpadek stránky** (*page fault*). Výpadek

stránky je interním přerušením (výjimkou), jež přeruší instrukci přistupující k neplatné stránce a předá řízení obslužné rutině v jádře systému. Tato rutina má v podstatě dvě možnosti jak na situaci reagovat:

- nalezne odpovídající fyzickou stránku a zajistí její propojení s logickou stránkou a logickou stránku následně zplatní (tj. na úrovni tabulek stránek uloží index fyzické stránky a nastaví bit platnosti)
- ukončí proces tj. neprovede ani návrat z obsluhy přerušení výpadku stránky místo toho aktivuje jiný proces resp. se procesu pošle výjimka (Win32) resp. signál (Unix), na nějž může proces ještě reagovat

O tom, jak obslužná rutina zareaguje, rozhoduje umístění logické stránky v logickém adresovém prostoru. Pokud stránka leží v alokovaném regionu, bude propojena s rámcem, jinak je proces ukončen, neboť se pokusil přistoupit k prostředku, který není k dispozici a jedná se tudíž s největší pravděpodobností o chybu programu.

Poslední fáze výpadku stránek (pokud není proces bezprostředně ukončen) spočívá v návratu z přerušení a opětném provedení přerušené instrukce, která již na druhý pokus výpadek stránky nezpůsobí (stránka je platná). Je nutno zdůraznit, že restartování instrukce není v moderních procesorech triviální operací, neboť instrukce jsou prováděny paralelně a jsou různě optimalizovány (MMU tak tvoří relativně významnou část procesoru).

Kromě mapování adres podporuje tabulka stránek i ochranu paměti. Nejčastěji je v každé položce tabulky soubor bitů definujících přípustné operace s daty adresovanými danou logickou stránkou (ať již jsou v operační paměti či odkládacím souboru). Běžný je systém RWX (*Read-Write-eXecute*) resp. jednodušší systém RW (vykonávání kódu je chápáno jako speciální případ čtení). Navíc bývá přítomen i bitový příznak určující, zda je možno ke stránce přistupovat v obou režimech (uživatelském či privilegovaném) nebo pouze privilegovaném (tj. stránka je přístupná pouze rutinám jádra).

2.3.2 logický adresový prostor procesu

Virtualizace paměti stránkováním umožňuje nejen vytvoření rozsáhlého adresového prostoru nad operační pamětí a odkládacím prostorem, ale umožňuje i vytvoření jedinečného logického adresového prostoru pro každý proces v systému.

Pro dosažení tohoto efektu musí mít každý proces vlastní tabulku stránek (nad níž si definuje své paměťové regiony). Při výměně procesů v rámci multitaskingu pak stačí uložit do specializovaného registru procesoru bázovou adresu tabulky stránek procesu, jež se stává běžícím (aktuálním) a tím změnit i interpretaci všech používaných adres (tj. stejné logické adresy začnou odkazovat zcela jiná data).

I když je přepnutí adresových prostorů jednoduchou operací (jedinou instrukcí je změněn obsah jediného adresového prostoru), jsou jeho důsledky obrovské, neboť touto jedinou instrukcí se pro procesor vlastně změní celý okolní svět. Aby však proces bezprostředně po změně světa mohl pokračovat, nesmí dojít ke změně v jeho bezprostředním okolí, tj. na stránce obsahující aktuálně vykonávaný kód. Toho je možno dosáhnout jen tehdy, pokud je aktuální logická stránka kódu mapována v obou prostorech na stejný rámec, což je možné pouze v jádře.

Logický adresový prostor každého procesu je rozdělen na dvě velké části. První obsahuje regiony užívané procesem v uživatelském režimu, tj. na úrovni aplikačního programu (kódový, datový, zásobníkový). Druhý je obrazem jádra, jeho rutin a systémových dat.

Pro aplikační oblast je typické ostré oddělení fyzické paměti užívané jednotlivými procesy, tj. regiony jednotlivých procesů se mapují na disjunktní množinu rámců operační paměti a bloků odkládací paměti. Toto rozdělení zcela znemožňuje procesům přístup do fyzické paměti ostatních procesů a tím brání negativnímu (neřízenému, chybovému) vzájemnému ovlivňování procesů. Proces může přistupovat pouze ke svému adresovému prostoru a tím pouze a jen ke svým rámcům fyzické paměti a odkládacího prostoru, přístup k ostatním rámcům je zcela nemožný (nejsou totiž mapovány ze žádné ze stránek jeho logického prostoru). Samozřejmě na požádání lze snadno implementovat i sdílenou (fyzickou) paměť, což může ušetřit operační paměť resp. umožnit přímou komunikaci procesů (viz kapitola 2.3.6 na straně 38)

Přesným opakem je oblast logického adresového prostoru užívaná jádrem, jejíž fyzická realizace (rámce apod.) je sdílena všemi procesy. Jinak řečeno, jádro existuje ve fyzické paměti v jediné kopii, která je mapována z logických adresových prostorů všech procesů (tj. virtuálně má každý proces své vlastní jádro). Fyzicky je toto sdílení realizováno shodným obsahem položek tabulky stránek v celé oblasti logického adresového prostoru jádra.

Zrcadlení jádra v adresovém prostoru procesů sice zmenšuje rozsah logických adres použitelných v aplikaci, je však nezbytné. Pokud by totiž jádro užívalo oddělený adresový prostor (což je technický možné, stačilo by aby k přepnutí došlo na začátku a konci každého volání služby jádra), nemohlo by kopírovat data mezi aplikačním adresovým prostorem a buffery na úrovni jádra. Toto kopírování je vyžadováno většinou služeb jádra, například při zápisu na disk je nutno kopírovat zapisovaná data z bufferu na aplikační úrovni do vyrovnávací paměti disku v jádře. Zrcadlení také mírně komplikuje ochranu paměti jádra před aplikačním kódem, neboť paměť jádra je adresovatelná i v uživatelském režimu (leží ve stejném adresovém prostoru!). Z tohoto důvodu musí být každá logická stránka opatřena příznakem, zda je přístupná v obou režimech (aplikační paměť) nebo jen v privilegovaném režimu jádra (chráněná paměť jádra). Pokud ke chráněné paměti přistoupí aplikační kód (v neprivilegovaném režimu), je vyvolána procesorem výjimka, v jejímž rámci je provinilý proces ukončen (resp. je mu poslána programová výjimka [WinNT] či signál [Unix] na níž reaguje proces ukončením, má však šanci uložit rozpracovaná data).

Přibližme se základní strukturu **logického adresového prostoru běžného procesu** (předpokládejme např. 32bitový logický prostor o velikosti 4GB). Na počátku tohoto prostoru (většinou to však není přímo od nulové adresy) leží kódový region o velikosti řádově od desítek KB až po desítky MB (běžně však maximálně v řádu MB). Velikost tohoto regionu se v průběhu života procesu nemění. Bezprostředně za kódovým regionem leží region datový o velikosti od několika KB po stovky MB. Velikost tohoto regionu se může měnit, neboť proces může žádat další paměť prostřednictvím explicitní dynamické alokace (v jazyce C funkce *malloc*, v C++ operátor *new*). Mnohé operační systémy (včetně WinNT a Linuxu) však pro některé dynamicky alokované paměti (především jsou-li rozsáhlé) vytvářejí zvláštní regiony.

Pevné umístění obou hlavních regionů na počátku logického adresového prostoru zcela eliminuje problémy s relokací paměti (paměťový obraz všech instancí je stejný).

Následuje velký blok paměti, jenž je na počátku existence procesu volný (v rozsahu řádu GB). Tento volný prostor je však postupně zaplňován rozšiřováním okolních standardních regionů (zespodu datovým regionem, shora zásobníkovým). Navíc jsou v tomto volném prostoru vytvářeny další nestandardní regiony počínaje regiony dynamicky linkovaných knihoven a sdílenými datovými pamětmi po rozsáhlé dynamicky alokované bloky.

Po volném bloku paměti následuje zásobníkový region. Také tento region může během existence procesu postupně růst, a to v souladu s organizací zásobníku směrem dolů (tj. k nižším adresám). Velikost zásobníku jen výjimečně přesáhne řád MB.

Zbytek adresového prostoru (v řádu GB, u mnohých přenositelných systémů jsou to 2GB, neboť tuto velikost si vynucují některé procesory) je vyhrazen pro jádro (tj. je spravováno zvláštním správcem). Pouze malá část (v rozsahu maximálně desítek MB je však skutečně alokována a ještě menší část je skutečně využita). Tato paměť je přístupná pouze v privilegovaném režimu, tj. z rutin jádra operačního systému.

2.3.3 stránkování na žádost

Na základě virtualizace paměti založené na stránkování lze vybudovat vícero typů správ paměti, počínaje transparentním setřásáním dynamických bloků (což však není příliš rozumné užití virtualizace) až po různé mechanismy odkládání procesů na odkládací (swapovací) zařízení. V současnosti je však téměř výhradně užíváno tzv. **stránkování na žádost** (*demand paging*). Základním principem stránkování na žádost je *lenivé vykonávání*, tj. vše je provedeno až v okamžiku, kdy je toho skutečně třeba.

stránkování na žádost

Tento princip se projevuje i v základním východisku stránkování na žádost: po přidělení *regionu jsou všechny jeho stránky neplatné* a platnými se stávají až při prvním přístupu k nim (tj. až po výpadku stránky a následném zplatnění v obslužné rutině přerušení). Jinak řečeno, fyzická paměť je procesu přidělována až v okamžiku, kdy ji skutečně potřebuje. Logické stránky uvnitř regionů, které jsou neplatné, protože k nim proces ještě ani jednou nepřistoupil (tj. do nich nezapisoval, ani je nečetl, ani nevykonával jejich kód) označuji dále jako **panenské stránky**.

panenska stránka

Jak však systém vyřeší výpadek stránky při přístupu k panenské stránce? To závisí na druhu regionu či dokonce menšího úseku paměti, v němž tato logická stránka leží.

Podívejme se nejprve na *kódový region*. V něm dojde k výpadku panenské stránky v okamžiku, kdy je poprvé prováděna instrukce ležící na této stránce (nemusí to být první instrukce na této stránce, program může dosáhnout stránky programovým skokem). Aby obslužná rutina zajistila zplatnění stránky, musí za prvé nalézt volný rámec (pokud není žádná fyzická stránka volná, musí ji ukrást, viz dále), za druhé musí do tohoto rámce nahrát odpovídající segment spustitelného souboru (celý obraz kódového regionu je uložen ve spustitelném souboru), za třetí rámec propojit s logickou stránkou (v tabulce stránek se v položce odpovídající pozici dané logické stránky uloží identifikace rámce) a nakonec se stránka zplatní (nastaví se bit platnosti v dané položce tabulky stránek). Poté výpadek stránky skončí návratem z přerušení a procesor opakovaným čtením požadovanou instrukci načte. Po určité době se tok programu přesune do další stránky, způsobí výpadek a vše se opakuje pro další logickou stránku. Nakonec zůstanou panenskými pouze stránky ty stránky kódového regionu, jimiž řízení programu neprošlo (stránkování na žádost tak minimalizuje skryté nevyužívání logických stránek!).

Složitější situace je u *datového regionu*. Zde také dojde k výpadku panenské stránky při prvním přístupu do stránky (z hlediska vyšších programovacích jazyků např. při přístupu ke statické proměnné resp. k dynamicky alokovanému poli či struktuře), ale reakce se liší v závislosti na požadované počáteční hodnotě daného paměťového místa.

Nejjednodušší je situace v případě, že na iniciální hodnotě nezáleží (tj. může být náhodná). Zde stačí logickou stránku propojit s libovolnou volnou fyz. stránkou a logickou stránku zplatnit (to vše je jednoduchá a velmi rychlá operace nad odpovídající položkou

tabulky stránek). U systémů, jež kladou důraz na bezpečnost, musí systém zajistit, že použitá stránka neobsahuje citlivé informace. Jinak by se totiž mohlo stát, že se v některém dynamicky alokovaném poli běžných uživatelů objeví fragment utajované informace ukradený jinému procesu (např. heslo administrátora). Nejjednodušší strategií je v tomto případě preferovat rámce, jež byly dříve vlastněny stejným uživatelem a pokud se jich nedostává, pak ostatní před použitím vynulovat.

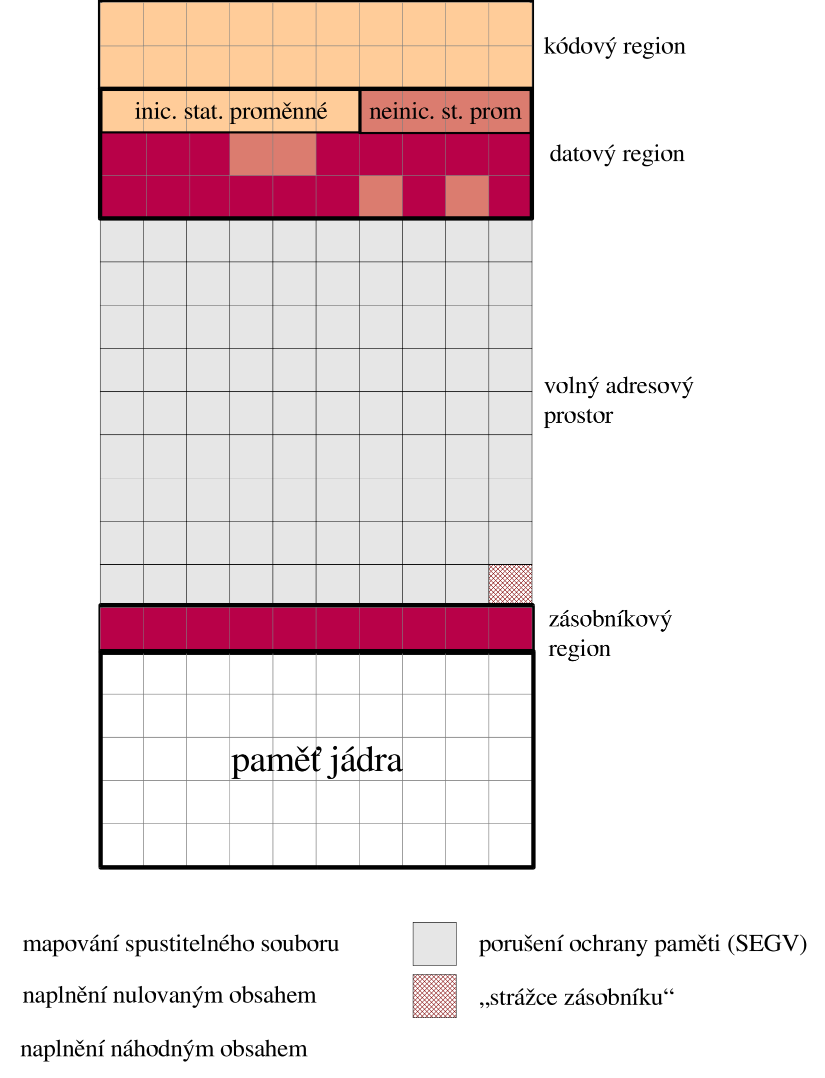
Pokud je požadována nulová hodnota (tak je tomu u neinicializovaných statických proměnných resp. u některých typů dynamické alokace [např. funkcí *calloc*]), je použit podobný přístup jako v předchozím případě, pouze při vyhledávání volného rámce jsou preferovány rámce, jež jsou vyplněny pouze nulovou hodnotou. Pokud se takovýto rámec nenajde, vezme systém zavděk libovolným volným rámcem, jehož však před připojením vynuluje. Virtualizátor tedy musí udržovat informaci nejen o volných rámcích, ale i rámcích jež mají (s jistotou) nulovou hodnotu, neboť ty jsou relativně cenným prostředkem (nulování je operace, jež sice trvá pouze mikrosekundy, ale i tak může častým výskytem výrazně zvýšit režii systému).

Posledním podtypem jsou stránky na pozicích statických proměnných s nenulovou počáteční (iniciální) hodnotou (tj. v definici inicializované statické globální resp. lokální proměnné). Iniciální hodnoty těchto proměnných jsou uloženy ve spustitelném souboru, kde tvoří zvláštní datový blok. Tento blok je jedinou částí datového regionu, jež je ve spustitelném souboru uložen (u neinicializovaných statických proměnných je uložena pouze velikost daného podregionu). Přístup k podregionu inicializovaných statických proměnných je řešen obdobně jako přístup ke kódu, tj. mapováním daného bloku spustitelného souboru do logické paměti tj. při výpadku panenské(!) stránky je vzat volný rámec, do něj je zkopírována odpovídající část spustitelného souboru a následně je rámec s danou logickou stránkou propojen.

Posledním obecným regionem v logickém adresovém prostoru je *region zásobníkový*. Panenské neplatné stránky tohoto regionu jsou pouze jediného druhu, neboť při jejich výpadku stačí připojit libovolný volný rámec, přičemž nulování není vyžadováno (zásobník nevyžaduje počáteční vynulování).

Paměťové stránky mimo regiony jsou přirozeně neplatné a při jejich výpadku je vyvolána výjimka (signál) signalizující porušení ochrany paměti, na níž proces nejčastěji reaguje svým předčasným ukončením. Jedinou výjimkou je u některých operačních systémů tzv. „strážce zásobníku“, což je stránka bezprostředně před zásobníkovým regionem. Pokud totiž zásobník při svém postupném růstu (směrem dolů!) překročí hranici zásobníkového regionu, může operační systém tento region transparentně zvětší, aby tak obsloužil zvýšené požadavky procesu (dané např. masivním použitím rekurze). Operačnímu systému stačí, pokud procesu na počátku přidělí relativně malý zásobník, jenž je však pro většinu procesů dostatečný, a opatří jej strážní stránkou. Přetečení zásobníkového regionu je detekováno prvním výpadkem strážní stránky a ošetřeno uvnitř obsluhy výpadku stránky zvětšením regionu v řádu MB a vytvořením nové strážní stránky, dokud se nevyčerpá volný adresový prostor před regionem resp. nepřekročí maximální velikost zásobníku.

Zjednodušený stav adresového prostoru procesu bezprostředně před vykonáním první instrukce (tj. i před prvním výpadkem stránky) znázorňuje následující obrázek. Označeny jsou jednotlivé regiony (včetně paměti jádra) v klasickém uspořádání a především je naznačena distribuce jednotlivých typů panenských (neplatných) stránek v aplikační části paměťového prostoru (viz legenda v dolní části obrázku).



2.3.4 jak si ukrást stránku

Výše uvedený přehled reakcí systému na výpadky stránek mlčky předpokládal (až na malé výjimky), že fyzické operační paměti je k dispozici téměř neomezené množství, tj. obsluha výpadku stránky vždy najde volný rámec. To je však pouze ideální stav, neboť operační paměti není nikdy dost a systém je tak nucen odebírat rámce (fyzické stránky) procesům, kteří tyto rámce vlastní (jsou propojeny s platnými logickými stránkami, jež leží v regionech v logických adresových prostorech těchto procesů), ale dostatečně je nevyužívají (resp. je sice využívají, ale už není kde brát).

kradení stránek

Tento proces, jenž je nejčastěji označován jako **kradení stránek** (*page stealing*), se výrazným způsobem promítá do mechanismu stránkování na žádost (činí ho poněkud složitějším).

Kradení fyzické stránky probíhá v následujících fázích:

počáteční podmínka: nenalezení volného rámce

Virtualizátor si udržuje informace o využití všech rámců ve spravované části operační paměti (operační paměť může být využita pro ukládání informací, jež nejsou spravovány správcem paměti — např. vyrovnávacích pamětí souborového systému). Každá položka

tabulka rámců

tzv. **tabulky rámců** popisuje jeho stav (volný, obsazený) a popřípadě i identifikace bloku odkládací paměti, jež obsahuje obraz daného rámce (rámec byl v minulosti již alespoň jednou ukraden). Při výpadku stránky je tabulka prohledána a pokud není nalezena volná stránka, je zahájen proces kradení rámce.

vytipování vhodného rámce — kandidáta na ukradení

Základní strategie vytipování nejvhodnějšího kandidáta-rámce na ukradení, tj. rámce, jehož ukradení by co nejméně zpomalilo systém a dotčené aplikace, jsou detailně popsány v následující kapitole. Nyní stačí předpokládat, že výsledkem této fáze je jediný rámec, který bude v následujících fázích zcizen procesu, který ho využívá (předpokládejme dále pro jednoduchost, že rámec není sdílen).

uložení obsahu vytipovaného rámce do odkládacího prostoru

Prvým krokem vlastního procesu kradení je odložení obsahu zvoleného rámce do odkládacího prostoru v sekundární paměti (na disku). Pokud nemá zvolený rámec svůj obraz na odkládacím prostoru (nebyl ještě ukraden), je zde nalezen volný blok resp. je tento prostor příslušným způsobem zvětšen. Není-li na odkládacím prostoru místo a nelze-li jej zvětšit, musí být proces, pro nějž je paměť kradena, bezpodmínečně a bezprostředně ukončen.

Má-li však rámec svůj obraz v sekundární paměti (byl již dříve ukraden), je rámec odložen do bloku původního obrazu.

Uložení obsahu paměti na disk je však velmi pomalá operace (vzhledem k běžnému přístupu k operační paměti) a virtualizátor se jí snaží pokud možno vyhnout. Z tohoto důvodu neukládá ty rámce, jejichž obraz je již na disku a jež nebyly od doby posledního uložení změněny. Tento obraz může být uložen buď přímo na odkládacím zařízení (obsah byl již v minulosti uložen) nebo ve spustitelném souboru (mapovaný kód a inicializovaná data).

dirty bit

Pro zjištění, zda nebyla data změněna, slouží bitový příznak tzv. **dirty bit**, jenž je nastaven při každém zápisu do logické stránky. Pokud není nastaven, pak rámec s touto stránkou spojený nebyl od posledního nulování *dirty* bitu modifikován.

Virtualizátor proto po každém obnovení rámce z diskového obrazu nastaví *dirty bit* u logické stránky k níž rámec patří na nulu. Pokud pak má být obsah dané stránky opět uložen na disk, je testována hodnota *dirty bitu* u logické stránky, s níž je rámec spojen (tj. musí existovat efektivní representace inverzního mapování *rámec* →*logická stránka*).

V případě nevyhnutelného přesunu dat do sekundární paměti naplánuje proces uložení daného rámce na odkládací zařízení a poté se zablokuje (tj. opustí dobrovolně procesor a čeká pozastaven na dokončení této operace).

zneplatnění logické stránky, jež na kradený rámec odkazovala

Po uložení obsahu kradeného rámce na disk je proces odblokován a poté, co opět získá procesor, zneplatní logickou stránku, s níž byl rámec propojen a nastaví, že při jejím potenciálním výpadku má být obnovena z odkládacího prostoru (a do tabulky stránek vloží identifikaci příslušného bloku odkládacího prostoru).

Nyní již je tato stránka volná a může být propojena s logickou stránkou, jejíž výpadek celý proces kradení odstartoval.

Ke kradení stránky může dojít u libovolného typu výpadku stránky vyjma chybového výpadku při přístupu vně regionů. Nejsložitější situace nastává při tzv. *velkém výpadku stránky* (*major fault*), kdy proces musí nejdříve rámec ukrást (tj. uložit obsah rámce do odkládacího prostoru) a pak ho obnovit z (jiného bloku) odkládacího prostoru, neboť původní rámec byl již dříve ukraden. Speciálním případem je obnovení stránky ze spustitelného souboru v případě stránky uvnitř *read-only* regionu (např. kódového segmentu).

Velký výpadek stránky je díky dvěma přístupovým operacím k pomalé vnější paměti velmi pomalý a virtualizátor se proto snaží jeho vzniku zabránit a pokud již vznikne, tak se alespoň snaží jeho provedení urychlit (viz dále tzv. *zloděj stránek*). Pokud však požadavky na paměť výrazně převýší kapacitní možnosti rychlé operační paměti a procesy se tudíž přetahují o rámce, dochází k velkým výpadkům tak často, že režie správce paměti roste nade všechny meze, tj. systém nedělá nic jiného než horečně swapuje na disk a procesy stojí. Tento stav je označován jako **zahlcení** (*thrashing*).

zahlcení

2.3.5 strategie kradení stránek

Strategie kradení stránek určuje, která z obsazených (propojených) fyzických stránek je nejvhodnějším kandidátem pro ukradení a přidělení jiné logické stránce. Ideální strategií je volba rámce, k němuž již nebude přistupováno resp. (pokud žádný takový neexistuje) rámce, k němuž nebude přistupováno nejdelší dobu. Tato strategie je sice optimální, neboť volba jakéhokoliv jiného rámce je méně efektivní (dříve u něj dojde k opětovnému výpadku), avšak bohužel nerealizovatelná, neboť virtualizátor nemůže předvídat budoucí chování procesů.

Opačný extrém je ukradení náhodně vybrané stránky (pozice kradeného rámce je generována generátorem pseudonáhodné posloupnosti). Tato strategie je sice snadno realizovatelná, často však dochází k ukradení rámce, jenž je právě užíván (tj. některý z procesů z něj čte nebo do něj zapisuje resp. vykonává kód, jenž je v něm uložen). Je však třeba zdůraznit, že u některých procesů a za některých situací neexistuje výrazně lepší strategie a obecně platí, že skutečně užívané strategie se efektivitou blíží spíše náhodnému výběru, než ideálnímu stavu.

LRU strategie

Nejznámější a nejčastěji používanou strategii je **LRU** (*Least Recently Used*) tj. volba nejdéle nepoužívané stránky. Tato strategie se může na první pohled jevit jako pravý opak ideální strategie, neboť se může zdát že pravděpodobnost dalšího přístupu k rámci roste s délkou doby jeho nevyužití (tj. ukraden je vlastně ten nejméně vhodný rámec). Toto tvrzení však u většiny procesů neplatí (neplatilo by ani při rovnoměrném náhodném přístupu k paměti) a naopak platí spíše tvrzení opačné, tj. pravděpodobnost dalšího přístupu při delší nečinnosti klesá.

princip lokality

Platnost tohoto tvrzení pro většinu procesů (a tím i úspěšnost LRU) je dána tzv. zachováním **principu lokality**, jež je většině programů (aplikací) v různé míře vlastní.

pracovní množina stránek

Pro detailní vysvětlení pojmu principu lokality je nejdříve nutno zavést důležitý pojem **pracovní množina stránek** (procesu) v čase t — $\mathrm{W}_p(t)$. Pracovní množina stránek procesu je množina těch fyzických stránek procesu, k nimž proces přistupoval v čase $\langle t - \delta, t\rangle$, kde δ je časový interval, jež je dostatečně krátký vzhledem k době běhu programů (ta je výrazně menší než doba jejich existence) a výrazně delší než takt procesoru (dnes řádově

nanosekundy). Navíc je shora omezen intervalem časovače, jenž je elementárním časem při preemptivním přepínání procesů (dnes řádově v milisekundách). Nejčastěji se proto volí v řádu mikrosekund. Pracovní množina stránek procesu je definována pouze za běhu procesu (tj. v době kdy proces využívá procesor), nikoliv během jeho dobrovolného či nuceného čekání na procesor či jinou událost (tehdy je *de facto* vždy rovna nule). Pro jednodušší použití je proto vhodnější tuto veličinu definovat nad interním časem procesu, jenž plyne pouze za běhu procesu (tj. je pozastaven při čekání procesu) a jenž je navíc diskrétní (minimální čas je roven taktu procesoru).

Pracovní množina procesu se v čase mění, tj. přibývají do ní nové rámce a jiné z ní naopak mizí, a to vše v důsledku postupného vývoje procesu, a to jak v oblasti kódu (postupné vykonávání instrukcí) tak datové (používání různých sad lokálních proměnných či procházení rozsáhlých polí). Tento posun se však děje relativně pomalu, neboť procesy nepřistupují k paměti náhodně, ale spíše lineárně či dokonce cyklicky, tj. programy *zachovávají lokalitu* (pohybují se v obmezeném prostoru)

Například tok programu je převahou lineární s četnými a mnohdy relativně krátkými cykly, přístup k rozsáhlým datovým blokům se děje lineárně (pole se procházejí vzestupně s pevným často jednotkovým krokem), přístup k zásobníku je lineární (resp. částečně cyklický) svou podstatou.

Míru zachování lokálnosti lze kvantitativně vyjádřit například takto:

$$\frac{\text{card}\,(\mathrm{W_P}(t+\delta) \cap \mathrm{W_P}(t))}{\text{card}\,\mathrm{W_P}(t)}$$

Blíží-li se tato míra jedné, je *princip lokality* zachován (a LRU je v daný okamžik pro daný proces použitelnou strategií), pokud je naopak blízký nule, není LRU v daný okamžik příliš efektivní (efektivnější by zde byla i náhodná volba stránek). Pro většinu programů ve většině situací však platí spíše první možnost.

Bohužel i když je strategie LRU teoreticky proveditelná, vyžádala by si její podpora neúměrnou režii, neboť by od procesoru vyžadovala při každém přístupu k paměti zanechání časového razítka (*timestamp*) v tabulce stránek a v okamžiku kradení nalezení minimální hodnoty tohoto razítka mezi všemi *logickými stránkami* (resp. lze zvolit strukturu, jež sice usnadňuje nalezení minima, přináší však větší zatížení při vkládání stránky).

strategie pseudoLRU

Z tohoto důvodu se používají strategie často označované jako **strategie pseudoLRU**, které však mají s touto strategií jen málo společného. Většina těchto strategií totiž volí libovolnou stránku, která není v aktuálních pracovních množinách procesů (tj. nikoliv tu nejdéle nepoužitou). Pro tyto účely stačí, když operační systém vynuluje *acess* bity u všech logických stránek v alokovaných regionech a nechá aplikace po určitou krátkou dobu běžet (viz interval δ u definice pracovní množiny). Po této době jsou všechny stránky, jež jsou součástí pracovní množiny alespoň jednoho procesu, identifikovány jedničkovým *access* bitem. Naopak stránky, u nichž zůstal tento příznak nulový, jsou kandidáty na ukradení, neboť dle *principu lokality* je málo pravděpodobné, že budou v pracovní množině v několika následujících intervalech. Závěrečné rozhodnutí o tom, která stránka z doplňku sjednocení pracovních množin bude nakonec ukradena, může být libovolné např. náhodné nebo poziční (ukradena je první vhodná stránka při postupném procházení od pozice naposledy ukradené stránky).

Strategie typu LRU (resp. pseudoLRU) jsou nejčastěji *globálními strategiemi*, tj. proces může ukradnout stránku libovolnému procesu včetně sebe sama. Jinak řečeno o tom, zda bude stránka ukradena přímo nerozhoduje její příslušnost k některému z procesů,

zloděj stránek

ale pouze její příslušnost resp. spíše nepříslušnost ke sjednocení aktuálních pracovních množin (priorita procesu ji však ovlivňuje nepřímo). Globální charakter této strategie je ještě zvýrazněn, pokud je volba a vlastní kradení svěřeno zvláštnímu procesu, tzv. **zloději stránek** (*page stealer*, *pageout deamon*). Tento systémový proces je (u málo zatíženého systému) většinu času pozastaven (zablokován) a probouzí se pouze tehdy, pokud podíl volných rámců neklesne pod stanovenou dolní mez (řádově v jednotkách procent). Po probuzení, zjistí jaké fyzické stránky nejsou využívány (tj. jsou mimo pracovní množiny), a potom je začne krást, nikoliv však pro sebe, ale pro ostatní procesy (tj. nepřipojuje je ke svým logickým stránkám, ale nechává je volné). Navíc nekrade jen jednu stránku, ale v kradení pokračuje dokud nedosáhne jisté horní meze (typicky kolem 10 %).

Jaké výhody má použití *zloděje stránek* oproti strategii „*ukradni si sám*“? Za prvé to urychluje ukládání odcizených stránek, neboť zloděj stránek ukládá najednou několik stovek stran a může zvolit jejich pořadí tak, aby plně využil hardwarového mechanismu jejich ukládání (tj. přednostně ukládá stránky, jejichž obrazy leží v odkládacím prostoru na jediné stopě či alespoň v těsné blízkosti). Za druhé zloděj stránek brání nestabilitě, která může vzniknout, pokud se více procesů snaží získat jedinou stránku (tj. procesu je ukradená stránka znovu ukradena a to dříve než ji stačí použít). Tento efekt zvyšuje režii systému, což ve svém důsledku může vést k předčasnému zahlcení systému (v okamžiku, kdy již nejsou volné rámce, není systém ještě zdaleka plně vytížen).

lokální strategii FIFO

Strategie LRU není jedinou v praxi používanou strategií volby odkládaných rámců. Například jádro Windows NT užívá **strategie FIFO**. V této strategii je zvolen ten rámec, jenž je nejdelší dobu využíván (tj. je nejdéle propojen s logickou stránkou) a to vždy rámec vlastněný týmž procesem (včetně rámců sdílených). Jinak řečeno, každý nově propojený rámec je zařazen na konec fronty (struktury typu FIFO), z přední části jsou naopak vybírány stránky, jež budou odebrány (ukradeny). Úspěšnost resp. akceptovatelnost této strategie není založena na principu lokality (např. může být ukradena stránka, jež je jistě v aktuální pracovní množině), ale na výhodnosti co nejdelšího držení rámce (rámec je ukládán až v okamžiku, kdy již není zbytí). I když je tato strategie u většiny typických aplikací méně efektivní než klasické LRU, je plně srovnatelná se skutečně použitelnými *pseudoLRU* strategiemi, a to při shodné režii.

Lokální charakter, jenž je pro FIFO strategii typický, přináší oproti globálním strategiím jisté výhody, a to především v oblasti ochrany důvěrných údajů (proces si nemůže namapovat rámec s údaji užívanými jiným procesem), vynucuje si však relativně složitou správu velikosti pracovních množin. Důvodem je skutečnost, že velikost pracovních množin se může navzdory převládajícímu principu lokality výrazně měnit. Například pokud proces dynamicky naalokuje region paměti a začne jej masívně využívat (například plní jej iniciálními nenulovými hodnotami), rozroste se jeho pracovní množina o rámce, na něž je tento region nakonec namapován (rámce se přidělují až v okamžiku prvního přístupu). Naopak pokud je proces dlouho pozastaven, např. čekáním na stisk klávesy, klesá velikost jeho pracovní množiny skokem k nule.

správce pracovních množin

Pokud by byla každému novému procesu přidělena neměnná množina rámců, docházelo by občas u procesu k překotnému kradení stránek (i když by byla operační paměť relativně volná) nebo naopak by existovaly nevyužívané rámce (tj. rámce, jež jsou propojeny s logickým adresovým prostorem, ale nejsou resp. dokonce dlouhodobě nebyly v pracovní množině procesů) navzdory akutnímu nedostatku rámců u jiných procesů. Proto musí existovat **správce pracovních množin** (ve formě rutiny či spíše systémového procesu), jenž kontroluje míru využití přidělených rámců a dle ní přiděluje či odebírá procesům rámce (jsa přirozeně omezen dostupnou velikostí operační paměti). Rámce, jež jsou procesům odebírány, jsou nulovány a ihned přidělovány jiným procesům (odebírání se

děje pouze při nedostatku operační paměti). Systém může dokonce nabízet systémovou službu, díky níž může aplikace (programátor) signalizovat očekávanou změnu velikosti pracovní množiny předem, což umožní správci lépe na tuto změnu reagovat.

2.3.6 sdílená paměť

sdílená paměť

V předchozí části kapitoly o virtualizaci paměti jsme (až na drobné výjimky) předpokládali, že každý region paměti je vlastněn pouze jedním procesem a že každý rámec operační paměti je propojen nejvýše s jednou logickou stránkou. Virtualizace paměti však umožňuje snadnou realizaci tzv. **sdílených pamětí**, což je fyzický paměťový prostor (operační paměť + odkládací prostor) přístupný z několika logických adresových prostorů, tj. i z několika různých procesů či dokonce aplikací. Pokud je tato sdílená paměť přístupná pro čtení a zápis, může sloužit pro velmi rychlou a rozsáhlou komunikaci mezi procesy, jestliže je určena pouze ke čtení či vykonávání, může výrazně snížit nároky na fyzickou paměť.

Základní implementace oblasti sdílené paměti spočívá ve sdílení obsahu souvislé části tabulky stránek, tj. položky tabulky stránek příslušné danému bloku obsahují v obou logických adresových prostorech shodné údaje. Navíc jsou sdíleny některé údaje i v dalších pomocných datových strukturách (např. tabulce bloků odkládacího souboru resp. v tabulce rámců).

Pokud je logická stránka uvnitř sdíleného bloku platná, pak její záznam ve stránkovacích tabulkách obou procesů obsahuje odkaz na stejnou fyzickou stránku. U neplatných stránek je sdílen stav a v případě stránek, jejichž obsah je odložen na odkládacím zařízení, je sdílen i tento obsah (tj. sdílen je index bloku v odkládacím prostoru).

Zajímavým důsledkem této implementace je skutečnost, že sdílený blok nemusí začínat v jednotlivých logických adresových prostorech na stejných logických adresách. To jest zapíše-li jeden proces do paměti na adrese X, změní se u druhého hodnota paměťového místa na zcela jiné adrese Y (u druhého procesu může vést přístup k adrese X dokonce k porušení ochrany paměti, neboť na této adrese nemusí ležet žádný alokovaný region). Tato vlastnost, i když může některé aplikační programátory překvapit, je nezbytná, neboť nucené použití jediné bázové adresy by vnášelo nepříjemnou závislost mezi procesy a vedlo by k nepříjemným kolizím (jak řešit požadavek na přístup ke dvěma různým sdíleným pamětem s překrývajícím se adresovým prostorem).

Sdílení fyzické paměti se ve většině případů řeší jako sdílení celých regionů (tj. sdílení logického a abstraktnějšího prostředku), což zjednodušuje návrh a implementaci a je pohodlnější i pro aplikační programátory. Na úrovni jádra OS se tento přístup projevuje použitím jen jediné datové struktury representující paměťový region. Tato struktura obsahuje velikost bloku, jeho přístupová práva (RWX resp. RW) a popis iniciálního stavu panenských stránek. Popis regionu neobsahuje bázovou adresu a dokonce ani příslušný sdílený úsek tabulky stránek (sdílen je obsah nikoliv vlastní úsek tabulky). Tento region mohou otvírat jednotlivé procesy (pokud mají dostatečná práva) a především si jej mohou připojovat do svého logického adresového prostoru. Při připojení je stanovena bázová adresa daného regionu v daném logickém adresovém prostoru (musí být zvolena tak aby nedošlo k překryvu s jiným regionem a musí být zarovnána na hranici stránky) a položky tabulky stránek odpovídající danému úseku logického adresového prostoru (od bázové adresy po adresu, jež se získá přičtením velikosti k bázové adrese a zaokrouhlením na hranici stránek) jsou nastaveny tak, aby byly v souladu s nastavením v popisu regionu (typ panenských stránek, ochrana RWX). Navíc u stránek, jež již

nejsou panenské, je nastaveno propojení na fyzickou paměť (rámec či blok odkládacího prostoru) a to podle stavu daného úseku v tabulkách stránek procesů, jež si již sdílený region již připojily.

Správa *sdílené paměti* je určena dvěma principy – principem konzistence (všechny procesy musí mít v každém okamžiku stejný obraz paměti) a principem lenivého vykonávání (vše je provedeno až tehdy, když již není zbytí). Například pokud dojde k ukradení rámce, jež je spojen se sdíleným regionem, musí být zneplatněny odpovídající logické stránky ve všech dotčených adresových prostorech. Naopak pokud dojde ke zplatnění stránky v rámci výpadku stránky způsobeném jedním z procesů, stačí když je zplatněna jen ta jediná výpadkem dotčená logická stránka. Pokud totiž dojde k výpadkům u dalších odpovídajících stran, může systém detekovat, že obraz této stránky již v operační paměti existuje (prostřednictvím jedinečného indexu bloku odkládacího prostoru, jenž je zaznamenán v tabulce rámců) a následně zajistit její propojení.

Hlavním využitím mechanismu sdílených regionů je sdílení programového kódu mezi všemi instancemi určité aplikace. Každá instance je procesem a tudíž i vlastníkem jedinečného logického adresového prostoru, běží však podle kódu, jenž je neměnný a především shodný ve všech instancích dané aplikace. Kódový region tak může být regionem sdíleným, jenž si připojí všechny instance daného programu na začátek svého logického adresového prostoru. I když je však region sdílený, nemusí vždy docházet ke sdílení vlastní operační paměti, neboť různé instance aplikace se nacházejí na různých místech programu a tudíž kódové části jejich pracovních množin mohou být disjunktní. V každém případě se však šetří odkládací prostor a urychluje se běh aplikací (především startování dalších instancí jednotlivých aplikací).

dynamicky linkovaná knihovna

Použití sdíleného kódu je ve většině současných operačních systémů dále rozvinut podporou tzv. **dynamicky linkovaných knihoven** (ve světě *Windows* označovány zkratkou DLL ve světě *Unixu* spíše zkratkou SO [*shared object*]). Tyto knihovny nejsou ke spustitelnému souboru připojeny již při překladu (a nejsou tudíž součástí jeho kódového segmentu), ale jsou na požádání procesu připojeny až při jeho spuštění (či dokonce až těsně před použitím). Navíc jsou sdíleny všemi aplikacemi, které je v daném okamžiku potřebují (tj. jejich stránky jsou ve fyzické paměti nejvýše jednou). Jediným problémem je skutečnost, že jejich umístění v logickém adresovém prostoru není pevné, tj. v různých instancích aplikace mohou ležet na různých místech adresových prostorů. Proto při každém přístupu k objektu DLL knihovny (nejčastěji je to volání procedury či funkce) musí být použita nepřímá adresa, jež je za běhu spočítána z bázové adresy DLL knihovny a offsetu (relativní adresy) objektu (např. procedury) v knihovně. Toto volání je však u většiny procesorů jen nepatrně pomalejší než přímé volání užívané ve statickém kódu.

> Význam dynamicky linkovaných knihoven u současných operačních systémů ukazuje zkrácená mapa logického adresového prostoru u editoru Lyx běžícím pod OS Linux (v tomto editoru právě edituji tento text). Legenda ke sloupcům: počáteční (bázová) adresa regionu, koncová adresa, velikost v KB, podíl regionu na uživatelské části logického adresového prostoru (v Linuxu na platformě Intel má velikost 3GB), přístupová práva (bloky s právy `r-x` obsahují kód, bloky s právy `rw-` běžná data, a práva `r--` přísluší *read-only* mapovaným datovým souborům) a nakonec je uveden soubor, který je do daného regionu namapován (pokud takový existuje). Cesta k souboru je v některých případech zkrácena, zkrácení je označeno elipsou (trojtečkou).

```
p.adresa-k.adresa    velikost    podíl   PP    soubor
------------------------------------------------------------------------
00000000-08048000 =  131360 KB ( 4.176%)       --- FREE ----
```

```
08048000-0846b000 =    4236 KB ( 0.135%) r-x  /usr/bin/lyx
0846b000-08491000 =     152 KB ( 0.005%) rw-  /usr/bin/lyx
08491000-086fc000 =    2476 KB ( 0.079%) rwx
086fc000-40000000 =  910352 KB (28.939%)       --- FREE ---
40000000-40013000 =      76 KB ( 0.002%) r-x  /lib/ld-2.2.5.so
40013000-40014000 =       4 KB ( 0.000%) rw-  /lib/ld-2.2.5.so
40014000-40015000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_MESSAGES...
40015000-40016000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_TIME
40016000-40017000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_IDENTIFICATION
40017000-40018000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_MEASUREMENT
40018000-40019000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_TELEPHONE
40019000-4001a000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_ADDRESS
4001a000-4001b000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_NAME
4001b000-4001c000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_PAPER
4001c000-4001d000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_MONETARY
4001d000-40023000 =      24 KB ( 0.001%) r--  .../cs_CZ/LC_COLLATE
40023000-40024000 =       4 KB ( 0.000%) r--  .../cs_CZ/LC_NUMERIC
40024000-40026000 =       8 KB ( 0.000%) r-x  .../common/xlcDef.so.2
40026000-40027000 =       4 KB ( 0.000%) rw-  .../common/xlcDef.so.2
40027000-40028000 =       4 KB ( 0.000%)       --- FREE ---
40028000-4002e000 =      24 KB ( 0.001%) r--  .../gconv-modules.cache
4002e000-40645000 =    6236 KB ( 0.198%) r-x  .../libqt-mt.so.3.0.3
40645000-40684000 =     252 KB ( 0.008%) rw-  .../libqt-mt.so.3.0.3
40684000-4068a000 =      24 KB ( 0.001%) rw-
4068a000-406a0000 =      88 KB ( 0.003%) r-x  .../libpspell.so.4.0.3
406a0000-406a4000 =      16 KB ( 0.001%) rw-  .../libpspell.so.4.0.3
406a4000-406aa000 =      24 KB ( 0.001%) r-x  .../libltdl.so.3.1.0
406aa000-406ab000 =       4 KB ( 0.000%) rw-  .../libltdl.so.3.1.0
406ab000-406ad000 =       8 KB ( 0.000%) r-x  /lib/libdl-2.2.5.so
406ad000-406ae000 =       4 KB ( 0.000%) rw-  /lib/libdl-2.2.5.so
406ae000-406af000 =       4 KB ( 0.000%) r-x  .../libpspell-modules.so.1.0.1
406af000-406b0000 =       4 KB ( 0.000%) rw-  .../libpspell-modules.so.1.0.1

--- zde je vynecháno cca 80 řádků s dalšími knihovnami a namapovanými daty

42000000-4212c000 =    1200 KB ( 0.038%) r-x  /lib/i686/libc-2.2.5.so
4212c000-42131000 =      20 KB ( 0.001%) rw-  /lib/i686/libc-2.2.5.so
42131000-42135000 =      16 KB ( 0.001%) rw-
42135000-bffeb000 = 2063064 KB (65.583%)       --- FREE ---
bffeb000-c0000000 =      84 KB ( 0.003%) rwx
```

Jak lze snadno odvodit, za prvním prázdným blokem (ten je vynucen použitým formátem spustitelných souborů) následuje kódový region o velikosti cca 4 MB (je namapován na kódový segment spustitelného souboru, tj. při výpadku je z něj obnoven). Dalším je klasický datový region, jenž je namapován na část spustitelného souboru, obsahují inicializovaná (a možná i neinicializovaná) statická data (pouze 152 KB). Zbytek dat je uložen ve třetím bloku, jenž obsahuje především dynamicky alokovanou paměť (tj. tzv. hromadu či *heap*) a není tudíž překvapením, že je výrazně větší (přes 2 MB). Pak následuje volný prostor do něhož může expandovat dynamicky alokovaná paměť (hromada) o velikosti téměř 900 MB. Další dva blo-

ky tvoří první dynamicky linkovaná knihovna, jejíž kódová část výrazně převažuje nad datovou (datová část slouží k úschově stavových a konfiguračních dat knihovny). Na datovém bloku lze také pozorovat minimální velikost regionu, jež je rovna velikosti stránky (zde 4 KB).

Zajímavou oblast tvoří namapované soubory s daty tzv. *locales* (unixovská representace národních a jazykových nastavení, zde nikoliv překvapivě pro češtinu v České republice). Tyto soubory jsou běžné datové soubory, které jsou stejně jako kód mapovány do adresového prostoru procesu (tj. při výpadku stránky jsou obnoveny z odpovídajícího čtyřkilobytového segmentu daného souboru). Dále již následují další dynamické knihovny, z nichž si pozornost zaslouží knihovna QT (*libqt-mt.so.3.0.3.so*), což je GUI framework a knihovna *libc* (*libc-2.2.5.so*), obsahující klasické API *Unixu* (resp. přesněji POSIXu). Téměř na konci seznamu je volný prostor (skoro 2 G) do něhož mohou expandovat jak sdílené knihovny (proces je může zavádět dynamicky na požádání) tak zásobník. A právě zásobník o velikosti pouhých 84 KB je až na samém konci výpisu. Zbytek adresového prostoru (adresy c0000000-ffffffff = 1 GB) zaujímá jádro Linuxu.

aplikační rozhraní sdílené paměti a mapování souborů

Mechanismus sdílené paměti a mapování souborů do logického adresového prostoru může být aplikačními programátory využíván i přímo prostřednictvím specializovaných služeb systému (ty jsou dostupné jak v posixových systémech tak ve Win32).

Sdílená paměť je použitelná pro sdílení informací mezi několika kooperujícími procesy resp. pro rychlý přenos informací mezi procesy (sdílená paměť je nejrychlejším komunikačním prostředkem). Použití explicitně vyžádané sdílené paměti je transparentní (tj. přístup se děje stejnými prostředky jako přístup k dynamicky alokované paměti, tj. na úrovni vyšších programovacích jazyků prostřednictvím ukazatelů), avšak proces alokace je poněkud složitější.

Prvním krokem je vytvoření vlastního prostředku, tj. regionu s požadovanou velikostí a přístupovými právy (nejčastěji RW) a to jedním z procesů, jež ho budou následně využívat. Ostatní procesy dají najevo svůj zájem na využívání sdílené paměti otevřením této již existující paměti, užívajíce při tom sdíleného identifikátoru (čísla resp. řetězce). Protože však není snadné zajistit nutnou synchronizaci (paměť musí být nejdříve vytvořena a teprve pak ji lze otevřít) mohou všechny kooperující procesy volat identickou službu a systém sám rozhodne, zda se jedná o vytvoření prostředku nebo pouze o otevření.

Po provedení tohoto kroku získá každý zúčastněný proces deskriptor (handler) regionu, avšak nemůže jej využívat, neboť region není mapován do jeho adresového prostoru (tj. záznamy o rámcích regionu nejsou součástí jeho tabulky stránek). Vlastní mapování (tj. příslušná modifikace tabulek stránek) se děje v poslední fázi voláním další služby operačního systému, jejíž parametrem je deskriptor sdílené paměti a volitelně i bázová adresa regionu v LAP procesu (tato možnost není příliš často užívána, neboť aplikační programátor nezná a prakticky ani nemůže znát aktuální obsazení LAP).

Vícefázové je i uvolňování sdílené paměti (odmapování, uzavření, destruování). U některých aplikačních knihoven jsou všechny fáze mapování resp. uvolňování sdílené paměti spojeny do jediného volání (např. do konstruktoru resp. destruktor.

Mapování kódové a datové oblasti spustitelných souborů je integrální součástí spuštění procesu u systému se stránkováním na žádost (včetně mapování dynamicky zaváděných knihoven). Stejného mechanismu však lze využít i pro zpřístupnění běžných datových

souborů. Hlavní výhodou je zjednodušení přístupu k souboru, neboť lze užít ukazatelů resp. operací pro práci s dynamicky alokovaným polem (tj. indexace, v C a C++ pointerové aritmetiky, řetězcových operací apod.), oproti složitějším IO funkcím (v jazyce C např. funkcí *fread*, *fwrite*, *fseek* apod.). Mezi hlavní nevýhody patří problematická změna velikosti souboru (především jeho prodlužování) a menší efektivita oproti specializovaným IO službám (pouze u některých implementací a i zde jen v řádu procent). Problémem může být i maximální velikost mapovatelného souboru, neboť ta je dána velikostí největšího volného bloku v logickém adresovém prostoru (u 32bitových systémů většinou nepřesahuje 3 GB, u 64bitových je však téměř neomezená). Toto omezení lze alespoň částečně obejít mapováním dílčích úseků souborů.

Mechanismus mapování souborů lze užít i pro sdílení paměti, neboť jednotlivý soubor lze namapovat do vícero logických adresových prostorů. Jediným rozdílem oproti klasickému sdílenému souboru je použití jiného odkládacího souboru (odkládacím souborem je namapovaný soubor a nikoliv standardní swapovací zařízení). Výhodou může být persistentní charakter tohoto souboru (tj. uložená data zůstanou zachována i po skončení mapování a dokonce i po ukončení kooperujících souborů). Některé systémy umožňují i mapování úseků standardního odkládacího prostoru, to jest úplnou ekvivalenci s klasickým mechanismem sdílené paměti (liší se jen použité názvosloví). Některé systémy proto nabízejí mechanismus sdílené paměti pod hlavičkou mapování souborů (Win32 výhradně a u POSIXu je to dnes převažující přístup).

dočasně sdílená paměť (copy-on-write)

Kromě klasické sdílené paměti podporují některé operační systémy i tzv. dočasně sdílenou paměť. Tu tvoří minimálně dva paměťové regiony, které zpočátku odkazují na stejnou fyzickou paměť (tj. jejich stránky odkazují na stejné rámce či bloky odkládacího prostoru). Pokud procesy provádějí toliko čtení situace se nemění (pouze se přirozeně přesouvá obsah mezi operační pamětí a odkládacím prostorem). Pokud ovšem jeden z procesů užívajících dočasně sdílenou paměť pokusí do paměti zapsat, je vyvolána procesorová výjimka a v jejím rámci dojde k rozštěpení fyzické paměti. Proces způsobivší výjimku získá svůj vlastní soukromý rámec, do něhož je zkopírován obsah původního (sdíleného) rámce. Po návratu z výjimky proces zápis dokončí, ale tentokrát již do vlastní a nesdílené paměti (tj. pro ostatní procesy nad dočasně sdílenou nedojde ke změně). Tato strategie se označuje jako **copy-on-write** (*kopírování při zápisu*).

copy-on--write

Jaký smysl má toto dočasné sdílení ukončené rozštěpením paměti při prvním zápise? Předpokládejme, že proces chce získat vlastní kopii rozsáhlého regionu jiného procesu. Při běžné implementaci by musel alokovat nový region odpovídající velikosti a následně by byl nucen zkopírovat celý obsah původního regionu (pokud by nebyl původní region sdílený, pouze prostřednictvím volání specializované služby jádra, neboť nemá přístup do fyzické paměti jiného procesu). Při tomto kopírování by nutně docházelo k alokování nových rámců a to jak pro nový region (zde pro každou kopírovanou stránku) tak mnohdy i u regionu starého (pokud je obsah některé jeho stránky odložen ve swapovacím prostoru) neboť kopírování se děje pouze z operační do operační paměti. Takovéto kopírování je velmi pomalé a především náročné na fyzické prostředky (rámce). Navíc je však zbytečné, neboť fyzické rozdělení (duplikování) není nutné, pokud budou oba procesy z paměti pouze číst (při čtení se obsah paměti nemění). Teprve při prvém zápisu do jednoho z regionů musí dojít k rozdělení, jež však může být omezeno na jedinou stránku. Opět se tu setkáváme s lenivým vykonáváním (vše udělej, až to bude skutečně třeba a pokud možno jen v té nejnutnější míře), jež je pro stránkování na žádost typické. Navíc

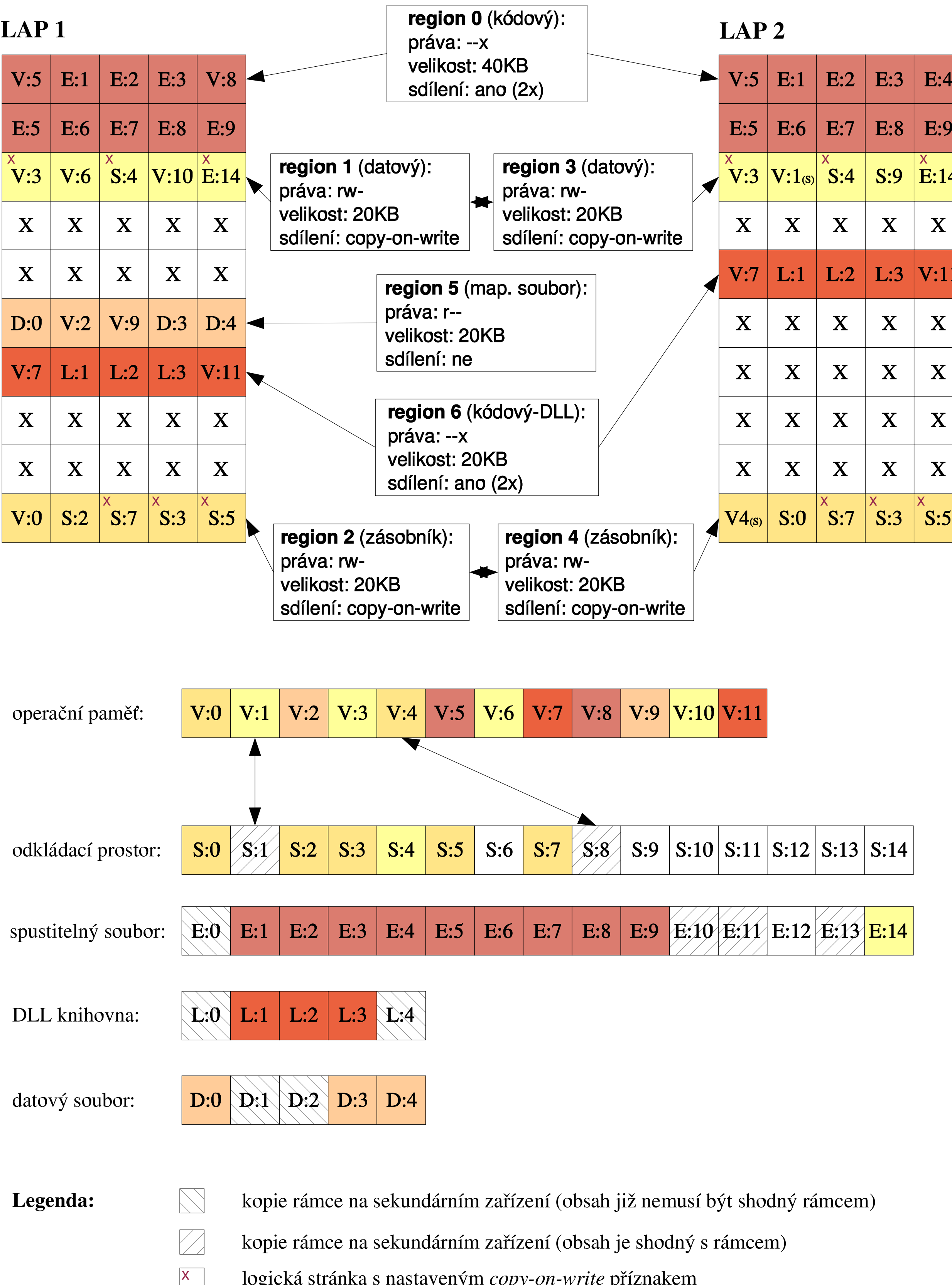
je tato strategie pro oba procesy plně transparentní, tj. oba procesy nemusí o dočasném sdílení nic vědět (a aplikační programátoři proto také většinou nic nevědí).

odložené kopírování při zápise

Odložené kopírování při zápise (*copy-on-write*, COW) je velmi výhodné především v případě, pokud oba procesy z paměti pouze čtou nebo pokud jeden z procesů region v krátkém časovém horizontu opět uvolní. Z tohoto důvodu je tento mechanismus nesmírně důležitý pro unixovské systémy, kde se nové procesy vytvářejí duplikací svých rodičů. To jest například, pokud unixovský shell potřebuje vyvolat nový proces jako reakci na příkaz uživatele, musí se nejdříve rozdvojit ve dva téměř identické procesy (oba běží podle programu původního shellu a užívají stejných dat). Zatímco však jeden z nich (ten původní) zůstává shellem, zavolá druhý téměř bezprostředně po rozdvojení službu *exec*, která uvolní všechny původní paměťové regiony a alokuje nové podle požadavků nového programu aplikace a následně začne tento nový program vykonávat. Pokud by neexistovala strategie *copy-on-write*, muselo by při rozdvojení procesu dojít ke kopírování celého datového a zásobníkového regionu, a to vše zcela zbytečně, neboť nově zkopírovaná paměť je bezprostředně poté uvolněna (často bez toho, že by některý z procesů do své kopie stačil cokoliv zapsat).

Implementace mechanismu *copy-on-write* je relativně snadná. Stačí pouze u každé logické stránky spojené s *copy-on-write* rámcem nastavit zákaz zápisu (podobně jako při běžné ochraně stránky) spolu s bitovým příznakem *copy-on-write* (ten není spravován procesorem, ale rutinami virtualizátoru). Oba příznaky jsou součástí příslušné položky tabulky stránek. Zákaz zápisu kontrolovaný procesorem při každém zápisu zde slouží pouze k vyvolání výjimky, v jejíž obsluze je kontrolován *copy-on-write* bit a pokud je nastaven, může dojít k duplikaci stránky (pokud nastaven není jedná se o porušení ochrany paměti). Nově alokovaný rámec je běžnou stránkou (tj. nemá nastaven žádný z výše uvedených příznaků), původní rámec si zachová příznak *copy-on-write* pouze tehdy, pokud zůstává dočasně sdílenou stránkou (tj. jen tehdy, existuje-li třetí proces, jenž ještě nevlastní svou kopii). Oba příznaky však mohou být nastaveny i u neplatné stránky odkazující na blok odkládacího prostoru (stránka může být odcizena před i po duplikaci regionu). V tomto případě je obsah nové stránky získán z odkládacího prostoru, původní stránka však zůstane na odkládacím zařízení (a logická stránka v LAP druhého procesu zůstane neplatná).

Problematika sdílená paměti je shrnuta na obrázku, který ukazuje ve značném zjednodušení logické adresové prostory dvou původně identických procesů vzniklých voláním služby *fork*. Každý z nich však po rozdvojení běžel nezávisle a vykonával jiný algoritmus. Pokuste se ze stavu paměti odvodit životní osudy obou procesů (pomůckou je i vzestupné číslování jednotlivých regionů).



LAP 1
V:5 E:1 E:2 E:3 V:8
E:5 E:6 E:7 E:8 E:9
V:3 V:6 S:4 V:10 E:14
X X X X X
X X X X X
D:0 V:2 V:9 D:3 D:4
V:7 L:1 L:2 L:3 V:11
X X X X X
X X X X X
V:0 S:2 S:7 S:3 S:5
LAP 2
V:5 E:1 E:2 E:3 E:4
E:5 E:6 E:7 E:8 E:9
V:3 V:1(S) S:4 S:9 E:14
X X X X X
V:7 L:1 L:2 L:3 V:11
X X X X X
X X X X X
X X X X X
X X X X X
V4(S) S:0 S:7 S:3 S:5
region 0 (kódový):
práva: --x
velikost: 40KB
sdílení: ano (2x)
region 1 (datový):
práva: rw-
velikost: 20KB
sdílení: copy-on-write
region 3 (datový):
práva: rw-
velikost: 20KB
sdílení: copy-on-write
region 5 (map. soubor):
práva: r--
velikost: 20KB
sdílení: ne
region 6 (kódový-DLL):
práva: --x
velikost: 20KB
sdílení: ano (2x)
region 2 (zásobník):
práva: rw-
velikost: 20KB
sdílení: copy-on-write
region 4 (zásobník):
práva: rw-
velikost: 20KB
sdílení: copy-on-write
operační paměť:
V:0 V:1 V:2 V:3 V:4 V:5 V:6 V:7 V:8 V:9 V:10 V:11
odkládací prostor:
S:0 S:1 S:2 S:3 S:4 S:5 S:6 S:7 S:8 S:9 S:10 S:11 S:12 S:13 S:14
spustitelný soubor:
E:0 E:1 E:2 E:3 E:4 E:5 E:6 E:7 E:8 E:9 E:10 E:11 E:12 E:13 E:14
DLL knihovna:
L:0 L:1 L:2 L:3 L:4
datový soubor:
D:0 D:1 D:2 D:3 D:4
Legenda:
kopie rámce na sekundárním zařízení (obsah již nemusí být shodný rámcem)
kopie rámce na sekundárním zařízení (obsah je shodný s rámcem)
logická stránka s nastaveným copy-on-write příznakem

PŘEČTĚTE SI

Alternativní pohled na správu paměti (především) nabízí sedmá a osmá kapitola knihy *Operating Systems Internals and Design Principles* (Stallings) [12]_

7. Memory management

8. Virtual memory

OTÁZKY

1. Co je lineární adresa?
2. Jak velký adresový prostor je schopna adresovat 32, 48 resp. 64 bitová jednotka? Vyjádřete v GiB, TiB a vyšších jednotkách.
3. Co obsahuje relokační tabulka?
4. Jak lze optimálně rozvrhnout statické bloky?
5. Kolik bloků může být maximálně zceleno v jednom kroku?
6. Jak OS může reagovat na výpadek stránky?
7. Proč je zásobníkový region běžně na konci uživatelského adresového prostoru?
8. U jakých regionů paměti se využívá mechanismu mapování souborů (sdíleného či nesdíleného)?
9. Jaké stránky krade zloděj stránky (logické či fyzické)?
10. Kdy dochází k malému výpadku stránky?
11. Jaká datová struktura narušuje více princip lokality: souvislé pole hodnot nebo hashovací tabulka?
12. Jaké jsou výhody resp. nevýhody dynamicky linkovaných knihoven?
13. Pro jaké operace lze využít mapování souborů na uživatelské úrovni?
14. Co je strategie lenivého přístupu? Kdy je užitečná a kdy nikoliv? (diskutujte na příkladu domácích prací a COW)

OTÁZKY K ZAMYŠLENÍ

1. Mechanismus odkládání (statických) bloků do sekundární paměti může uložený obsah bloku obnovit i do jiného bloku, než byl ten původní (tj. obsah se v adresovém prostoru posune). Jaké výhody má tato strategie proti jednoduššímu obnovení na původním místě? Jakou technickou podporu na úrovni procesoru vyžaduje?
2. Běžná velikost stránky se pohybuje okolo jednotek KB. Zdůvodněte proč je nevýhodné užívat menší resp. větší stránky.
3. Chování paměťového subsystému je závislé na zatížení paměti? Jak je možno definovat míru zatížení paměti? (nápověda: sjednocení W_P). K jakým typům výpadků dochází při různém zatížení? Dochází k nim i při malém zatížení?
4. Uveďte případy programovacích technik či přístupů, jež narušují princip lokality a označte hlavní negativní důsledky narušení tohoto principu.
5. Navrhněte efektivní mechanismus přesunu celého regionu uvnitř jediného datového prostoru (data nemusí být fyzicky kopírována)

6. Některé procesory neposkytují *dirty bit*, navrhněte jeho alternativní softwarovou implementaci.
7. Dalším možným mechanismem virtualizace paměti je kromě stránkování i tzv. segmentace. Segmenty mohou mít na rozdíl od stránek různou velikost (od řádu KB po GB) a jejich použití si tudíž vynucuje složené (nelineární) logické adresy (segment+offset), ale jinak se mechanismus segmentace od stránkování příliš neliší (např. logické segmenty mají příznak platnosti). V konkurenci se stránkováním však segmentace neobstála, zkuste se zamyslet proč.
8. Popis správy fyzické paměti vychází jediného adresového prostoru von Neumanovské architektury? Jak se změní u počítačů s architekturou harvardského typu?
9. Jak funguje buddy systém pro alokaci statických bloků?
10. Jak by měl systém zareagovat v případě že k přístupu k neplatné stránce dojde v režimu jádra? Zohledněte původ a funkci neplatné adresy?
11. Většina moderních procesorů podporuje tzv. NX bit (not-execute), jehož nastavení brání vykonání stránky jako programu. Proč byl tento bit zaveden?
12. Jak probíhá nulování rámců? Jaké nebezpečí přináší v systémech, které využívají procesorové vyrovnávací paměti (cache)?
13. Zamyslete se nad kradením stránek jako nad kriminálním případem. Co bylo ukradeno, jakým způsobem, kdo byl poškozen, kdo a proč kradl, je kradení stránek etické, apod.
14. Diskutujte použití flash pamětí (SSD disků) jako odkládacího prostoru.
15. Proč je strategie FIFO běžně implementována jako lokální a LRU jako globální?
16. Uveďte příklady typů datových souborů, pro něž je vhodné použití mapování resp. pro něž vhodné není (např. logovací soubory)

ÚKOLY

1. Zobrazte a diskutujte adresový prostor různých procesů v Linuxu.
2. Vytvořte skript zobrazující velikost globální pracovní množiny v průběhu času (/proc/meminfo)
3. Zobrazte využití fyzické paměti / logických adresových prostorů jednotlivými procesy (ps, /proc/pid).

3 Správa procesů

CÍLE KAPITOLY

Hlavním cílem této kapitoly je mechanismus zdánlivého souběhu více procesů na jediném procesoru tzv. *preemptivní multitasking*.

Dosažení dokonalého multitaskingu není jednoduché a mnoha operačním systémům trvalo desetiletí, než toho dosáhli. Z tohoto důvodu se pozornost musí nejprve zaměřit na tzv. kontext procesu a mechanismus jeho ukládání (či spíše neukládání).

Poté co si dokážeme, jak je těžké dokonalého multitaskingu dosáhnout a jak obtížně je pochopitelný pro převážně neparalelně myslícího člověka, zjistíme, že ve skutečnosti systém využívá plného multitaskingu jen výjimečně a ve většině případů se omezuje na tzv. kooperativní multitasking. Jedním z cílů je tak pochopení klíčového významu kooperace (nejen) ve světě procesů.

Závěrem se podíváme na multitasking z pohledu jednotlivého procesu. Cílem je využití této perspektivy při programování aplikací.

3.1 Proces a jeho kontext

Procesy patří mezi základní prostředky všech operačních systémů (bez výjimky).sedmá a osmá Veškerá aktivita celého počítačového systému je soustředěna do procesů.

Proces je možno popisovat z mnoha různých pohledů, jež se navzájem doplňují, přičemž mezi ty základní patří:

genetický — proces je instancí algoritmu nebo programu. Program je nejčastěji representován souborem, jenž obsahuje strojový kód aplikace a její iniciální data (dále jen ***spustitelný soubor***), ale existují i výjimky neboť program nemusí být representován jediným souborem (např. procesy jádra) a ani nemusí obsahovat strojový kód (interpretované skripty podporované na úrovni OS)

dynamický — proces je postupným vykonáváním instrukcí (*jediným smyslem existence procesu je vykonávání instrukcí*). Proces během vykonávání instrukcí mění stavy prostředků (primárně operační paměti) a může ovlivňovat i okolí výpočetního systému (např. uživatele). Sám je však prostředky a okolím (zpětně) ovlivňován (pro *procesy* je typická interakce s okolím, a to od prostředků, přes ostatní procesy, až po vnější systémy).

systémový — proces je logickým prostředkem operačního systému, jenž je vytvářen i destruován prostřednictvím rutin (jádra) operačního systému a je operačním systémem podporován i během své existence (např. jsou udržovány informace o jím

užívaných prostředcích či o jeho základním stavu). Důležitou vlastností procesu z tohoto pohledu je jeho jednoznačná identifikace mezi všemi ostatními procesy, a to jak současnými tak minulými i budoucími (v praxi stačí jednoznačnost v dostatečně dlouhém, ale omezeném časovém intervalu).

Dynamický pohled na proces předpokládá změnu stavu po provedení každé instrukce (proces mění stavy diskrétně). Tato změny nejsou u valné většiny procesů deterministické (tj. existují stavy procesu, z nichž nelze odvodit všechny stavy budoucí), neboť proces interaguje s nezávislým okolím, ale závislost budoucích stavů na stavu aktuálním je velmi vysoká. Tato skutečnost vede k definici důležitého pomocného pojmu:

Kontext procesu

kontext procesu je sjednocení stavů všech prostředků OS, jež proces v daném okamžiku používá.

Bohužel tato stručná „definice" není definicí, ale spíše základním vymezením pojmu. Doufám, že i vy se nyní v duchu zamýšlíte, co znamená tvrzení „v daném okamžiku používá?"

Východiskem k odpovědi je převažující determinismus procesů. Do kontextu procesu patří stavy všech prostředků, u nichž je alespoň potenciální možnost, že by jejich změna mimo proces (např. ztráta informací), vedla ke změně chování programu (typicky ale nikoliv výhradně ve směru nefunkčnosti). Mezi tyto prostředky mohou v konečném důsledku patřit všechny prostředky operačního systému. Naštěstí většina operačních systémů vyžaduje, aby procesy u většiny prostředků daly explicitně najevo, že prostředek chtějí v budoucnu využívat (tzv. otevření prostředku), či naopak, že prostředek již nepotřebují (a jeho budoucí stavy jsou pro proces irelevantní) [tzv. uzavření prostředku]. Kontext procesu se v tomto případě omezuje pouze na již otevřené a ještě neuzavřené prostředky.

Do široce chápaného kontextu spadají i stavy entit mimo operační systém, s nimiž proces interagoval, či dokonce bude interagovat v budoucnu. Mezi tyto entity patří hardware, jež není spravován operačním systémem (u MS DOSu to byla např. paměť grafické karty v netextových modech), člověk (např. odejde-li člověk od neukončené aplikace na oběd, může po návratu ztratit kontext rozdělané práce). Tyto stavy se však z pohledu OS do kontextu nezapočítávají, neboť systém není schopen tyto stavy ovlivnit, nebo dokonce uložit a později obnovit.

I přes tato elementární omezení je kontext běžného procesu velmi široký a jeho datová representace může mít rozsah několika megabytů či dokonce gigabytů. Například do kontextu běžného programu mohou patřit stavy (uložené informace) následujících prostředků:

- **procesor** (representován množinou registrů, nejdůležitější jsou registry IP [*instruction pointer* – ukazatel aktuální či následující instrukce] a SP [*stack pointer* — ukazatel vrcholu programového zásobníku])
- **zásobník** (uživatelský i rutin jádra)
- kódový paměťový region
- datový paměťový region
- paměť grafické karty
- řídící a stavové registry a vyrovnávací paměti portů (na základní desce resp. zásuvných kartách)
- vyrovnávací paměti a registry zabudované do periferních zařízení (klávesnice, tiskárny)

- částečně potištěná stránka v tiskárně
- obsah otevřených souborů na disku
- vypálené stopy na právě vypalovaném CD-R(W)

Velikost takto široce pojatého prostoru je v řádu megabytů a pouhé uložení tohoto kontextu na odkládací zařízení (pevný disk) by trvalo celé vteřiny. Podobně dlouhou dobu by probíhala obnova kontextu. Obě operace se však ve víceúlohovém systému musí provádět při každé výměně procesů (= přepínání kontextů), což se pro dosažení iluze souběžného běhu procesů děje mnohokrát za sekundu.

Jak vyřešit tento zřejmý rozpor? Cesty jsou dvě, buď je nutno snížit velikost kontextu, nebo alespoň minimalizovat kontext, jenž je pro úspěšné obnovení procesu nutno uložit (tzv. *persistentní kontext*).

vyhrazený prostředek

Hlavním východiskem, je skutečnost, že pokud je prostředek užíván výhradně jedním procesem (dále jen **vyhrazený prostředek**), není nutné jeho stav obnovovat a tím ani ukládat (nepatří tudíž do persistentního kontextu). Důvodem je neměnnost stavu prostředku během odložení procesu, neboť ostatní procesy nemají k prostředku přístup. Protože však fyzických prostředků je omezený počet (pokud nejsou dokonce jedinečné) lze stavu úplného vyhražení prostředků dosáhnout pouze virtualizací prostředků.

Pro virtualizaci, jejímž cílem je vyhrazení prostředků, jsou užívány dvě základní strategie: *rozdělení sdílených prostředků* a *vyhrazený server.*

Strategie rozdělení sdílených prostředků je užívána u většiny prostředků, neboť téměř vždy je možno prostředek alespoň virtuálně rozdělit na menší části, jež mohou být přiděleny procesům do výhradního užívání.

Klasickým příkladem této strategie je rozdělení vnějšího paměťového zařízení na soubory, které jsou ve většině případů ve výhradním vlastnictvím procesů. Pokud je (spíše výjimečně) soubor sdílen, lze jej rozdělit na záznamy a u nich pomocí synchronizace zajistit vzájemné vyloučení (tj. prostředek je opět vyhrazen).

Dalším důležitým příkladem této strategie je rozdělení displeje (monitoru) na okna, jež jsou opět ve výhradním vlastnictvím procesů. Zde je dělení poněkud složitější, neboť dělen není samotný fyzický prostředek (zobrazovací plocha monitoru), ale jeho softwarové rozšíření, jež lze interpretovat jako vícevrstvá kreslící (bitmapová) plocha, v níž se mohou okna nacházet v různých vrstvách a tudíž se i vzájemně překrývat.

Příkladem této strategie je *de facto* i paměťový region, jenž vzniká rozdělením fyzické či spíše virtuální paměti a je užíván právě jedním procesem.

Strategie vyhrazených serverů je užíván u prostředků, jež lze obtížně rozdělit a pro něž nelze užít ani časový multiplex s dostatečně krátkými intervaly. V tomto případě se jeden z procesů stává výhradním vlastníkem prostředku, jehož funkčnost nabízí prostřednictvím odvozeného a snadněji sdílitelného prostředku nejčastěji ve formě fronty požadavků.

Klasickým příkladem této strategie je správa tiskárny tiskovým serverem (pouze proces server má přístup k tiskárně) další procesy musí využívat nepřímý prostředek — tiskovou frontu. Mezi další příklady patří dial-up připojení prostřednictvím modemu (odvozeným prostředkem je IP paketové spojení), či CD vypalovačka.

Ve svém důsledku vede minimalizace kontextu a persistentního kontextu k vyloučení stavů všech prostředků z ukládaného a obnovovaného kontextu vyjma stavu registrů procesoru a koprocesorů a zásobníku procesu na úrovni jádra. Tento tzv. minimální kontext má rozsah v řádku kilobytů (např. v Linuxu je to cca 16 KB), jehož uložení či obnovení je v řádu mikrosekund (pro uložení stačí operační paměť).

3.2 Multitasking

Jako multitasking se označuje taková správa procesů, jež umožňuje existenci vícero nezávislých procesů v jednom okamžiku. Důležité je především slovo „nezávislých“, neboť nezávislost procesů je mírou skutečné víceúlohovosti systémů. Pokud například proces musí čekat na dokončení *synovského procesu* (= proces vytvořený a spuštěný tímto procesem), nelze ještě hovořit o multitaskingu (viz vzájemné volání procesů). Jestliže proces musí čekat na volání systémové služby běžícím procesem, je tento systém mnohdy označován jako multitáskový (tzv. *kooperativní multitasking*). Skutečný multitasking (označovaný jako *preemptivní*) nemá žádné z těchto omezení, ani v tomto případě však nelze hovořit o úplné nezávislosti (závislosti však v tomto případě nejsou dány návrhem, ale omezeným množstvím prostředků).

Jedním z efektů skutečného multitaskingu (a částečně i multitaskingu kooperativního) je při dostatečně rychlé výměně procesů (v řádu milisekund) iluze souběžného běhu více procesů i na jednoprocesorových strojích. Z tohoto důvodu je multitasking nutností nejen u víceuživatelských systémů, ale i u nededikovaných síťových serverů (tj. serverů, které poskytují více různých služeb většímu počtu klientů) a v neposlední řadě u interaktivních operačních systému s grafickým rozhraním (zde by principiálně stačil i multitasking kooperativní, ale většina systémů tohoto druhu nabízí *preemptivní* multitasking).

3.2.1 vzájemné volání procesů

Systémy, které nabízejí pouze vzájemné volání procesů, jsou již zastaralé (a to i v oblasti těch nejmenších univerzálních počítačů), poskytují však dobrý úvod do teorie multitaskingu.

Předpokládejme, že v systému se vzájemným voláním procesů existuje (a tím i běží) jeden proces. U většiny systémů tohoto druhu to byl příkazový procesor (shell), jenž vznikl při bootování operačního systému. Nový proces může vzniknout pouze v rámci volání služby jádra tímto procesem. Volání pozastaví aktuální proces, uloží jeho kontext a vytvoří iniciální kontext nového procesu, kterému následně předá řízení. Nyní běží jen tento nový proces, který po jisté době zavolá službu jádra *exit* [*API::exit(void)*], aby byl ukončen. Ve službě *exit* synovského procesu je tento proces destruován (jsou uvolněny jím užívané vyhrazené prostředky) a následně je obnoven kontext původního procesu, jenž tak může ukončit svůj běh. Samozřejmě i synovský proces může spustit nový proces a pozastavit se (a tak dále rekurzívně) čímž vzniká celý zásobník procesů, v němž pouze proces na vrcholu běží.

Úschova kontextu se děje v rámci systémového volání v okamžiku, kdy systém užívá relativně málo prostředků (mimo jiné nezapisuje na disk, nepřistupuje k počítačové síti, apod.) a ukládání a obnovování kontextu nemusí být extrémně rychlé (řádově jde o setiny resp. dokonce desetiny sekundy). Z tohoto důvodu není nutná důsledná virtualizace prostředků, což výrazně zjednoduší návrh systému a může zvýšit jeho výkon (to je však sporné). Určitým problémem byla u těchto systému správa tiskárny, u níž je správa specializovaným procesem (tiskovým) serverem vhodná i u těch nejjednodušších systémů. Proto například v MS DOSu řešil tisk program, jenž běžel v omezeném kooperativním multitaskingu (tento program přirozeně neběžel jako běžný proces, ale jako rutina vykonávaná v obsluhách některých externích i programových přerušení).

3.2.2 kooperativní multitasking

kooperativní multitasking

U **kooperativního multitaskingu** k výměně procesů nedochází pouze na požádání běžícího procesu (tj. ve specializované službě operačního systému), ale vždy, když aktuální proces nemůže pokračovat v běhu, neboť čeká, až bude splněna podmínka (jejíž splnění nemůže ovlivnit), např. stisk klávesy nebo pohyb myši. Jinak řečeno proces se vzdá procesoru tehdy a jen tehdy, pokud ho v danou chvíli nepotřebuje (dobrovolně, kooperativně).

Na druhou stranu musí systém zajistit, že běh procesu bude obnoven poté co se podmínka stane pravdivou (tj. nastane událost, na niž proces čekal).

Tyto předpoklady vedou k následujícím požadavkům na operační systém:

- k výměně procesů může docházet pouze ve službách jádra, neboť pouze jádro řídí přístup k prostředkům a zajišťuje komunikaci procesů (změnu stavu může vyvolat pouze jiný proces nebo přerušení od fyzického zařízení). Navíc pouze v jádře může systém zajistit dostatečnou minimalizaci prostředků.
- speciální rutina, nazývaná dispečer (dispacher) musí rozhodnout, který proces bude obnoven, neboť může existovat více pozastavených procesů schopných dalšího běhu (tj. procesů, u nichž očekávaná událost již nastala)
- systém musí vyřešit i situaci, kdy neexistuje žádný proces schopný běhu (tj. všechny čekají na událost). Systém v tomto případě nejčastěji spouští tzv. *idle proces,* jenž v nekonečném cyklu volá dispečera bez toho, že by cokoliv jiného provedl (pouze u systémů se správou napájení může při delší nečinnosti suspendovat hardware včetně procesoru)

Pokud aplikace komunikuje s okolím (tj. s ostatními procesy a uživateli), tj. v pravidelných intervalech čeká na externí událost (zprávu-odpověď, stisk klávesy, pohyb myši, příchod síťového paketu, časový okamžik apod.), je *kooperativní multitasking* velmi vhodnou strategií. Méně vhodný je však pro procesy, jež si vystačí pouze s výpočty nad operační pamětí (např. vědeckotechnické výpočty), které je nutno uměle rozčlenit do úseků, mezi kterými se procesor vzdává procesoru voláním některé z příslušných služeb operačního systému. Největší nevýhodou kooperativního multitaskingu je možnost časově neohraničeného držení procesoru jedním procesem (např. pokud proces omylem vstoupí do nekonečné smyčky bez volání systémových služeb). Pokud tato situace nastane, není postižen jen daný proces, ale i všechny ostatní včetně procesů systémových, neboť ty již nikdy nezískají procesor. Dokonce i když proces drží procesor konečnou, ale relativně dlouhou dobu (od desetin sekund výše), mohou být postiženy procesy, jež vyžadují rychlou reakci na vnější událost resp. by měly běžet bez výrazných prodlení. Příkladem budiž přehrávače multimédií či servery síťových služeb (např. HTTP servery).

Z tohoto důvodu se tyto systémy prosadily pouze z důvodů snadnější implementace a menších nároků na hardware (nevyžadují totiž důslednou virtualizaci prostředků). Princip vzájemné dobrovolné spolupráce procesů však není překonán a je součástí i vyspělejšího preemptivního multitaskingu (a je tam dokonce převažující).

Z hlediska aplikačního programátora je důležité, že kooperativní multitasking si vynucuje použití specifického stylu ***programování***, jež je označováno jako ***událostmi řízené*** (to je možno použít i u preemptivního multitaskingu, tam je však nepovinný!). Jádrem událostmi řízeného programu je tzv. událostní smyčka, což je nekonečný cyklus v jehož rámci se volá dispečer. Dispečer rozhodne, který proces bude pokračovat v běhu (pokračovat mohou pouze procesy jejichž fronta událostí není prázdná). Pokud je proces zvolen,

pokračuje smyčka převzetím události na vrcholu fronty a na základě jejího typu zavolá obslužnou rutinu (to jest proceduru, jež je registrována pro danou událost před voláním událostní smyčky). Rutina událost ošetří (to nesmí trvat příliš dlouho) a bezprostředně poté se vrací do smyčky, jež pokračuje novou iterací tj. dalším voláním dispečera.

Událostmi řízené programování klade důraz na interakci procesu s okolím, snižuje nebezpečí tvorby programů, jež drží procesor neadekvátní či dokonce neohraničenou dobu (ale neodstraňuje jej) a je velmi vhodné pro širokou třídu programů (interaktivní GUI programy).

3.2.3 preemptivní multitasking

Omezení a nevýhody *kooperativního multitaskingu* lze vyřešit relativně malou změnou strategie přepínání procesů. U kooperativního multitaskingu může k výměně procesů dojít pouze tehdy, čeká-li proces na externí událost a nepotřebuje tudíž procesor. Existují však i další místa, v nichž může potenciálně dojít k výměně procesů:

1. okamžik bezprostředně před návratem z libovolné služby jádra (jádro vždy uvolní nevyhrazené prostředky před skončením služby)
2. při návratu z obsluhy přerušení do uživatelského režimu (tj. jen u přerušení, jež byla vyvolána za běhu procesu v uživatelském režimu)

Důsledky této relativně malé změny strategie jsou obrovské. Mezi ty pozitivní patří především fakt, že proces může být zbaven procesoru i nedobrovolně (tj. mohl by procesor využívat, ale místo toho bude nuceně vystřídán [předběhnut][1]jiným procesem). Multitasking připouštějící nucené vystřídání je proto označován jako **preemptivní**. Další výhoda je spojena s existencí speciálního hardwarového zařízení — časovače, jenž v pravidelných intervalech (u dnešních systému v řádu milisekund) vyvolává (externí) přerušení, což umožňuje výměnu procesů v pravidelných a relativně krátkých intervalech (viz dále *sdílení času*).

preemptivní multitasking

Bohužel i tak malá změna strategie výměny procesů si vyžádá rozsáhlé změny ve filozofii systému a zesložiťuje jeho návrh. Důvody jsou následující:

- povinná systémová správa všech hardwarových prostředků a jejich úplná virtualizace (proces může být v uživatelském režimu přerušen v libovolném [a tudíž nepředvídatelném] okamžiku)
- minimalizace (časových) závislostí mezi procesy (nejdůležitější je odstranění překážek nezávislého běhu ve správě paměti ⇒ nutnost virtualizace paměti)
- proces musí explicitně poskytovat alespoň částečnou informaci o svém nároku na využití procesoru tj. musí být vytvořen dynamický systém priorit procesů

Při popisu preemptivního multitaskingu je nutné zdůraznit, že vynucené odebrání procesoru je spíše výjimkou a že i zde převažuje kooperace. *Preempce* se uplatňuje pouze v *uživatelském režimu* a i zde jen tehdy, drží-li proces procesor příliš dlouho (tj. nevzdává se ho při čekání na externí událost) a systém je zároveň alespoň částečně zatížen. Zásadním rysem je pak kooperativní charakter jádra, jehož rutiny prováděné v kontextu procesu se vzdávají procesoru pouze tehdy, jestliže musí čekat na událost (uvolnění prostředku, vstup resp. výstup dat, apod.) a nikoliv např. při přerušení (včetně časovače). Jedinou výjimkou je přirozeně potenciální výměna procesů bezprostředně před

[1] *preemption* = nucená výměna, přednostní získání (nabytí)

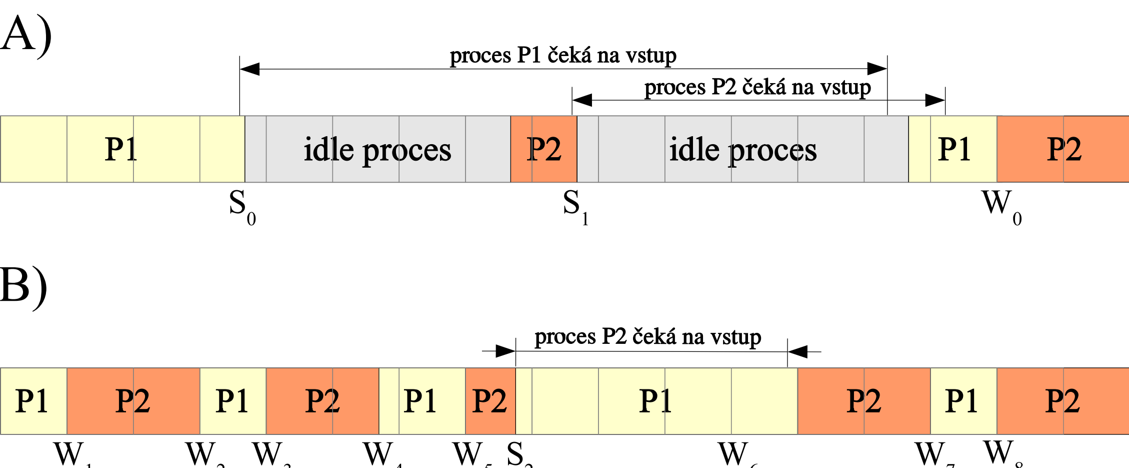
návratem do uživatelského režimu (ta je sice provedena v privilegovaném režimu jádra, ale mimo jeho výkonné rutiny). Tento přístup výrazně usnadňuje návrh jádra, neboť na straně jedné se omezí používání mechanismů explicitní synchronizace (viz dále), na straně druhé se při pečlivé implementaci jádra neprojeví ztrátou výhod preemptivního multitaskingu (rutiny se vzdávají procesoru, kdykoliv je to možné). Pokud by však kooperativní jádro nevyhovovalo, jako např. u systémů reálného času, lze nepreemptivní části jádra omezit na několik málo rutin (tzv. synchronizační primitiva).

Jak bylo řečeno, kooperativní charakter práce systému se projevuje především při malém zatížení systémů. Tento stav je však u interaktivních jednouživatelských operačních systémů na osobních počítačích téměř pravidlem (běžně je 95–99 % procesorového času nevyužito). V tomto případě procesy zůstávají na procesoru, tak dlouho jak potřebují (tj. vzdávají se procesoru dobrovolně) a většinu času běží *idle proces*. Občasnou výjimkou jsou situace, kdy se více procesů dočká události nedlouho po sobě a jeden nestačí událost dostatečně rychle ošetřit, v tomto případě může být některému procesu odebrán procesor nedobrovolně.

Pokud je systém silněji zatížen procesy, které příliš nevyužívají vstupně výstupní operace ani navzájem nekomunikují (typicky vědeckotechnické výpočty či multimediální aplikace), je průběh procesů zcela jiný. Každý proces získává procesor na určitý pevný časový interval (jenž je nepřímo dán jeho statickou prioritou), tj. procesor je mu odebrán v obsluze přerušení od časovače (intervaly přiděleného času jsou celočíselnými násobky kvant časovače). Přidělené intervaly jsou relativně krátké (jednotky max. desítky kvant časovače = řádově max. desítky milisekund), a jsou tudíž neregistrovatelné člověkem.

sdílení času

Vzniká tak časový multiplex čili **sdílení času** (angl. *time slicing* = dosl. plátkování času), jenž vytváří iluzi souběžného (paralelního) běhu procesů. V reálu je však zcela pravidelné sdílení času méně či více často porušováno voláním systémových služeb (proces nevyužije celý přidělený časový interval) a především čekáním procesů na vstup a výstup (resp. jejich explicitní synchronizací), neboť čekající proces se neúčastní časového multiplexu.

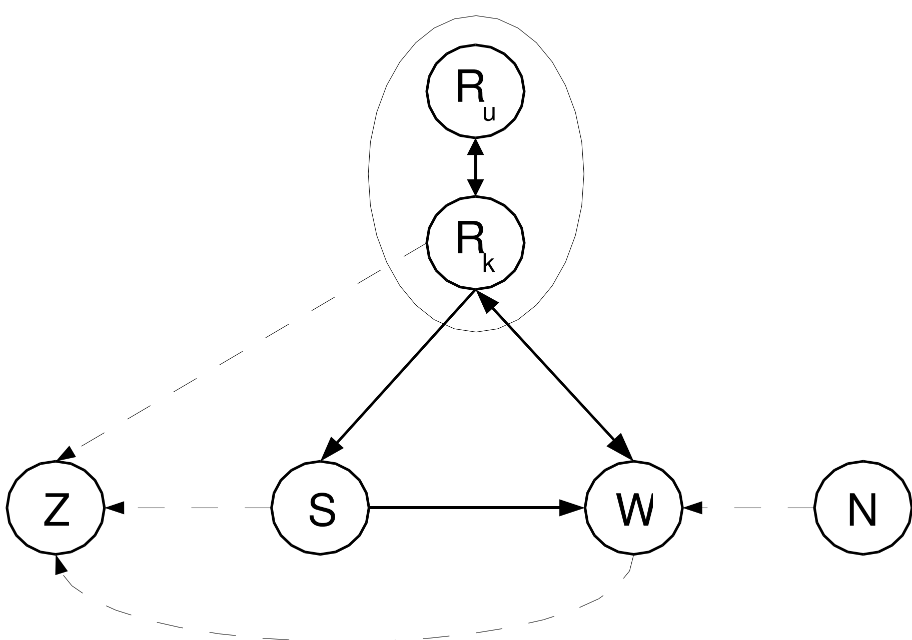


Obrázek ukazuje sdílení procesového času v obou výše uvedených případech. Část **A** ukazuje systém při malém zatížení a s interaktivními procesy (např. událostmi řízené GUI aplikace). Výměny procesů (= přepnutí kontextu) označené S_i representují dobrovolné uvolnění procesoru (proces se zablokuje a přechází do stavu SLEEP), body W_i pak preemptivní (vynucené) přepnutí procesu provedené nejčastěji v obsluze přerušení od časovače. Priority procesů se v tomto případě prakticky neuplatňují (systém je *de facto* kooperativní) s výjimkou přepnutí kontextu po dvou sousledných událostech (proces P1

nestihl dokončit ošetření události a byl předběhnut procesem P2). Část **B** zobrazuje (mírně narušené) sdílení času v zatíženém systému s neinteraktivními a nekomunikujícími procesy. Proces P2 má vyšší prioritu než P1 a tak získává větší podíl na strojovém čase (řádově dvojnásobnou). Na druhou stranu tento proces zavolal dvě služby jádra (v bodě W_4 a S_2), přičemž v druhém se zablokoval čekáním na vstup a přenechal tak procesor procesu P1.

3.3 Stavový diagram procesů

Životní peripetie procesu v preemptivně multitáskovém OS lze vyjádřit tzv. stavovým diagramem procesu. Obecný diagram platný ve valné většině moderních operačních systémů je zobrazen na následujícím obrázku (stavový diagram pro konkrétní operační systém se může lišit jak v názvosloví stavů tak úrovní detailnosti [např. specifikací dílčích stavů]).



Životní pouť procesu začíná ve stavu nový [N] a končí ve stavu zombie [Z]. Většinu života však proces stráví ve dvou základních cyklech: kratší obsahuje stavy *wait* [W] (proces čeká na procesor) a [R] proces běží (tento stav má dva podstavy R_u a R_k podle aktuální úrovně privilegovanosti), delší obsahuje navíc stav *sleep* (S), v němž proces čeká na externí událost (stisk klávesy, dosažení určitého časového okamžiku).

3.3.1 nový proces (stav NEW)

Nový proces je vždy vytvářen na popud jiného procesu (s jedinou výjimkou u bootovacího procesu). Rodičovský proces zavolá systémovou službu pro vytvoření nového (dětského) procesu a rutiny této služby nový proces vytvoří (tj. nový proces je vytvářen v kontextu a na úkor svého rodiče). Vytváření nového procesu počíná vytvořením záznamu v tabulce procesů, pokračuje alokováním základních logických prostředků (především paměťových regionů) a končí emulací iniciálního stavu (iniciální stav emuluje

stav po přerušení již běžícího procesu). Během celé této doby se proces nachází ve stavu „NEW“, ve kterém není schopen samostatného běhu (tj. nesmí být naplánován). Teprve po úplném vytvoření (stále v kontextu rodičovského procesu) je proces přesunut do stavu WAITING (pouhou změnou příznaku v tabulce procesů) a tím začíná jeho samostatná existence (vykonávání však začne až po prvním naplánování). Na straně rodičovského procesu je vytvoření ukončeno návratem ze systémové služby (se signalizací úspěšnosti), oba procesy jsou však již zcela nezávislé a nemají žádné zvláštní vztahy (výjimkou je pouze interakce při zániku dětského procesu).

3.3.2 čekající proces (stav WAITING)

Ve stavu „WAITING“ se nacházejí procesy, které jsou připraveny pro vykonání, ale musí čekat na procesor, jenž je v danou chvíli obsazen (jiným procesem). Není proto divu, že procesy ve stavu WAITING jsou organizovány do jediné fronty (abstraktní datová struktura s chováním první dovnitř první ven = FIFO). Pokud operační systém podporuje prioritu procesů (u reálně použitelného víceúlohového OS je to nutné) je tato fronta frontou prioritní (tj. frontou s předbíháním).

Princip prioritní fronty je jednoduchý (a známý z běžného života). U procesů se stejnou prioritou je funkce prioritní fronty stejná jako u fronty klasické (FIFO). Při neshodě priorit se však procesy řadí podle svých priorit, tj. procesy ve frontě předbíhají ty procesy, jejichž priorita je nižší (tj. poté frontu i dříve opouští). Pro označování priorit se povětšinou používá souvislá podmnožina reálných čísel, kde menší číslo symbolizuje vyšší prioritu (stejně jako například při známkování). Například klasický prioritní systém Unixu používá celočíselný interval $-20 \ldots 20$, kde -20 označuje absolutně nejvyšší prioritu, 0 prioritu běžnou a 20 prioritu absolutně nejnižší (tento systém užívám i zde).

Základní mechanismus plánování je jednoduchý, aktuální (právě běžící) proces je vložen na konec fronty a pro další vykonávání je vybrán proces na samém začátku fronty (což při jisté konstelaci může být tentýž proces).

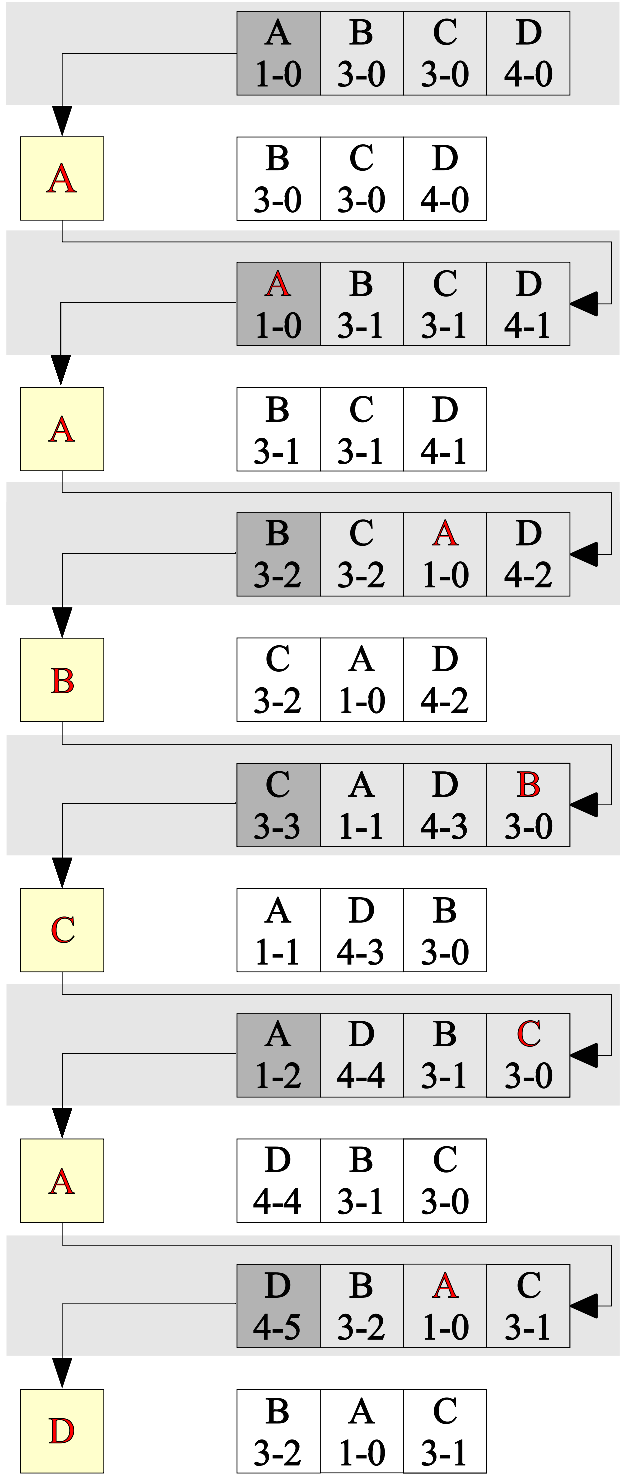
Použijeme-li však tento mechanismus a klasickou prioritní frontu, dosáhneme chování typického pro *preemptivní multitasking* (nedobrovolné sdílení času) jen při shodě priorit. Pokud se priority liší, jsou vždy naplánovány jen procesy (resp. proces) s nejvyšší prioritou, ostatní procesy (resp. procesy na dalším stupni priorit) získají procesor jen tehdy, pokud se všechny procesy z vrcholové skupiny vzdají dobrovolně procesoru (tj. čekají na nějakou událost ve stavu SLEEPING). Operační systém tak získá charakter systému kooperativního, což nemusí být vždy zcela neužitečné (viz dále *reálně-časové procesy*), ale obecně je to zcela nevhodné (výhodou preemptivních systémů je totiž právě téměř absolutní nezávislost procesů).

Je tedy zřejmé, že pro preemptivní plánování procesů je nutné využít složitějšího mechanismu. Nejjednodušším z těchto mechanismů je systém dynamických aktuálních priorit, jenž spočívá v dynamické změně priorit zvýhodňováním čekajících procesů resp. penalizací běžícího procesu (resp. obojím). Tento mechanismus předpokládá že aktuální priorita se skládá ze dvou částí, **priority statické**, jež se během života procesu nemění (pokud si ji proces výjimečně nezmění explicitním voláním specializované služby systému) a ***priority dynamické*****priority dynamické**, jež se mění podle aktuálního stavu tj. $AP = SP + DP$.

statická priorita
dynamická priorita

U systémů se zvýhodněním čekajících procesů se dynamická priorita zvyšuje o stupeň u všech procesů ve *wait* frontě na každý přeplánovací cyklus (v nejjednodušším příkladě

před každým *přeplánováním*). Nulová je pak u procesů, jež do fronty nově vstoupily (ze stavů RUNNING nebo SLEEPING). Duální strategií je inkrementace dynamické priority (tj. snižování priority) u běžícího procesu. Praktická ukázka systému se zvýhodněním čekajících procesů je na následujícím obrázku. Procesy jsou označeny A–D, priority jsou uvedeny ve tvaru SP-DP, čekající procesy jsou zvýhodněny jednotkovým vzrůstem priority na přeplánovací cyklus.



Aplikace této strategie zajišťuje, že v horizontu sekund (resp. dokonce desetin sekundy) získá alespoň jedno časové kvantum procesoru každý proces bez ohledu na svou prioritu (výjimkou jsou pouze přetížené systémy) a že relativní podíl procesů na procesorovém času odpovídá jejich statickým prioritám (tj. proces s vyšší prioritou stráví na procesoru

více času). Bohužel pro takto navržený prioritní systém je velmi obtížné zajistit absolutní podíl procesu na procesorovém čase (resp. jednoduchou závislost tohoto podílu na prioritě), lze jej však zajistit použitím sofistikovanějších strategií.

Druhým nedostatkem, jenž je společný pro všechny preemptivní strategie (u výše uvedené je však velmi výrazný), je nevhodnost preemptivního plánování pro procesy, které musí řídit jisté zařízení v reálném čase (tzv. procesy reálného času). Tyto procesy musí zajistit rychlou odezvu na vnější události (= změny stavu řízeného systému) a to v čase, jenž je striktně shora omezen (tzv. maximální doba odezvy), což při preemptivním plánování zajistit nelze. Důvodem je skutečnost, že proces i při maximální statické prioritě nezíská procesor ihned po události (přerušení), neboť může být předběhnut procesy s nižší statickou ale vyšší aktuální prioritou (navíc ani počet předběhnuvších procesů nelze shora omezit) a tak (i když je to v málo zatíženém systému málo pravděpodobné) může doba čekání procesu na procesor i při nejvyšší možné statické prioritě převýšit maximální dobu odezvy.

Z tohoto důvodu nabízí některé moderní systémy (včetně MS Windows a Linuxu) speciální úroveň resp. úrovně priorit tzv. priority reálného času (*real time priority*). Procesy na této úrovni nemohou být předběhnuty běžnými procesy (tj. pokud se proces nevzdá procesoru, zůstává na něm). Pokud je procesů na úrovni reálného času více, jsou obsluhovány cyklicky. Podporuje-li systém více úrovní procesů reálného času (POSIX a Linux 255 úrovní) nemohou být procesy vyšších úrovní předbíhány procesy úrovní nižší.

3.3.3 běžící proces (stav RUNNING)

V jednoprocesorovém systému existuje právě jeden běžící proces. Běžící proces může běžet buď v režimu neprivilegovaném (uživatelském, aplikačním) nebo privilegovaném (jádra, systémovém), což se zrcadlí i v rozdělení stavu na dva podstavy. Proces může opustit tento stav buď dobrovolně (proces se vzdá procesoru přechodem do stavu SLEEPING) jestliže čeká na systémovou událost; nebo nedobrovolně, je-li mu odebrán procesor při přeplánování (přechází do stavu WAITING a řadí se do prioritní fronty).

3.3.4 zablokovaný proces (stav SLEEPING)

Ve stavu SLEEPING se proces nachází, pokud se dobrovolně vzdal procesoru a čeká na určitou systémovou událost. Mezi hlavní typy událostí patří:

1. vstupní událost na vstupní periférii (stisk klávesy, příchod paketu)
2. výstupní událost (provedení zápisu na disk, odeslání znaku na terminál)
3. uvolnění nesdílitelného fyzického či spíše logického prostředku
4. signalizace (uvolnění) synchronizačního prostředku (viz kapitola 4.2)
5. dosažení jistého časového okamžiku systémového času (např. proces je pozdržen na několik vteřin)

zablokování procesu

Při přechodu do stavu SLEEPING (dále jen **zablokování procesu**) by měl proces využívat minimum dalších prostředků, neboť by je zbytečně blokoval (odblokování může nastat až po několika hodinách či dnech). Protože však do tohoto stavu přechází během vykonávání jádra (tj. kooperativně), může systém uvolnit téměř všechny prostředky vlastněné

procesem. Výjimkou jsou pouze regiony paměti, jež procesů zůstávají, avšak je-li využita virtualizace paměti, jsou procesu postupně ukradeny (či lépe rozkradeny) všechny jím užívané rámce fyzické paměti (proces neběží tj. jeho pracovní množina stránek je prázdná).

Ve stavu SLEEPING se může nacházet více procesů čekajících na stejný vyhrazený prostředek (včetně synchronizačních). Pro jejich uvolňování existují dva mechanismy:

fronta čekajících procesů — procesy čekající na prostředek jsou řazeny do fronty a po uvolnění prostředku je pouze první z nich (= nejdéle čekající) odblokován.

probuďte se a předbíhejte — probuzeny jsou všechny procesy (tj. jsou přeřazeny do fronty). Po naplánování se proces **přesvědčí**, zda je prostředek volný, pokud ano, tak jej začne využívat, jinak se opět zablokuje. O tom, který proces prostředek získá, rozhoduje absolutní priorita (u procesů právě probuzených pak její statická část) resp. při její shodě náhoda.

3.3.5 proces mátoha (stav ZOMBIE)

zombie

Procesy končí svůj pohnutý život jako **zombie** (mátoha). Ze stavu zombie nevede cesta zpět a proces v něm zůstává až do svého úplného odstranění (likvidace).

Proces se do stavu zombie může dostat ze tří příčin, jež lze v analogii s lidským životem nazvat *vražda*, *sebevražda* a *smrtelný úraz*.

vražda — proces je ukončen jiným (aktuálně běžícím) procesem a do stavu mátoha přechází ze stavu WAITING či SLEEPING. Vzájemné zabíjení procesů je přirozeně omezené, například běžný uživatel smí zabíjet pouze své procesy.

sebevražda — běžící proces zavolá systémovou službu pro své ukončení (nejčastěji je označována jako *exit*), v jejímž rámci je přesunut do stavu *zombie* (služba se již nikdy nevrátí do uživatelského programu). Služba *exit* je implicitně volána i na konci programů (volá ji zavaděč uživatelského programu).

smrtelný úraz — pokud program poruší ochranu paměti, použije v uživatelském režimu privilegovanou instrukci, či se jinak snaží narušit stabilitu operačního systému resp. jiných aktuálních procesů (OS však nemůže detekovat všechny snahy tohoto druhu) může být nedobrovolně ukončen. Aplikace je přerušena vyvoláním přerušení od procesoru (tzv. výjimkou např. výpadkem stránky) a v rámci obslužné rutiny je přesunut do stavu zombie.

Při přechodu do stavu mátoha jsou programu odebrány všechny prostředky, sám proces si však nemůže uvolnit zásobník jádra ani nemůže zrušit svůj záznam v tabulce procesů. V obou případech by totiž vznikl problematický stav, neboť proces, jež *de iure* neexistuje (není v tabulce procesů) dále *de facto* běží a navíc užívá uvolněný zásobník pro uložení rámců, přeplánování a přepnutí kontextu (mátoha se přirozeně vzdá i procesoru). Existence speciálního koncového stavu tyto problémy řeší, přesouvá však povinnost konečného odstranění procesu na jiný proces v systému. V otázce volby tohoto procesu-hrobníka se operační systémy liší, například v Unixu za odstranění (pohřbení) odpovídá rodič ukončeného procesu (pokud ještě existuje), jenž tak může převzít i tzv. návratovou hodnotu procesu, respektive proces *init*, jenž existuje po celou dobu standardního běhu systému.

3.4 Vlákna (thready)

Pro preemptivní systémy je typická zdánlivá či skutečná paralelnost na úrovni celého systému (= všech procesů) na straně jedné a lineární provádění instrukcí jednotlivých procesů na straně druhé. To je však mnohdy značně omezující, neboť některé programy vyžadují paralelnost i na aplikační úrovni. Například pokud interaktivní program čeká na klávesnici může v nevyužívaném čase provádět na pozadí pomocné operace (např. textové procesory mohou hledat překlepy, grafické programy mohou optimalizovat zobrazení apod.), nemluvě o plně paralelních algoritmech, jež mohou být víceprocesorových systémech výrazně urychleny, pokud je jejich zpracování rozloženo na několik paralelně pracujících procesorů.

Každou aplikaci lze sice rozdělit do několika procesů a tím dosáhnout požadovaného paralelismu, je to však nepohodlné a náročné na výpočetní prostředky. V obou případech je hlavním důvodem absolutní oddělení procesů ve víceúlohovém systému, které brání přímé komunikaci (lze sice použít komunikační prostředky, jež jsou však pomalé a poněkud neohrabané) a spotřebovává značné množství prostředků (především ve fázi tvorby procesu).

vlákno

Jedním z možných řešení jsou tzv. **vlákna** *(thread)*, jež přinášejí paralelismus výpočtu i na úroveň jednotlivých procesů. Tato vlákna jsou obdobou procesů, sdílejí však datový region, což usnadňuje jejich interakci, neboť mohou komunikovat prostřednictvím globálních statických proměnných. Vlákna tak představují nezávislé toky řízení (tj. každé vlákno má svou aktuální instrukci a svůj programový zásobník), jež souběžně (*concurrently*) běží nad společnou pamětí.

Vlákna lze implementovat i na aplikační úrovni (správce threadů je součástí aplikačního kódu) — *userspace thread*, ale výhodnější je přímá podpora na úrovni jádra (*kernel thread*). Mezi hlavní nevýhody uživatelských vláken patří globální zablokování (jestliže dojde v jednom z threadů k zablokování procesu, jsou pozastavena i ostatní vlákna procesu) a nemožnost rozložení vláken na více procesorů.

korutina

Na aplikační úrovni lze vlákna implementovat jako kooperativní — pomocí tzv. **korutin** či jako preemptivní. Korutiny se relativně snadno implementují na úrovni programovacího jazyka, s minimálními nároky na paměť (obecně stačí jen několika desítek bytů na korutinu) a žádnými nároky na operační systém (celá implementace je v uživatelském prostoru). Hlavní výhodou korutin je téměř neomezený počet souběžných instancí (i stovky tisíc). Proto se stále využívají i když většinou v kooperaci s vlákny na úrovni jádra (korutiny jsou systémem automaticky či řízeně rozprostřeny na preemptivní vlákna poskytovaná jádrem OS, které většinou využívají paralelismu na úrovni procesoru). Preemptivní vlákna na aplikační úrovni vyžadují určitou podporu na úrovni OS (softwarový časovač) a nepřinášejí žádné výhody oproti přímé podpoře v jádru (která je dnes dostupná téměř u všech OS). Z tohoto důvodu se dnes používají spíše výjimečně.

Ahmad Mohsin, Syed Irfan Raza , Syda Fatima. Thread models Semantics: Solaris and Linux M:N to 1:1 thread model. *First International Conference on Modern Communication & Computing Technologies (MCCT'14)*, At Qaid e Awam University Nawabshah. `https://www.researchgate.net/publication/270217184_Thread_models_Semantics_Solaris_and_Linux_MN_to_11_thread_model`

Podpora vláken přímo v jádře si na druhou stranu vyžádá výrazné změny ve správě (a také chápání) procesů. Základní entitou správy procesů se stávají vlákna, neboť jsou objektem přeplánovávání, procházejí stejnými stavy jako klasické procesy (tj. všechna

vlákna v celém systému jsou na této úrovni rovnocenná). Proces se stává odvozeným prostředkem, neboť jej lze popsat jako množinu vláken sdílející společný datový region. Kromě datového regionu jsou však s procesem svázány i další prostředky jako je identifikace procesu (PID), kódový segment (a tím i program), deskriptory otevřených souborů, apod. Každý proces má tzv. hlavní vlákno, jež je vytvořeno během vzniku procesu (stav „N“ procesu) a po přidělení prostředků prochází běžným stavovým cyklem (W-R-(S)--W). Pokud proces vystačí s tímto hlavním vláknem, lze jej spolu s vláknem považovat za dokonalou obdobu klasického procesu. Operační systém však může na požádání (= voláním služby jádra) vytvořit pro proces nové vlákno, které začne vykonávat část programu (nejčastěji funkci či proceduru) v souběhu s ostatními vlákny procesu (včetně hlavního vlákna).

Hlavní výhodou vláken oproti procesům je kromě snadnější a efektivnější kooperace paralelních toků řízení i rychlejší a na prostředky méně náročné vytváření nových vláken, což umožňuje užití většího počtu souběžně běžících vláken bez výrazného zpomalení systému. S touto vlastností vláken souvisí i alternativní označení vláken jako *odlehčených* (*lightweight*) *procesů.*

PŘEČTĚTE SI

Alternativní pohled na správu procesů nabízí třetí kapitola knihy *Operating Systems Internals and Design Principles* (Stallings) [12]_

3. Process Description and Control

OTÁZKY

1. Proč jsou procesy jen částečně deterministické? (z hlediska OS)
2. Proč není potřeba ukládat stav vyhrazených prostředků?
3. Popište funkci správce tiskové fronty.
4. Jaký styl programování je typický pro kooperativní multitasking?
5. Uveďte příklady aplikací, které nelze efektivně implementovat v kooperativním multitaskingu?
6. Jakou funkci má rodičovský proces?
7. Jak by vypadal stavový diagram pro systém s kooperativním multistaskingem?
8. Co určuje statická priorita procesu? K čemu ji lze využít (popište
9. na příkladech aplikací)
10. Co sdílejí vlákna?

OTÁZKY K ZAMYŠLENÍ

1. Každý proces v systému je popsán záznamem v tabulce procesů. Pokuste se navrhnout, jaké položky by měla mít struktura popisující proces (viz především jednotlivé stavy).
2. Definujte, do kontextu jakého procesu patří klávesnice (předpokládejte klasický okenní GUI systém s jedinou grafickou konzolí resp. textový systém s podporou více terminálů).
3. Pokud proces-rodič neodstraňuje své potomky ve stavu zombie, začnou se procesy-mátohy hromadit. Jaké to může mít negativní důsledky? Jak se můžete zombie zbavit?
4. Programy typu *benchmark*, zjišťující (mimo jiné) výkon procesoru, využívají měření čistého času provedení jistého algoritmu. Navrhněte, jak zajistit správnou funkci tohoto programu i ve víceúlohovém prostředí.
5. Jak vzniká nový proces v Linuxu? (volání fork)
6. Co je tzv. inverze priorit?

ÚKOLY

1. Zkuste spustit proces s realtimovou prioritou (musíte být *root*).
2. Vytvořte program vytvářející zombii. Stačí provést funkci *fork*, zablokovat se ve větvi rodiče a ve větvi nového procesu ukončit program.
3. Vytvořte skript, který bude průběžně hlídat vznik zombií a vypisovat údaje o nich. Zdrojem informací by měl být výstup příkazu *ps*. jak často déletrvající zombie vznikají?

4 Synchronizace a meziprocesová komunikace

CÍLE KAPITOLY

Synchronizace a meziprocesorová komunikace patří mezi ty aspekty operačního systému, které se výrazně projevují i na úrovni aplikačního programování (a jejich důležitost stále roste).

Pochopení této problematiky by Vám mělo pomoci při:

- určení kdy a kde explicitně synchronizovat procesy nebo vlákna
- volbě optimálních synchronizačních prostředků
- správné interpretaci chyb, které vzniknou nadbytečnou nebo chybnou synchronizací (hlavně problém uváznutí)
- volbě optimálních nízkoúrovňových prostředků meziprocesorové komunikace

Problematika synchronizace je klíčová pro velkou třídu aplikačních programů od GUI aplikací s činností prováděnou na pozadí až po masivně paralelní výpočty. To mimo jiné znamená, že rozsah této kapitoly je nedostatečný i pro začínajícího programátora. Pohled na vyšší úrovni abstrakce se zaměřením na paralelní systémy proto nabízí i kurz „Paralelní programování“.

Jako doplňkový zdroj informací lze využít knihu věnovanou paralelnímu programování (což je jedno z možných aplikací synchronizace a komunikace).

GOVE, Darryl. *Programování aplikací pro vícejádrové procesory.* Vyd. 1. Brno: Computer Press, 2011, 416 s. ISBN 978-80-251-3487-0.

Klíčové jsou především kapitoly 4 a 5.

4.1 Kritický kód a vzájemné vyloučení

Pro preemptivní multitasking je typická téměř úplná vzájemná nezávislost procesů. Jedním z prostředků dosažení tohoto stavu je striktní oddělení prostředků užívaných jednotlivými procesy (nejdůležitější je striktní oddělení adresových prostorů), což zcela brání kolizi procesů při jejich užívání.

Bohužel tento ideální stav panuje pouze na uživatelské úrovni, a to výhradně u procesů, jež s okolními procesy nikdy nekomunikují. Zcela jiná situace je rutin jádra, jež přistupují ke sdíleným prostředkům (především fyzickým, neboť ty jsou jedinečné nebo

jsou k dispozici v omezeném počtu), u vzájemně komunikujících procesů a především u preemptivních threadů, jež sdílejí všechny globální proměnné.

Předpokládejme například, že dva nezávislé procesy P a K si vyměňují data prostřednictvím sdílené paměti o velikosti v řádech KB tak, že P do paměti zapisuje (produkuje data, dále je nazýván [proces] producent) a K z paměti čte (konzumuje data, je proto označován jako konzument). Pro zjednodušení předpokládejme, že producent data produkuje jako celek a konzument je jako celek vyžaduje.

Například program, v němž P produkuje (donekonečna) šifrovací klíče a K je používá, může mít následující podobu:

producent (P):

```
API::SharedRegion shmem ("keySHM",2048);

void production(void)
{
  char localbuff [2048];

  while(1)
    {
      genkey(localbuff); #generovani klice
      shmem.write(localbuff);
    }
}
```

konzument (K):

```
API::SharedRegion shmem ("keySHM",2048);

void consumation(void)
{
  char localbuff [2048];

  while(1)
    {
      shmem.read(localbuff);
      usekey(localbuf);
    }
}
```

náhodný souběh

U preemptivního systému, kdy oba procesy běží v **náhodném souběhu** (tj. k přepnutí mezi procesy může dojít v libovolném místě obou procesů a pro žádnou dvojici instrukcí z obou procesů neexistuje předem zjistitelná časová následnost), nemusí spolupráce mezi programy fungovat. Důvody lze rozdělit do čtyř skupin:

1. první operace čtení sdílené paměti u konzumenta proběhne před prvním zápisem konzumenta, tudíž konzument přečte neinicializovanou sdílenou paměť, jinak řečeno konzument využívá prostředek (zde šifrovací klíč), který ještě není k dispozici. Tato souběhová chyba je typická pro programy užívající pro komunikaci

dočasné prostředky ve formě zpráv. Je nutné zdůraznit, že program nemusí mít v obecném případě možnost tuto situaci detekovat, a tudíž může interpretovat původní obsah paměti jako zprávu.

2. konzument přečte dvakrát či dokonce vícekrát tentýž klíč (tj. po zápisu zprávy proběhne alespoň dvakrát cyklus konzumenta). Tento případ je mírnou modifikací předchozího a je též typický pro zprávově orientovanou komunikaci. Navíc stejně jako v předchozím případě nemusí existovat možnost detekce.
3. nepřečtení klíče (klíčů), z důvodů jeho (jejich) bezprostředního přepsání producentem (tj. konzument nestačí klíče dostatečně rychle číst).
4. nejsložitějším případem je překryv zpráv, kdy konzument přečte sdílenou paměť, jež obsahuje fragmenty dvou klíčů, přičemž na počátku je kus klíče novějšího (zbývající kus zprávy se ještě nestačil zapsat, neboť producent byl uprostřed metody *write* zbaven procesoru [změna stavu R->W]) na konci nepřepsaný zbytek klíče staršího. Výsledkem je tudíž přečtení zcela nekonzistentních dat, jejichž použití vede ve většině případů k chybám (mnohdy jen velmi obtížně odhalitelným).

Situace dané náhodným souběhem jsou velmi komplikované, neboť nejenže může nastat libovolný z chybových stavů, ale může dojít i ke vzniku kombinovaných chyb (např. stejná nekonzistentní zpráva je přečtena několikrát), nebo naopak nemusí po dlouho dobu dojít k žádné negativní kolizi (tj. program se jeví jako správný). **Bohužel právě nahodilý a v mnoha případech v čase velmi řídký výskyt chyb vznikajících kolizním souběhem patří k největším problémům, jež problematika synchronizace přináší, neboť to velmi znesnadňuje detekci chyb ve fázi ladění programů.**

Například pokud by doba generování a použití klíče ve výše uvedené příkladu převyšovala o mnoho řádů dobu zápisu a čtení do sdílené paměti (což je ve skutečnosti typický případ), docházelo by k chybám čtvrtého druhu jen zřídka (např. jen v jednom z miliónu případů, což může být v reálném čase jen jednou za několik dní) a navíc zcela náhodně (tj. mezi chybami by nebyl stále stejný interval), což je černý sen každého programátora nebo informačního analytika.

> Pokud Vás při při studiu ukázky napadlo, že problému se synchronizací by se bylo možno vyhnout zavedením pomocné sdílené proměnné, jež by popisovala stav sdílené paměti (tj. obsahovala by stavy prázdná, plná, [právě] čtena, [právě] zapisována), musím Vás zklamat, neboť problémy se synchronizací se pouze přesunou na onu proměnou a její nastavování a testování. Nanejvýše se tak pouze poněkud omezí a budou méně časté (což však z pohledu ladění není příliš dobrá zpráva). Detailnější rozbor problému s nastavováním a testováním jedné sdílené proměnné viz kapitolu o binárním semaforu.

Souběhové kolize však nenastávají pouze při komunikaci, ale hrozí při téměř každém přístupu k libovolnému sdílenému prostředku. Například pokud by například výstupní port vyžadoval nejdříve nastavení datového registru vystupujících dat spolu s registrem cílovým (identifikující cíl vystupujících dat), pak by při nepříznivém souběhu dvou požadavků mohlo dojít k odeslání dat na špatnou adresu (a to bez ohledu na pořadí nastavení obou registrů). Například při následujícím souběhu jsou data druhého procesu odeslána na adresu určenou prvním procesem (tento souběh není příliš pravděpodobný, ale nelze jej vyloučit, pravděpodobnější je vícenásobné odeslání týchž dat).

proces 1	proces 2
nastav datový registr	
	nastav datový registr
nastav cílový registr	
odešli data	
	nastav cílový registr

kritický kód

Souhrnně lze říci, že každý přístup ke sdílenému prostředku resp. pokus o nesynchronizovanou komunikaci je potenciálně nebezpečný (bez ohledu na skutečnou existenci kolidujícího procesu) a zaslouží si tudíž běžně užívané označení **kritický kód**. Bohužel i přes přesnou definici kritického kódu je jeho praktické vytyčení mnohdy velmi obtížné a aplikační programátoři jej mnohdy ani nepoznají tím spíše, že je v aplikačních programech nepříliš četný (četnost však výrazně roste u silně kooperujících vícethreadových aplikací). Jeho nezohlednění je však závažnou sémantickou chybou s krajně obtížným laděním.

synchronizace procesů

čekání na událost

Negativním (kolizním) souběhům a komunikačním ztrátám lze zabránit pouze **synchronizací procesů** tj. uvedením procesů do jisté předem známé časové závislosti. V podstatě existují dva základní typy synchronizace:

čekání na událost — proces čeká na událost, jež je výsledkem činnosti jiného procesu. Synchronizace musí zajistit nejen časovou následnost obslužné rutiny události (kauzalita akce → reakce), ale ve většině případů i zabránit propásnutí události čekajícím procesem (například je-li tato zaneprázdněna obsluhou předchozí události). Tento typ synchronizace svázaný s komunikací řeší problémy dané rozdílnou rychlostí producenta a konzumenta a prvotního načasování (tj. typy 1–3 příkladu se šifrovacími klíči).

vzájemné vyloučení

vzájemné vyloučení (*mutual exclusion*) — synchronizace musí zabránit současnému vykonávání dvou kritických kódů nad stejným prostředkem. Přesněji řečeno, pokud jeden proces vykonává kritický kód tj. zahájil nebo dokončil vykonávání první instrukce (= vstoupil do kritického kódu) a ještě neprovedl instrukci poslední (= nevystoupil z kritického kódu), nesmí jiný proces vstoupit do kritického kódu nad týmž prostředkem. Vzájemné vyloučení brání vzniku přechodných nekonzistencí ve stavu sdíleného prostředku.

Oba typy synchronizace lze kombinovat a vyřešit tak všechny problémy dané nepředvídatelným souběhem. Bohužel to není zadarmo, neboť synchronizace vnáší závislosti mezi dříve nezávislé procesy, což může vytvářet celé složité sítě závislostí (např. proces A čeká na B a ten na C a D, proces E na B a D atd.) což vede v lepším případě k výraznému zpomalení procesů (procesy jsou většinu času zablokovány ve vzájemném čekání a procesor je téměř nevyužit), v horším pak k *uváznutí* dvou nebo více procesů (procesy jsou navždy zablokovány ve vzájemném čekání a alespoň jeden z nich proto musí být ukončen).

Uváznutím se detailněji zabývá kapitola 4.3 na straně 72, již zde je však možno uvést důležitou zásadu, že stejně jako nedostatek synchronizace škodí naopak i její nadbytek. Je proto chybou vyžadovat vzájemné vyloučení nad kódem, jež není kritický (a to včetně kódu, který bezprostředně kritický kód předchází nebo jej následuje).

4.2 Synchronizační prostředky

4.2.1 binární semafor

Binární (dvojstavový) semafor patří mezi nejdéle známé a používané synchronizační prostředky (v rámci tzv. obecných semaforů je zavedl v polovině 60. let Dijkstra). Hlavní výhodou použití binárních semaforů je jejich jednoduchá sémantika, nevýhodou vyšší chybovost (semafory lze snadno použít nepřípustným způsobem).

binární semafor

Binární semafor je sdílený (logický) prostředek se dvěma stavy {ČERVENÁ, ZELENÁ} a dvěma operacemi *wait* (Dijkstra označuje tuto operaci symbolem „P") a *signal* Dijkstra označuje tuto operaci symbolem „V") a následující **předběžnou** sémantikou:

wait: pokud je semafor ve stavu ZELENÁ, je bezprostředně přepnut do stavu ČERVENÁ, jinak proces vyčká změny stavu semaforu na ZELENOU (ta je provedena voláním operace *signal* jiným procesem) a až poté sám změní stav semaforu na ČERVENOU.

signal: stav semaforu je nastaven na ZELENOU

> Z definice vyplývá, že semafory používané pro řízení dopravy jsou i přes stejnou základní funkci (i dopravní semafory mají za účel zabránit souběhové kolizi, zde dopravních prostředků) pouze částečnou obdobou binárních semaforů. Hlavním rozdílem je především větší složitost dopravních semaforů, neboť dopravní semafory jsou často závislými členy celých semaforových systémů (např. systém na klasické křižovatce musí být tvořen ne méně než osmi semafory) resp. jsou více než dvoustavové (u silniční dopravy jsou třístavové, v železniční je stavů často ještě více). Silniční semafory nejsou také na rozdíl od binárních semaforů přímo řízeny prostředky, jež je využívají tj. automobily, ale jejich stavy jsou funkcí času (včetně semaforů na střídavých jednosměrkách).
>
> Nejbližší analogií binárního semaforu jsou tedy jednoduché (dvoustavové) semafory na jednokolejných nerozvětvených úsecích železniční tratě (např. hradla). Čekání před červeným semaforem na začátku úseku a automatické nastavení na zelenou po projetí vlaku celým úsekem, zajišťují stejně jako u softwarových semaforů vzájemné vyloučení, tj. nejvýše jeden vlak může v libovolném okamžiku projíždět daným úsekem.

Popisy operací *wait* a *signal* uvedené v definici binárního semaforu nejsou zcela úplné a především korektní. Předpokládejme například předběžnou implementaci semaforu v jazyce C++ uvedenou v následujícím výpisu.

Základním problémem je skutečnost, že pokud by byl semafor implementován tímto způsobem na uživatelské úrovni (tj. v aplikaci), nezajistil by vzájemné vyloučení, neboť při následujícím nepříznivém souběhu by propustil dva procesy do kritické sekce:

proces1	proces2
testování stavu, je zelená	
⟶	testování stavu, je zelená
	změna na červenou
	... (kritický kód)
změna na červenou	⟵
... (kritický kód)	

atomická operace

Aby byl semafor funkční, musí se testování hodnoty semaforu a jeho nastavení v operaci *wait* dít **atomicky**, tj nesmí být přerušeno přepnutím kontextu. To však lze ve většině případů zajistit pouze na úrovni (kooperativního) jádra, tj. semafor musí být systémovým prostředkem, jenž je dostupný přes systémové rozhraní.

Ani implementace v jádře však mnohdy nestačí, neboť atomičnost je nutno zajistit i v případě asynchronních přerušení (typicky od časovače) a dokonce i v případě multiprocesorových systémů. To vyžaduje podporu atomického testování a nastavení (tj. elementárního semaforu) na úrovni procesoru (speciální elementární instrukcí) resp. na úrovni hardwaru (hardwarový protokol mezi procesory). Požadavek atomičnosti je nutno klást i na operaci nastavení semaforu na zelenou v operaci *signal* (nikoliv však na celou operaci včetně odblokování).

Kromě požadavku na atomičnost si zpracování v jádře vynucuje i požadavek na zablokování a odblokování procesů, jež musí být provedeno rutinou jádra.

Výše uvedená implementace semaforu využívá mechanismus fronty blokovaných procesů a odblokování prvního, lze jej však zavést i bez této fronty (mechanismus *probuď se a předbíhej*).

Použití semaforu

Binární semafor lze v prvé řadě užít pro zajištění vzájemného vyloučení nad kritickým kódem (každý sdílený prostředek má přidělen binární semafor), kde má však silnou konkurenci v mutexu (pokud je mutex k dispozici, měl by být použit přednostně).

Nezastupitelnou roli hraje semafor v synchronizaci producenta a konzumenta nad sdílenou pamětí (viz příklad v předchozí kapitole). Podrobnější rozbor však ukazuje, že použití jednoho semaforu nestačí (vyloučeny jsou pouze kolize dané neexistencí vzájemného vyloučení, nikoliv souvislé vícenásobné čtení či zápis). Situaci vyřeší až křížové použití dvou semaforů:

producent (P):

```
API::SharedRegion shmem ("keySHM",2048);
API::semaphore prod("producer",GREEN);
API::semaphore cons("consumer",RED);

void production(void)
```

```
{
  char localbuff [2048];

  while(1)
    {
      genkey(localbuff);
      prod.wait();
      shmem.write(localbuff);
      cons.signal();
    }
}
```

konzument (K):

```
API::SharedRegion shmem ("keySHM",2048);
API::semaphore prod("producer",GREEN);
API::semaphore cons("consumer",RED);

void consumation(void)
{
  char localbuff [2048];

  while(1)
    {
      cons.wait();
      shmem.read(localbuff);
      prod.signal();
      usekey(localbuf);
    }
}
```

Synchronizace na obou stranách komunikace se téměř neliší, oba procesy cyklicky čekají na svůj semafor, provádějí zápis/čtení a následně signalizují semafor komunikačního partnera. Jediným rozdílem je různý stav semaforů před počátkem komunikace.

dualita konzumenta a producenta

Tato téměř dokonalá symetričnost se stane zřejmou, pokud si uvědomíme, že pohled, který za producenta označuje proces zapisující do paměti a za konzumenta čtenáře, není jediný možný, neboť data nejsou jediným vytvářeným a spotřebovávaným prostředkem. Stejně tak lze za prostředek označit i volné místo, jež je produkováno čtoucím procesem (jenž je tedy z tohoto pohledu producentem) a spotřebováváno procesem zapisovacím (tj. z tohoto pohledu producentem). Oba pohledy jsou rovnocenné (přesněji řečeno **duální**) a označení procesu jako producenta resp. konzumenta je tudíž zcela konvenční.

> Příklad: V běžném životě je hostinský čepující pivo chápán jako producent a (potenciální) opilci jako konzumenti. Lze však použít i opačný úhel pohledu, v němž hosté produkují prázdné půllitry, jež jsou spotřebovávány hostinským (samozřejmě peníze do tohoto procesu vnášejí jistou asymetrii).

4.2.2 mutex

Mutex (zkratka z ***mut**ual **ex**clusion* = vzájemné vyloučení) je synchronizační prostředek, jenž se od binárního semaforu liší v jediném avšak velmi podstatném směru:

mutex může být signalizován (uvolněn) pouze procesem, jenž daný mutex drží (= vstoupil přes něj do kritického kódu)

Tento rozdíl se projevuje i v používání jiného názvosloví, jež vychází z nejbližší analogie *mutexu* v běžném světě, jíž je je petlice zamykaná prostřednictvím visacích zámků (petlici může otevřít pouze ten, kdo ji prostřednictvím svého zámku uzamkl) u níž může majitel prvního zámku použít další zámky (resp. dalších otočení klíče — západů).

mutex

Mutex je synchronizační prostředek, jehož stav je určen dvěma hodnotami — identifikátorem procesu, jenž mutex drží (pokud takový proces existuje, jinak je tato hodnota nedefinována) a počtem uzamknutí mutexu (nezáporné celé číslo), a nad nímž jsou definovány dvě atomické operace *lock* (uzamčení resp. získání mutexu) a *unlock* (odemčení resp. uvolnění mutexu). Základním pravidlem je absolutní zamezení držení mutexu více procesy najednou a při použití vícenásobného zamknutí odpovídající počet odemknutí.

lock

je-li mutex volný (= držitel mutexu není definován) stává se proces držitelem mutexu (bez zablokování) a počet uzamknutí nabývá hodnoty 1 (proces pokračuje ve vykonávání bez zablokování)

je-li mutex vlastněn aktuálním procesem, je pouze zvýšen počet uzamknutí (bez zablokování)

je-li mutex vlastněn jiným než aktuálním procesem, je proces zablokován a čeká na uvolnění mutexu

unlock

je-li mutex volný, je chování nedefinováno (v praxi chyba, výjimka, ale i pouhé ignorování)

je-li mutex vlastněn aktuálním procesem, je počet uzamknutí snížen o jedničku, a je-li poté nulový, je mutex uvolněn (odemčen) a jeden z čekajících procesů je odblokován (resp. jsou odblokovány všechny čekající procesy)

je-li mutex vlastněn jiným než aktuálním procesem, je chování nedefinováno (chyba resp. ignorování), ale mutex není v žádném případě uvolněn

Stejně jako semafor musí být i mutex implementován na úrovni jádra a musí být zaručena atomičnost uvnitř operací (jejich struktura je podobná binárnímu semaforu).

použití mutexu

Mutex je používán pro zajištění vzájemné vyloučení nad kritickým kódem (především v případě, že se jedná o jediný atomický prostředek), ale lze jej použít i ve složitějších synchronizačních konstrukcích (viz příklad v následující podkapitole). Hlavní výhodou ve srovnání s binárním semaforem je snazší použití především v rozsáhlejších projektech dané možností vícenásobného zamykání, neboť zatímco u semaforu vede opakované volání operace *wait* bez mezilehlé signalizace k časově neomezenému zablokování procesu,

je opakované volání operace *lock* (bez mezilehlého odemknutí) tolerováno. Pokud se tedy tvůrci/tvůrce programu drží zásady, že při vstupu do kritického kódu je volána operace *lock* a při výstupu *unlock*, nemohou nic zkazit a to ani tehdy, když tak učiní vícekrát na různých úrovních programu (nejčastěji programátor použije nadbytečné uzamknutí kolem funkce, netuše, že to jeho kolega [resp. on sám!] učinil i uvnitř funkce).

4.2.3 událost (event)

Událost (*event*) je velmi jednoduchý synchronizační prostředek, který na rozdíl od semaforu a především mutexu nezajišťuje vzájemné vyloučení, ale zaručenou následnost dvou akcí. Nejčastějším (ale nikoliv jediným) případem je spolupráce dvou procesů, v níž jeden musí čekat, pokud je resp. není splněna podmínka, jejíž pravdivost je ovlivňována druhým procesem.

Základním interním rozdílem mezi semaforem (resp. mutexem) a událostí je bezestavový charakter události, tj. událost si nepamatuje svůj stav. Událost je tedy velmi blízká interním systémovým rutinám blokování a odblokování, na rozdíl od nich je však na straně jedné dostupná na uživatelské úrovni, na straně druhé je omezena na explicitní programovou signalizaci (událost nemůže být přímo vyvolána externí událostí typu stisku klávesy).

Bezestavový charakter události problematizuje jejich použití v případech, kdy je požadována zaručená detekce asynchronního dosažení jistého stavu, neboť pokud je dosažení stavu signalizováno jedním z procesů ještě před vstupem druhého do čekání na událost, nebude tato událost tímto procesem zachycena a ten čeká až na následující výskyt události (u jedinečné události se tudíž zablokuje navždy). Řešením je použití sdílené stavové proměnné, jež reflektuje dosažený stav (nastavována je prvním a testována druhým procesem). Protože však ani nastavování ani testování není *atomickou činností*, musí být obě činnosti (resp. další operace nad sdíleným objektem, jehož je stavová proměnná integrální součástí) chráněny mutexem. Ani to však neřeší problémy se synchronizací, neboť na operaci čekání jsou v tomto případě kladeny dvě téměř protikladné podmínky:

- testování stavové proměnné spolu s reakcí na událost se musí dít uvnitř jediného chráněného kódu (tj. mezi uzamčením a uvolněním mutexu)
- během čekání na událost musí být mutex uvolněn (v opačném případě by proces signalizující dosažení stavu nemohl přistoupit k chráněné stavové proměnné)

Řešením je dočasné uvolnění mutexu při čekání na událost, jež se však musí dít atomicky se vstupem do čekání (tj. proces nesmí být přerušen mezi uvolněním mutexu a vstupem do čekání, neboť i v tak malý okamžik by mohlo dojít k přepnutí na jiný proces, jež by stačil událost signalizovat). Tato spolupráce mutexu a události tedy musí být podporována přímo na úrovni služeb operačních systémů, např. v POSIXu se tak děje navázáním mutexu na událost v konstruktoru události, což zajistí jeho automatické (a atomické) uvolňování při vstupu do čekání na událost a automatické (nikoliv však již atomické) uzamknutí mutexu při výstup.

Složitější spolupráci dvou událostí a jednoho mutexu ukazuje příklad na následující stránce, jež obsahuje implementace synchronizované kruhové fronty (základní sémantika odpovídá POSIXU).

Výše popsaný typ synchronizačních událostí je pouze jedním z možných přístupů k signalizaci asynchronních stavů mezi procesy. Zmínit lze např. trvale klopné události ve

Listing 4.1: synchronizovaná kruhová fronta

```
#include "API.h"

template <typename T, int maxsize>
class CQueue
{
  T data [maxsize];
  int asize;
  T *first, *last;
  API::mutex m;
  API::event nonempty;
  API::event nonfull;

public:
  CQueue(void): asize(0), first(data), last(data),
                notempty(m), notfull(m) {};
  void push(T item);
  T pop (void);
  bool empty(void) {return asize == 0;};
};

template <typename T, int maxsize>
void CQueue<T,maxsize>::push(T item)
{
  m.lock();
  while (size == maxsize) nonfull.wait();

  *last=item;
  last=data+(last-data)%maxsize;
  size++;

  if (size == 1) nonempty.signalAll();
  m.unlock();
}

template <typename T, int maxsize>
T CQueue<T,maxsize>::pop(void)
{
  m.lock();
  while (size == 0) nonempty.wait();

  T aux = *first;
  first=data+(first-data)%maxsize;
  size--;

  if (size == maxsize-1) nonfull.signalAll();
  m.unlock();
  return aux;
}
```

Win32 (událost zůstává v signálním stavu navždy) oproti výše uvedeným pulsním (ty ve Win32 také existují). Mezi synchronizační prostředky tohoto druhu patří i unixovské *signály*, jež sice prvotně sloužily pro signalizaci výjimečných stavů a externí ukončování procesů, ale po rozšíření (tzv. bezpečné signály) mohou sloužit i ke vzájemné synchronizaci procesů.

4.2.4 obecný semafor

Obecný semafor je synchronizační prostředek vhodný pro synchronizaci přístupu procesů k víceprvkové množině ekvivalentních prostředků. Tento prostředek je na jedné straně zobecněním semaforu (binární semafor je pouhým speciálním případem obecného semaforu pro jednoprvkovou množinu prostředků), na straně druhé má společné rysy s událostmi (události jsou obecnější, tj. pomocí událostí [s vázaným mutexem] lze řešit stejnou množinu synchronizačních problémů).

obecný semafor

Obecný semafor je synchronizační prostředek, jehož stavy nabývají hodnot 0 až MAX (konstanta MAX je neměnnou charakteristikou daného semaforu a je určena při jeho definici) a dvěma operacemi *post* a *release*.

operace *acquire* — je-li stav obecného semaforu nulový, proces se zablokuje, dokud není semafor uvolněn, jinak se o jedničku sníží čítač semaforu

operace *release* — zvýšení čítače semaforu o jedničku (je-li současný stav menší než MAX, jinak je operace nedefinována)

Hodnota tedy semaforu určuje počet aktuálně volných prostředků (tj. na počátku je roven celkovému počtu disponibilních prostředků), proces při vstupu do kritického kódu alokuje jeden prostředek (operace *post* + vlastní alokace [registrace] prostředku) a na konci prostředek uvolní (vlastní uvolnění + operace *release*).

Binární semafor je obecným semaforem s hodnotou MAX rovnou jedné, kde operace *wait* odpovídá operaci *acquire* a operace *signal* operaci *release*. Většina systémů proto nabízí ve svém rozhraní pouze obecné semafory (což se mi však nejeví jako zcela vhodné, neboť binární semafory jsou elementárnějším prostředkem).

Podívejte se na synchronizaci pomocí specializovaných instrukcí procesoru. Dobrým východiskem je:

Wikipedia contributors. *Compare-and-swap* [Internet]. Wikipedia, The Free Encyclopedia; 2016 Jan 25, 11:13 UTC [cited 2016 Feb 3]. Available from: `https://en.wikipedia.org/w/index.php?title=Compare-and-swap&oldid=701583818`.

4.3 Uváznutí

V předchozí kapitole jsme se několikrát dotkli problematického místa veškeré synchronizace — trvalého zablokování procesu (procesů) při chybném použití synchronizačních prostředků. Příkladem budiž například situace, kdy proces, jenž získal semafor, opětovně zavolá jeho operaci *wait*. Tento negativní důsledek chybně provedené synchronizace je tak závažný, že si zaslouží vlastní název a následnou detailnější diskusi.

uváznutí

Stav, v němž se jeden (resp. několik procesů) vinou chybné synchronizace navždy zablokuje (tj. bez ukončení nemůže opustit stav SLEEP), označujeme jako **uváznutí** čili *deadlock*.

Bohužel uváznutí zmíněná výše patří do skupiny méně nebezpečných, tj. snadněji odhalitelných a opravitelných. Uváznutí tohoto typu vznikají důsledkem hrubých chyb v použití synchronizačních prostředků, za přispění pouze jednoho procesu a především při každém běhu inkriminované části programu. Označují se jako triviální uváznutí a mnohdy se za skutečná uváznutí ani nepovažují (to však bohužel neznamená, že se v praxi nevyskytují).

Druhou a mnohem nebezpečnější skupinu uváznutí tvoří uváznutí vzniklá *nepříznivým souběhem* (angl.*[adverse] race condition*) dvou procesů. Typickým znakem těchto uváznutí je jejich občasný a zdánlivě náhodný výskyt, jenž je v mnoha případech velmi řídký (např. nastává až po mnoha dnech používání obou kolidujících aplikací). Není proto divu, že se velmi obtížně detekují (ladí), resp. se nedetekují a neladí vůbec. Základním a typickým případem je souběhové uváznutí hrozící tehdy, když dva procesy používají dva společné prostředky. Ostatní (složitější) typy uváznutí jsou analogické (avšak jsou samozřejmě ještě obtížněji detekovatelné).

Příklad:

Thready *T1* a *T2* používají dvou prostředků *A* a *B*, jež jsou chráněny dvojicí mutexů *a*, *b* a jsou implementovány podle výpisu na následující straně.

Předpokládejme následující dva souběhy:

příznivý souběh		nepříznivý souběh	
T1	**T2**	**T1**	**T2**
a.lock		a.lock	
b.lock		$\longrightarrow$	b.lock
$\longrightarrow$	b.lock		a.lock
...	$\Longleftarrow$	b.lock	$\Longleftarrow$
b.unlock		**uváznutí!**	
b.unlock			
...			
$\longrightarrow$	...		
	a.lock		
	...		
	a.unlock		
	b.unlock		

Zatímco při prvním souběhu proběhne vše v pořádku, dojde ve druhém souběhu k uváznutí obou vláken, neboť první vlákno čeká na mutex *b* vlastněný druhým vláknem a naopak vlákno T2 na mutex *a* vlastněný vláknem prvním. Pokud existují další vlákna užívající alespoň jeden z dotčených semaforů uváznou také (při pokusu o zamknutí semaforu). Podrobnější rozbor souběhu klíčových kódů by ukázal, že většina souběhů je příznivých (podíl příznivých roste např. při prodlužování kódu v kritických sekcích), avšak nebezpečí uváznutí stále zůstává (pravděpodobnost uváznutí se blíží limitně nule, tím se však zároveň snižuje i pravděpodobnost jeho odhalení).

```
}
```

Listing 4.2: vzájemné uváznutí dvou threadů

```
API::mutex a, b;

void * thread1 (void*);
void * thread2 (void*);

API::thread T1(thread1), T2(thread2);

void * thread1 (void *)
{
  while(1)
    {
      a.lock();
      b.lock();

      //... ���u¾vn ��prostedk A,B

      b.unlock();
      a.unlock();
    }
}

void * thread2 (void *)
{
  while(1)
    {
      b.lock();
      a.lock();

      //... ���u¾vn ��prostedk A,B

      a.unlock();
      b.unlock();
    }
}

int main ()
{
  T1.start();
  T2.start();

}
```

Uváznutí jsou nepříjemným problémem v každém víceúlohovém systému, dlouhou dobu však byla omezena na oblast systémového programování. Rozšíření nástrojů programování paralelních a distribuovaných programů úrovni aplikačních programů však tuto problematiku přesunulo i do sféry povinných znalostí aplikačních programátorů. Z tohoto důvodu uvádím alespoň ve zkratce základní metody, jak deadlockům zabránit.

4.3.1 eliminace uváznutí

Nejjednodušší metodou eliminace vzniku uváznutí je vyloučení překryvu kritických kódů (tj. v každém místě programu je kritický kód nejvýše jednoho prostředku). Tuto podmínku lze splnit pouze tehdy, není-li vyžadována přímá spolupráce několika prostředků (například při kopírování dat apod.). V mnoha případech však není přímá spolupráce nutná (často je tomu při jednosměrném kopírování), nemluvě již o překryvech vzniklých zbytečně širokým vymezením kritického kódu.

V ukázce uváznutí byly oba prostředky (resp. jejich mutexy) získávány ve *threadu* T1 v opačném pořadí oproti *threadu* T2. To není náhoda, protože pokud by byly získávány ve stejném pořadí, k uváznutí by nikdy nedošlo. Matematickou indukcí lze dokázat, že by k tomu nedošlo ani v případě většího počtu prostředků, pokud by tyto byly alokovány ve stejném pořadí. Pokud tedy existuje uspořádání prostředků (resp. jejich mutexů), lze uváznutí zabránit správným pořadím jejich alokace.

Avšak ani tato strategie není bez nedostatků. Prvním nedostatkem je neexistence úplného, obecného a zároveň přirozeného uspořádání prostředků. Naštěstí lze pro důležité podmnožiny prostředků zavést přijatelné a snadno použitelné uspořádání ve formě lexikografického uspořádání jejich identifikátorů (lze jej použít například pro mutexy v multithreadovém prostředí).

Závažnějším nedostatkem je skutečnost, že vynucené pořadí alokací může vést k větší časové závislosti procesů, což může vést k výraznému zpomalení běhu dotčených aplikací (jinak řečeno aplikace musí na sebe čekat, i když to není z charakteru jejich interakce již nutné).

Na obrázku je prostředek A užíván výrazně kratší dobu než prostředek B (např. je-li B nutno zdlouhavě inicializovat). Z důvodu prevence uváznutí však musí být prostředek

A alokován před prostředkem B, čímž se výrazně prodlouží doba, po níž je prostředek A exklusivně vlastněn daným procesem, což může ve svém důsledku výrazně zpomalit ostatní procesy. Zpomalení je výrazné, pokud prostředek A je užíván mnoha procesy či thready.

Uváznutí lze zabránit i aplikací přístupu *vše nebo nic*, tj. proces se pokusí získat všechny prostředky, které potřebuje, a pokud se mu to nepodaří, všechny prozatím získané uvolní. Aby však tento přístup fungoval, musí být celý proces alokace proveden atomicky, tj. musí mít podporu v jádře (je podporován např. u IPC semaforů POSIXu, u nichž lze provádět atomické operace nad množinou semaforů). Bohužel, i když tento přístup skutečnému uváznutí zabrání, mohou vznikat stavy, kdy jsou procesy zablokovány na dobu, jež je teoreticky konečná, ale může být značně dlouhá (hodiny, dny atd.). Předpokládejme, například že proces vyžaduje dva prostředky. Nejdříve [počátek atomické sekce] testuje semafor prvního a zjistí, že je volný a tudíž tento prostředek alokuje. Následně testuje druhý semafor a zjistí, že prostředek je již obsazen. Uvolní tudíž první semafor a čeká na druhém [konec atomické sekce]. Po probuzení (uvolnění druhého prostředku) jej alokuje a opětovně testuje první semafor, pokud je volný má vyhráno, jinak uvolní (signalizuje) druhý semafor a čeká na prvním (a tak dále). Tj. při nepříznivém souběhu může služba stále přecházet mezi semafory bez šance na úspěšné ukončení služby. Při větším počtu prostředků pravděpodobnost úspěšného dokončení relativně rychle klesá.

Další možností je detekce potenciálního *deadlocku* ještě před jeho vznikem. Bohužel algoritmy, které jsou schopny potenciální uváznutí odhalit, jsou buď velmi pomalé (vzrůstá režie operačního systému) nebo příliš defenzivní (situace, v nichž hrozí potenciální *deadlock*, vedou jen ve velmi malém počtu případů ke skutečnému uváznutí, algoritmus je přesto musí vždy vyloučit). Nejznámějším algoritmem tohoto typu je tzv. bankéřův algoritmus, který sice zabrání vzniku uváznutí, je však výrazně defenzivní (tj. bankéř, jenž by se řídil tímto algoritmem, by sice nezbankrotoval, ale trh by trpěl nedostatkem volných finančních prostředků).

V některých případech lze uváznutí zabránit vskutku revolučním řešením, a to **úplným zrušením synchronizace**.

To lze učinit dvěma cestami:

A) eliminací sdílených prostředků kromě (interně synchronizovaného) prostředku pro výměnu zpráv — eliminována je především sdílená paměť. I zde však může dojít k uváznutí např. nekonečným čekáním na odpověď (na zprávu, kterou jsem nevyslali)

Toto řešení nabízí mnoho distribuovaných platforem. Klasickou (a relativně snadno použitelnou) je platforma jazyka Erlang.

CESARINI, Francesco a Simon THOMPSON. Erlang programming. 1st ed. Cambridge [Mass.]: O'Reilly, 2009, xxi, 470 p. ISBN 0596518188.

B) připuštěním kolizí a jejich následnou detekcí

Kolize, jež jsou v tomto případě nevyhnutelné, musí být dokonale detekovatelné (tj. systém musí rozborem dat či stavů systému kolizi detekovat bezprostředně po jejím vzniku) a především musí existovat možnost návratu do stavu před vznikem kolize a opětovného restartování procesů (kolize je odstraněna návratem do původního nenarušeného stavu a opětným obnovením běhu procesů). Požadavek na identifikaci kolize a především na schopnost návratu do předchozího stavu je dosti omezující, přesto však nebránící praktickému uplatnění této myšlenky (nejznámějším příkladem, jež však již leží mimo oblast OS, jsou transakce v relačních databázích). Odstranění synchronizace je výhodné především v těch případech, kdy je pravděpodobnost vzniku uváznutí malá (obnova je pomalá,

děje se však opravdu výjimečně), u silně zatížených systémů je spíše kontraproduktivní (systém tráví většinu času obnovováním původního stavu).

4.4 Základní komunikační prostředky

Současné operační systémy nabízejí celou škálu prostředků umožňujících přesun dat mezi jednotlivými procesy (či vlákny, pokud není vhodná výměna přes sdílený datový region). Tyto prostředky se liší svou rychlostí, způsobem použití i dostupností na jednotlivých platformách. Z tohoto důvodu je zde podán pouze základní přehled nejdůležitějších a nejtypičtějších komunikačních prostředků a základních mechanismů jejich použití.

4.4.1 klasifikace komunikačních prostředků:

spoluúčast jádra:

- jádro se komunikace neúčastní (pouze sdílená paměť)
- data proudí přes jádro, tj. jsou minimálně dvakrát kopírována (jednou z uživatelského adresového prostoru do bufferu uvnitř jádra, podruhé z bufferu do jiného adresového prostoru)

vnitřní struktura dat:

proudově orientované (datovody) — data tvoří jediný nečleněný proud (struktura může být dodána na úrovni aplikací)

zprávově orientované — data jsou organizována do zpráv (tj. hranice mezi zprávami jsou dány již operačním systémem)

směrování přenášených dat:

jednosměrné — data proudí z jednoho procesu do druhého

obousměrné — data proudí mezi dvěma procesy v obou směrech

všesměrové — data proudí z jednoho procesu k více procesům

dostředné — data proudí od více procesů k jedinému procesu

přenášený objem dat (resp. přenosová rychlost):

malý: bity až jednotky kilobytů za sekundu (fronty zpráv, signály)

střední: stovky kilobytů až desítky megabytů za sekundu (roury, sokety)

velké: stovky megabytů až gigabity za sekundu (sdílená paměť)

transparentnost použití

netransparentní (nízkoúrovňové) prostředky — aplikační programátor používá mechanismů, jež se neužívají v jednoduchých nekomunikujících aplikacích (tj. musí si být vědom meziprocesorové komunikace)

transparentní prostředky — pro komunikaci se používají stejné přístupy jako u entit v rámci jediného procesu (např. volání metod a získávání návratové hodnoty apod.)

4.4.2 roura

roura

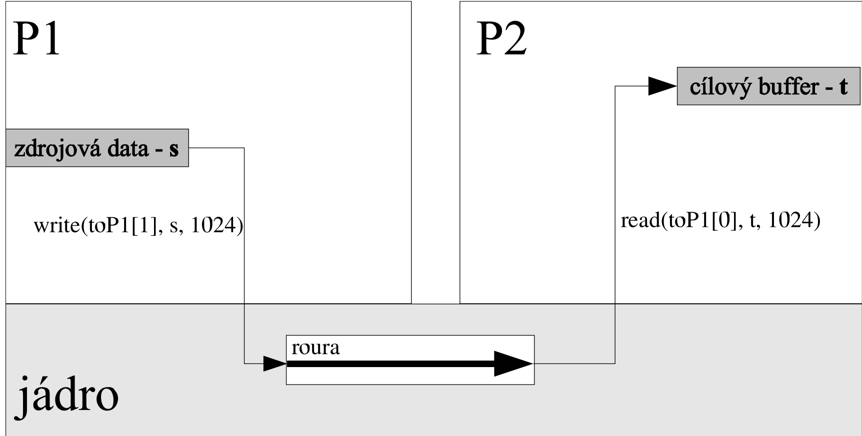
Roura (*pipe*) je klasickým komunikačním prostředkem operačního systému Unix (existuje však i v dalších OS). Klasická roura je jednosměrným datovodem vhodným především k výměně malých a středních objemů dat mezi dvěma procesy (v případě obousměrného přenosu je nutno použít dvojici rour) v rámci jediného počítače (přesněji jediné instance OS). Modernější implementace mohou být obousměrné resp. dokonce nemusí být omezené na jediný počítač (zde je však výhodnější použít sokety).

Implementace roury je jednoduchá, jedná se o (kruhovou) frontu o velikosti jednotek KB (v Linuxu jsou to 4 KB). Jediným rozdílem je těsné propojení rour se souborovým systémem. To umožňuje pracovat s rourou pomocí základních souborových operací, tj. zápis do roury a čtení se děje běžnými souborově orientovanými službami (včetně různých knihovních a jazykových nadstaveb). Jediným rozdílem je skutečnost, že s rourou jsou spojeny dva deskriptory otevřených souborů — jeden je určen pro zápis (vstup do datovodu), druhý pro čtení (výstup z datovodu).

Nad rourou existuje i jednoduchá synchronizace obdobná synchronizaci nad kruhovou frontou: zapisující proces (producent dat) se zablokuje, pokud je roura plná; čtoucí proces (konzument dat), pokud je prázdná.

V Unixu existují dva typy rour lišící se mechanismem své identifikace a tím i použitelností v různých situacích. Nepojmenované (anonymní) roury nemají vlastní identifikátor a jsou přístupné pouze prostřednictvím souborových deskriptorů (tj. stejně jako již otevřené soubory). Z tohoto důvodu nelze tyto roury použít pro komunikaci mezi libovolnými procesy, ale pouze mezi procesy, které jsou potomky procesu, jenž danou rouru vytvořil (včetně samotného procesu-tvůrce). Využívá se zde skutečnost, že unixovské procesy dědí otevřené soubory od svého rodičovského procesu (včetně otevřených konců anonymní roury). Nepojmenované roury se mimo jiné používají pro implementaci tzv. *kolon*, tj. příkazů spojených prostřednictvím rour do jediného datového proudu, jenž je těmito procesy filtrován resp. jinak modifikován. Příkladem budiž např. následující jednoduchá kolona: `who | grep '^fiser' | wc -l`, která za běhu využije dvou anonymních rour (na místě svislítek).

Použití anonymní roury v POSIXovském systému z hlediska aplikačního programátora v jazyce C ukazuje následující obrázek. Kopírování dat z jednoho procesu do druhého (resp. z jednoho LAP do druhého) se děje prostřednictvím jádra a to na straně zdrojové zápisem do otevřeného souboru (služba *write*), na straně druhé čtením (z jiného) otevřeného souboru (služba *read*).



Roury druhého typu jsou viditelné v souborovém systému (tj. mají jméno a umístění v adresářové struktuře) a mohou je tudíž užívat libovolné dva procesy v dané instanci OS. Pojmenované roury (také jsou označovány jako FIFO) je možno otvírat stejně jako běžné soubory (tj. pro čtení i zápis). Při otevírání však může dojít k zablokování procesu, neboť služba *open* čeká, jestliže není přítomen proces na druhém konci roury (tj. např. při otvírání pro zápis čeká, dokud se neobjeví proces čtenář tj. proces otvírající rouru pro čtení).

4.4.3 soket (schránka)

soket

Sokety (ang. *sockets*, česky *schránky* resp. přesněji *patice*) jsou obecné síťové komunikační prostředky, jež je možno implementovat nad většinou síťových protokolů, nejčastěji se však používají jako nadstavba protokolu TCP/IP (formálně by patřily do transakční vrstvy modelu OSI).

Sokety umožňují navázat dva typy spojení — datagramové, jež je zprávově orientované (neposkytuje však záruky ohledně úspěšného doručení či dokonce pořadí doručených zpráv) a proudové, jež nabízí obousměrný datovod obdobný (obousměrné) rouře.

Nejzajímavějším rysem soketů je mechanismus navázání spojení, který předpokládá jistou asymetrii mezi komunikujícími procesy a identifikace službou a nikoliv sdíleným identifikátorem (např. jménem souboru).

Soket jako obousměrný komunikační prostředek je určen pěticí údajů. První dvojici tvoří adresa počítače, na němž běží proces poskytující určitou službu, (v TCP/IP je to tzv. IP adresa) a port (port je přirozené číslo identifikující poskytovanou službu). Proces poskytující své služby na pevně určeném portu a čekající na požadavky od jiných procesů je označován jako server. Druhá dvojice údajů je tvořena údaji o procesu klienta, tj. žadatele a uživatele dané služby. Je to opět adresa počítače klienta (client host) a číslo portu. Číslo portu je však v tomto případě přiděleno prakticky náhodně a jen pro danou relaci (slouží pouze k případnému rozlišení více klientů na stejném počítači). Poslední údaj pětice určuje použitý nízkoúrovňový protokol, přičemž se téměř výhradně užívá buď protokolu TCP (proudové spojení) nebo UDP (datagramové spojení).

Navazování spojení začíná otevřením tzv. půlsoketu na straně serveru. Ten nabídne potenciální spojení na dané IP adrese (ta je rovna některé z IP adres stroje na němž server běží) a daném portu (a samozřejmě i protokolu). Jako číslo portu užije buď hodnotu registrovanou pro široce rozšířené služby (např. port 80 je registrován pro službu HTTP) nebo

libovolnou hodnotu u níž existuje dohoda s klientem. Půlsoket je tedy jakousi zdířkou, která čeká až se k ní připojí první klient (server je při tomto čekání zablokován).

Na druhé straně komunikace klient vyšle požadavek na spojení, který obsahuje identifikaci serveru (adresu, port, protokol). Pokud je na specifikovaném místě nalezen soket, jenž čeká na spojení (tzv. naslouchá), dojde ke vzniku spojení (relace). Zbytek údajů o spojení (adresa a port klienta) se získá ze strany klienta (port je zvolen náhodně z množiny volných portů na straně klienta).

Co se však stane, pokud se chce k serveru přihlásit další klient? Půlsoket je již obsazen a spojení by bylo nutno odmítnout (nepomohlo by ani nové otevření půlsoketu, neboť každý port může být užíván jen jedním procesem a v něm pouze jednou).

Implementace soketů řeší tento problém duplikací půlsoketu před vytvořením relace (komunikačního kanálu). Nově vytvořený (serverový) půlsoket mající jedinečné číslo portu (odlišné od základního portu služby) je navázán na klienta (tj. vytvoří s ním komunikační kanál). Původní soket na portu služby zůstává nepropojen a je k dispozici dalším klientům. Pro obsluhu nově vytvořené relace může vzniknout nový thread nebo proces (typické pro Unix).

> Obecně lze tedy říci, že navazování spojení u soketů je obdobou spojení přes spojovatelku, která propojí hovor dle požadavku na službu (tj. na jednotlivé odbory či osoby). Stejně tak je připravena přijímat další požadavky, i když daný odbor je právě v telefonickém kontaktu s jiným zákazníkem (samozřejmě nikoliv duplikací osob, ale přesměrováním hovoru na jinou odpovědnou osobu).

4.4.4 fronta zpráv

fronta zpráv

fronta zpráv (*message queue*) je elementárním zprávově orientovaným komunikačním prostředkem. Prostředek umožňuje cílenou i všesměrovou výměnu zpráv omezené velikosti mezi dvěma či více procesy. Každá zpráva se skládá z hlavičky, jež obsahuje základní informaci o zprávě (repertoár se liší u každé implementace, mezi nejčastější patří identifikace adresáta, priorita zprávy, délka zprávy, mezi další patří identifikace odesílatele, jednoznačná identifikace zprávy apod.), a těla zprávy (jednoduché implementace si vynucují pevnou velikost těla v řádu nejvýše několika bytů, složitější umožňují proměnnou velikost až do řádu KB).

Fronta zpráv zajišťuje směrování řazení zpráv (do front či prioritních front) i základní synchronizaci procesů-uživatelů. Vysílající proces se blokuje pokud je fronta plná (množství i celková velikost všech zpráv ve frontě je omezená), přijímající proces se blokuje, není-li ve frontě žádná použitelná zpráva (tj. je mu adresována resp. má dostatečnou prioritu).

Zprávy jsou nezbytné u operačních systému s architekturou klient-server (optimalizovaná podpora zpráv je přímo v mikrojádře), ale jsou užívány i v jiných systémech (i když často jen pro distribuci systémových zpráv s omezeným repertoárem a silně omezenou strukturou). Fronta zpráv je často užívána jako nízkoúrovňový základ složitějších a snadněji použitelných prostředků (např. klasické schránky známé z GUI nadstaveb). Navíc lze mechanismus fronty zpráv po mírném rozšíření daném složitější adresací použít i pro komunikaci procesů v počítačových sítích.

PŘEČTĚTE SI

Knihy *Operating Systems Internals and Design Principles* (Stallings) [12] se synchronizaci věnuje velmi zevrubně a to ve třech kapitolách:

4. Threads (částečně souvisí s problematikou procesů)

5. Mutual Exclusion and Synchronization

6. Concurrency: Deadlock and Starvation

OTÁZKY

1. Uveďte praktické příklady (ze života) kdy vzájemné vyloučení nestačí ke správné synchronizaci.
2. Jak lze representovat binární semafor pomocí obecného semaforu?
3. Proč je nutná implementace synchronizačních prostředků v jádře?
4. Uveďte příklady obecného semaforu v běžném životě?
5. Jak může vzniknout uváznutí u synchronizačních událostí?
6. Jaký je základní rozdíl mezi rourou a soketem?
7. Proč je v systému POSIX při vytváření synchronizační události předán odkaz na mutex?

OTÁZKY K ZAMYŠLENÍ

1. V ukázce implementace binárního semaforu se předpokládá využití fronty procesů čekajících na odblokování (tj. na signál). Jak se implementace změní, užívá-li operační systém strategii probuďte se a předbíhejte? (viz kapitola 3.3.4).
2. Pět domorodců sedí kolem hranice dřeva a mezi nimi leží pět křesacích kamenů. Domorodec potřebuje dva kameny, aby mohl zapálit oheň (a dosáhne přirozeně jen na sousední kameny). Zamyslete se, k jakým typům uváznutí může za této situace docházet. Navrhněte synchronizační algoritmus bránící uváznutí a zapište pomocí pseudokódu.
3. Při otevírání dvojice jednosměrných pojmenovaných rour pro zajištění obousměrné komunikace může dojít k uváznutí. Vysvětlete, jak k němu může dojít (tj. za jakých souběhů) a jak mu lze zabránit.
4. Mechanismus zpráv lze použít i jako synchronizační prostředek. Navrhněte, jak pomocí fronty zpráv implementovat mutex. (nápověda: je nutný specializovaný proces)
5. Prověřte že k uváznutí nedojde, pokud se prostředky alokují v obou procesech ve stejném pořadí.
6. Jaký efekt by mělo prodloužení vyrovnávací paměti roury?

5 Souborový systém

CÍLE KAPITOLY

Kapitola věnovaná souborovým systémů se od ostatních kapitol odlišuje tím, že se zaměřuje na jediný operační systém — Unix. To umožňuje presentovat jednoduchý, ale přitom elegantní návrh, který sice vznikl před několika desetiletími, ale stále je využíván nejen v Unixu, ale v různých modifikacích i na ostatních platformách.

Návrh unixovského systému umožňuje dokonale prezentovat hierarchický model s jasně definovanými vrstvami, jehož rozhraní nejvyšší úrovně nabízí jednoduchý model — jediný strom, jehož listy jsou datové soubory. Vaším cílem by tak nemělo být jen poznání unixovského systému, ale výhod jednoduchého hierarchického návrhu.

5.1 Souborový systém — úložiště persistentních dat

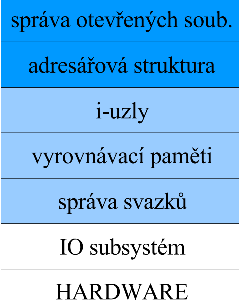
souborový systém

Souborový systém (*file system*, FS) poskytuje úložiště persistentních dat a to jak pro operační systém, tak pro aplikace v uživatelském prostoru. Typickým rysem souborového systému je uložení dat v pojmenovaných souborech.

Souborový systém není nezbytným subsystémem operačního systému, existuje však jen velmi málo systémů bez jeho podpory. Důvodem je skutečnost, že pro běh operačního systému je nutné persistentní (trvalé) uložení dat (kódu aplikačních programů i dokumentů, jež jsou jimi vytvořeny a zpracovávány). U většiny OS lze dokonce říci, že jsou tvořeny souborovým systémem, jenž je pomalu mění a vyvíjí prostřednictvím procesů.

Klasickým souborovým systémem je souborový systém Unixu, který se stal vzorem pro většinu současných operačních systémů (některé jako MS-DOS či nověji MS Windows poskytují systém, jež se od unixovského liší navenek jen v detailech). Z tohoto důvodu se tato kapitola věnuje souborovému systému, jež je součástí klasického Unixu a jež je i po 30 letech bez větších změn ve světě Unixu stále užíván.

Vertikální strukturu unixovského souborového systému lze vyjádřit následujícím schématem (schéma platí v hrubých rysech i pro většinu současných OS):

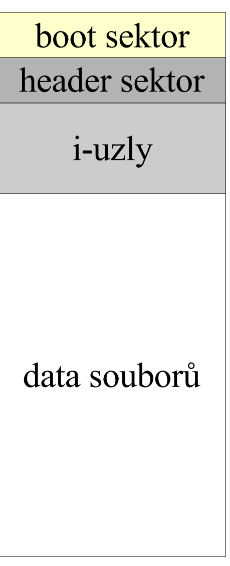


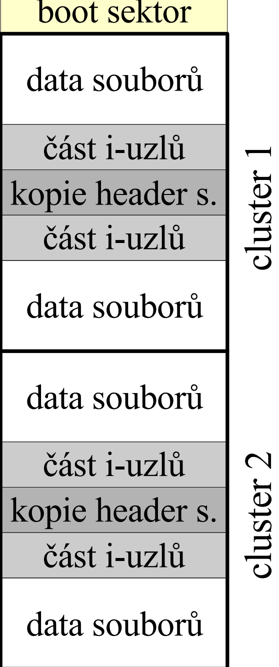
5.2 Svazek

Nejnižší vrstvu souborového systému tvoří správa svazků, která se stará o fyzické uložení souborového systému na blokovém zařízení. Souborový systém lze vybudovat na libovolném zařízení s blokovou strukturou (tj. zařízení je tvořeno bloky stejné velikosti) a náhodným přístupem (v každý okamžik lze přistupovat ke všem blokům v konstantním čase). Mezi tyto zařízení patří kromě různých typů disků i různé elektronické paměti (kromě flash pamětí to může být i region operační paměti). I když se adresovací strategie jednotlivých zařízení mohou výrazně lišit (často odpovídají fyzické geometrii daného zařízení), nabízí IO subsystém vyšším vrstvám tzv. logické disky, jejichž bloky (stejné velikosti) jsou adresovány lineárně (tj. bloky lze adresovat čísly $0, 1 \dots n-1$, kde n je počet bloků).

svazek

Na logickém disku vytváří správa svazku další logický prostředek — **svazek** (*volume*). Ten je tvořen několika sekcemi, z nichž některá obsahují globální informace o dané instanci souborového systému, jiné metadata o uložených souborech a dalších dílčích prostředcích a přirozeně i vlastní data souborů (a dalších prostředků). Struktura svazku se u jednotlivých typů souborových systémů může lišit, ale v zásadě existují dva přístupy k rozdělení disku.





Starší (a jednodušší) přístup rozděluje logický disk na několik málo sektorů. Boot sektor, který zaujímá jen několik málo prvních bloků není de facto součástí svazku, neboť není využíván operačním systémem, ale zavaděčem operačních systémů (boot managerem). Z hlediska souborového systému jsou to jen alokované, ale jinak nevyužívané bloky logického disku.

Další sekce označovaná v Unixu jako header sektor, obsahuje globální metadata celého souborového systému. Mezi ně patří kromě tzv. magického čísla (určuje typ souborového systému) i jedinečný identifikátor dané instance svazku. Mezi další globální metadata patří informace o obsazení svazku, o volných blocích resp. další statistické údaje. Header sektor bývá relativně malý (maximálně desítky bloků).

Další sekce obsahuje metadata souborů, v Unixu je to tzv. tabulka i-uzlů (v případě FAT systému jsou to FAT tabulky). Velikost této sekce je pevná (tj. např. v Unixu nelze přidávat další uzly po vyčerpání tabulky alokované při vytváření disku).

Poslední (a největší sekce) obsahuje vlastní data souborů a dalších datových prostředků na disku. I zde se však mohou vyskytovat metadata (tabulky odkazů, rozšiřující atributy apod.)

Toto jednoduché rozdělení má několik nevýhod:

- metadata (včetně těch nejkritičtějších) jsou uložena jen jednou (pokud jsou poškozeny, stává se celý svazek nepřístupným)
- při přístupu k datům uloženým souborem je nutno číst i jeho metadata (která jsou uložena v jiné a často velmi vzdálené sekci), to znamená že u mechanických pevných disků může docházet k častému pohybu hlaviček (což výrazně prodlužuje přístupové doby)

Modernější systémy proto využívají schémata, v nichž jsou klíčová data duplikována a především jsou na disku rozmístěna rovnoměrněji (a pokud možno blíže datům, která popisují). Obrázek ukazuje typicku strukturu modernějších verzí unixovského souborového systému (velmi podobný přístup užívá i ext2 systém). Disk je primárně rozdělen na tzv. cylindrické skupiny (clustery), v nichž je obsažena kopie superbloku (ty jsou v zásadě identické) a částí tabulky i-uzlů (která se primárně vztahuje na soubory ležící v rámci dané skupiny, ale podporovány jsou i složitější vztahy). Bloky souborů jsou primárně alokovány v nejbližším okolí příslušných metadat (jen u rozsáhlejších souborů se volí jiné strategie).

Tato struktura je nejvhodnější pro cylindrické disky (s hlavičkami na na několika plotnách) a s jednoduchým adresováním založeným na geometrii disku. V současnosti se však často používají různé virtuální geometrie resp. disky bez cylindrického uspořádání (např. flash disky) a tak se výhody tohoto uspořádání poněkud stírají (zůstává pouze výhoda duplicity dat).

5.3 Vyrovnávací paměti

Mechanismus vyrovnávacích pamětí nad blokovými zařízeními, je ve skutečnosti spíše součástí vstupně výstupního subsystému je však úzce provázán se souborovým systémem (především na úrovni svazků) a je proto uveden v této kapitole.

vyrovnávací paměť

Vyrovnávací paměti (*buffery*) jsou paměťové bloky v operační paměti (nespravované správcem paměti), které zrcadlí obsah fyzických bloků na blokových zařízeních (vněj-

ších pamětech). Vyrovnávací paměti obsahují nepersistentní (= netrvalou) kopii dat na vnejších pamětech.

Vyrovnávací paměti mají tři základní funkce (v pořadí podle důležitosti):

tvoří sdílenou paměť mezi jádrem a asynchronním čtením a zápisem na disk (ten je zajišťován zvláštním koprocesorem, resp. DMA řadičem). Jedná se o speciální případ vztahu producenta (asynchronní přenos bloku dat se zařízení do vyrovnávací paměti, zápis programu do vyrovnávací paměti) a konzumenta (asynchronní přenos dat do zařízení resp. čtená dat z vyrovnávací paměti procesem). jedná se o skutečně asynchrooní činnosti (tj. svou podstatou paralelní a nesynchronizované) a musí tedy být synchronizovány explicitně (proto se u vyrovnávacích pamětí setkáváme s různými zámky)

sjednocují rozhraní k blokovým zařízením. Zatímco na vrstvě hardwaru, existují různě velké fyzické bloky a různé typy jejich adresování (které nemusí být ani lineární), očekávají vyšší vrstvy logický disk s jednotnými bloky (např. v Unixu 1KB) a lineárním adresováním. Vyrovnávací paměti právě takové rozhraní vyšším vrstvám nabízejí.

urychlení přístupu na disk. Pokud vyrovnávací paměť používají lenivý zápis, pak může podobně jako mapování souboru do virtuálního adresového prostoru urychlit přístup k disku. Většina čtení a zápisů se totiž děje do vyrovnávacích paměti (které jsou umístěny v paměti operační) a nikoliv do pomalých vnějších (diskových pamětí). Přináší to však i jisté nevýhody, neboť se vždy nemusí shodovat obsah logického disku (jak ho vidí vyšší vrstvy OS resp. aplikace) a fyzické paměti (disku).

Při návrhu systému vyrovnávacích pamětí je možno vycházet ze čtyř základních principů:

princip konzistence: každý blok fyzického zařízení je zrcadlen v nejvýše jedné vyrovnávací paměti. Pokud by byla zrcadlena ve dvou (a více) vyrovnávacích pamětech mohlo by docházet ke ztrátě dat (tj. data zapsaná do vyrovnácí paměti by byla přepsána obsahem jiné vyrovnávací paměti)

princip shodnosti a sdílenosti: všechny vyrovnávací paměti jsou funkčně shodné a mohou být sdíleny všemi blokovými zařízeními v systému (tj. konkrétní vyrovnávací pamět může být použita během své existence prů různá bloková zařízení). Jinak řečeno existuje jen jediný globální fond vyrovnávacích pamětí.

princip konečných prostředků: vyrovnávacích paměti je jen omezené množství (a toto množství nestačí pro zrcadlení všech vnějších pamětíjinak by použití blokových zařízení téměř postrádalo smysl (i když je zde samozřejmě persistence dat)). Tento princip může mít ještě silnější podobu: konstatní počet vyrovnávacích pamětí (tj. vyrovnávací paměti jsou přiděleny již při zavádění OS a dále se nemění). Většina moderních systémů však fond vyrovnávacích pamětí dynamicky mění (paměť se rozděluje mezi správcem paměti a buffery dynamicky podle zatížení obou systémů)

princip lenivosti: zápis vyrovnávací paměti se odkládá tak dlouho jak je možno (tzv. odložený zápis). Existují i další strategie (např. zápis se provádí bezprostředně po změně vyrovnávací paměti tzv. *write-through*), které nejsou tak efektivní (ale bezpečnější).

Na základě těchto principů lze vytvořit několik různých implementací. Zde se podíváme na klasickou implementaci systému Unix (System V z poloviny 80. let, v zásadě však stále použitelnou).

V této implmentaci jsou jednotlivé vyrovnávací paměti složeny z hlavičky a vlastního bufferu (o velikosti 1KB). Hlavička obsahuje informace o právě zrcadleném bloku

(v podobě dvojice číslo-svazku : číslo bloku), informaci zda je vyrovnávací paměť právě uzamčena a především několik dvojic ukazatelů, které zařazují paměť do některého ze seznamů, ve kterých jsou vyrovnávací paměti organizovány (vyrovnávací paměť je alespoň v jednom, ale může být i ve více seznamech).

Nejjednodušší je ***seznam volných vyrov. pamětí*** tj. stránek které nezrcadlí žádný blok. Tento seznam tvoři jednoduše organizovaný fond volných vyrov. pamětí (např. frontu, ale není to podmínkou). Tento fond obsahuje na začátku všechny vyrovnávací paměti, a během běhu systému se postupně vyprazdňuje až se zcela vyčerpá, neboť za normálních situací se nedoplňuje (jednou přidělená vyrovnávací paměť se již neuvolňuje). Jedinou výjimkou je odmountování svazku (uvolní se všechny vyrovnávací paměti zrcadlící bloky z tohoto svazku) a rozšiřování globálního fondu.

Dalším (obousměrným) seznamem je ***fronta odemčených vyrovnávacích pamětí***. V ní jsou všechny přidělené (tj. nikoliv volné) vyrovnávací paměti, které právě nejsou uzamčeny. Uzamčení je relativně krátkodobé (vzhledem k životnosti vyrovnávacích pamětí) a proto je většina přidělených pamětí v této formě.

Poslední strukturou je ***otevřená hashovací tabulka všech přidělených pamětí***. Tato tabulka se skládá z většího počtu oddělených seznamů, které obsahují všechny vyrovnávací paměti pro něž hashovací funkce vrací stejnou hodnotu. Běžnou hashovací funkcí je modulo nad číslem bloku (bez ohledu na číslo svazku). Při modulu *n* je hashovací tabulka složena z n dílčích seznamů, přičemž v prvním jsou vyrovnávací paměti zrcadlící bloky s číslem dělitelným n (tj. i mod n je roven 0), ve druhém vyrovnávací paměti bloků, jejichž číslo je dělitelné n se zbytkem 1, a nakonec poslední (n-tý) obsahuje vyrovnávací paměti, u nichž dělením čísla jejich bloků vzniká zbytek $n - 1$. Funkce hashovací tabulky je zřejmá, umožňuje rychlé vyhledání vyrovnávací paměti pro daný blok (a svazek), pokud tato paměť existuje, resp. rychlé zjištění opaku (blok není aktuálně zrcadlen).

5.4 Vnitřní representace souborů

Další vrstva správce systému souborů v Unixu poskytuje klíčový logický prostředek — *soubor*. Soubor je na této úrovni representován nepříliš rozsáhlou datovou strukturou tzv. **i-uzlem** (*i-node*). Tato datová struktura vždy obsahuje základní atributy souboru jako je základní typ souboru, přístupová práva (v Unixu trojice práv pro tři skupiny uživatelů) a trojice časů (poslední modifikace – *modification time*, přístupu – *access time*, poslední změny i-uzlu – *change time*).

i-uzel

Další obsah se již liší dle typu souboru:

regulární soubory — datové soubory, jejichž data leží na logickém disku (svazku). I-uzel obsahuje odkazy na bloky obsahující data souboru (jež jsou přímo přístupná z aplikační úrovně)

adresáře — datové soubory, jež se na této úrovni neliší od regulárních souborů (obsah je opět uložen na disku a i-uzel obsahuje odkazy na bloky). Formát jejich dat je však pevně dán souborovým systémem a jsou použity jako základní konstrukční prvek pro budování hierarchické (adresářové) struktury na další úrovni správy souborového systému.

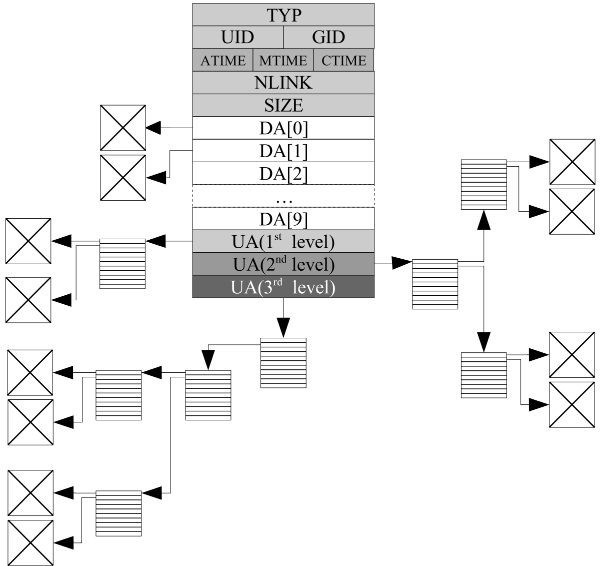
symbolické odkazy — datové soubory, jež se opět na této úrovni strukturálně neliší od regulárních souborů (jejich data jsou uložena na svazku). Hrají však pomocnou roli v adresářové struktuře.

bloková a znaková zařízení — pseudosoubory, obsahující odkazy na zařízení spravovaná vstupně-výstupním subsystémem. Neobsahují odkazy na datové bloky, ale interní identifikaci zařízení (tzv. čísla major a minor). Nejedná se tedy o skutečné soubory, ale pouze o odkazy, jež využívají jmenný prostor poskytovaný jednotnou adresářovou strukturu Unixu (v ní sídlí vesměs v adresáři /dev).

pojmenované roury (FIFO) a lokální sokety — opět pseudosoubory bez datové reprezentace, jež pouze využívají jmenný prostor adresářové struktury. Obsahují odkazy na lokální komunikační prostředky.

Jednotlivé i-uzly byly v klasickém Unixu uloženy na svazku v jediné (a souvislé) tabulce na počátku svazku a byly identifikovány svou pozicí (indexem) v tabulce tzv. ***i-číslem*** (i-number). V modernějších unixovských souborových systémech nemusí být uspořádání uzlů tak jednoduché (mohou být například rozmístěny v jednotlivých shlucích [clusterech]), ale vždy jsou opatřeny jedinečným *i-číslem*, které je jedinou identifikací souboru na této úrovni. I-uzly jsou však lokální strukturou svazku, to jest i-číslo soubor jedinečně určuje pouze v rámci svazku (na jiném svazku může být další soubor se stejným i-číslem). Při použití více svazků je nutno soubor identifikovat uspořádanou dvojicí (číslo svazku, i-číslo).

Typickou strukturu i-uzlu datového souboru ukazuje následující obrázek. První část i-uzlu je tvořena atributy. Kromě identifikace základního typu [TYPE] a výše uvedených obecných atributů (práva - [PERM], časy [MTIME, ATIME, CTIME]) obsahuje i-uzel datových souborů i velikost souboru ([SIZE] 32-bitová hodnota, dnes často i 64 bitová).



Nejdůležitější součástí i-uzlu datového souboru jsou však odkazy na datové bloky. Problematickým místem libovolného mechanismu odkazů na datové bloky jsou velmi různorodé požadavky na velikost souborů (od jednotek kilobytů po gigabyty) spolu s požadavkem na konstantní dobu přístupu k jednotlivým částem souboru na straně jedné resp. omezená (nebo dokonce konstantní) velikost i-uzlů (resp. jejich obdob) na straně druhé. Klasický Unix řeší tento problém použitím složeného adresování, jež využívá různého stupně nepřímého adresování pro přístup k souborům různé velikosti.

Klasické unixovské i-uzly používají pro malé soubory množiny přímých odkazů (typicky 10 odkazů). Při velikosti logického bloku 1024 bytů lze však přímými odkazy adresovat pouze soubory do velikosti 10KB. K libovolnému bytu u takto malých souborů však lze

přistupovat za cenu maximálně dvou přístupů na disk (ve skutečnosti je při použití vyrovnávacích pamětí průměrný počet přístupů mnohem menší). První přístup je nutný pro načtení obsahu i-uzlu (tento přístup je de facto eleminován použitím paměťových kopií i-uzlů u všech otevřených souborů, viz dále), druhý již přímo přistupuje k datovému bloku.

U rozsáhlejších souborů je nutné použít nepřímého adresování. U prvého stupně nepřímého adresování obsahuje i-uzel jediný odkaz na datový blok, který je interpretován jako tabulka odkazů na vlastní datové bloky souboru. Za předpokladu velikosti bloku 1024 bytů a 32-bitovém odkazu na bloky lze adresovat soubory o velikosti až 256KB = $2^{18} = 2^8.2^{10}$(+10KB, neboť nižší adresy jsou adresovány přímo). Pro přístup k blokům nad hranicí 10KB je však nutno provést v nejhorším případě až tři přístupy k disku. Soubor však zaujímá na svazku kromě svých datových bloků navíc jeden blok pro tabulku odkazů prvé úrovně, což zvětší soubor v nejhorším případě o 9% (soubor o velikosti 10KB+1byte) v nejlepším však pouze necelé čtyři desetiny procenta (soubor o velikosti 266KB).

Ani to však není dostatečné, pro ještě rozsáhlejší soubory je nutno použít víceúrovňové nepřímé adresování. Dvouúrovňové nepřímé adresování dokáže adresovat soubory o velikosti 64MB = $2^{26} = 2^8.2^8.2^{10}$(+266KB adresované přímo a jednou úrovní). Paměťová režie není příliš velká (relativně v malých jednotkách promile), přístup je však pomalejší o další přístup k disku. Nepříznivý důsledek čtyřnásobného přístupu je však téměř eliminován jak geometrií svazku, tak a to v podstatné míře použitím vyrovnávacích pamětí.

Pro ještě rozsáhlejší soubory je využito adresování tříúrovňové. I když by tak bylo teoreticky možno adresovat soubory o velikosti až 16GB (= $2^{34} = 2^8.2^8.2^8.2^{10}$), je velikost souborů v klasickém Unixu omezena na 2GB (= 2^{31}), neboť velikost souboru je v i-uzlu uložena jako 31-bitová hodnota (jeden bit je vyhrazen pro záporné relativní adresy ve službě *lseek*).

Detailnější informace o i-uzlech a adresářové struktuře najdete v klasické knize.

BACH, Maurice J. *Principy operačního systému UNIX*. 1. vyd. Praha: Softwarové Aplikace a Systémy, 1993, 16, 514 s. ISBN 80-901507-0-5.

5.5 Paměťové i-uzly a VFS

I-uzly jsou primárně uloženy na disku (svazku). Z důvodů větší efektivity (a nejen jí) jsou však kopie i-uzlů právě otevřených souborů uložena v paměti (v adresovém prostoru). Tento tzv. paměťový i-uzel však není pouhou kopií diskového i-uzlu. Vždy například obsahuje číslo i-uzlu (*i-number*) a identifikaci svazku resp. příznak uzamčení, neboť paměťové i-uzly jsou stejně jako vyrovnávací paměti organizovány do seznamu (resp. jiné výhodnější topologie) a během čekání na dokončení diskových operací musí být uzamykány.

Kromě toho paměťový i-uzel obsahuje i čítač otevření, neboť jediný soubor (i-uzel) může být otevřen v několika různých procesech (výjimečně i vícekrát v jediném procesu). Kdy k této situaci může dojít?

1. několik procesů nezávisle na sobě otevře stejný soubor (nejčastěji pro čtení, při zápisu může dojít ke kolizi, viz první díl skripta)
2. procesy využívají soubor jako sdílený (např. komunikační) prostředek

3. je spuštěno více instancí aplikace (při spuštění procesuje odpovídající spustitelný soubor vždy znovu otevřen)
4. při namapování sdílené knihovny do vícero procesů (běžný stav)
5. adresář (to je také na této úrovni soubor) je pracovním adresářem některého z procesů nebo je místem připojení

Čítač otevření je kontrolován při každém uzavření souboru (služba close nebo přesunutí procesu do stavu zombie resp. volání *exec*) a po poklesu čítače k nule je paměťový i-uzel uvolněn (diskový však přirozeně zůstane zachován). Čítač je důležitý také při výmazu souboru, neboť brání bezprostřednímu vymazání, pokud je soubor alespoň jednou otevřen (soubor je vymazán až poté co jej uzavře poslední proces).

Důležitou roli hraje dynamický i-uzel u Unixů s podporu vícero typů souborových systémů (svazků). Právě na úrovni paměťového i-uzlu je implementován tzv. VFS (*Virtual File System*), jenž skrývá před vyššími vrstvami detaily návrhu jednotlivých souborových systémů.

Paměťový uzel ve virtuálním souborovém systému ve své struktuře napodobuje klasickou strukturu unixovského i-uzlu vyjma odkazů na bloky. Paměťový i-uzel ve VFS (dále jen virtuální i-uzel) však nevzniká kopírováním diskového i-uzlu (jenž přirozeně u mnoha typů souborových systémů ani neexistuje), ale transformací odpovídajících struktur nativní interní representace souboru. Pokud tato struktura požadovanou informaci neposkytuje je nahrazena implicitní hodnotou (ta je většinou sdílena všemi soubory na svazku) nebo je odvozena z hodnot dostupných.

Následující ukázka výpisu příkazu *stat* aplikovaném na soubor v souborovém systému VFAT ukazuje tento přístup v praxi:

```
File: `/c/xxx'  Size: 3561     Blocks: 8   IO Block: 4096
Regular File
Device: 301h/769d       Inode: 88          Links: 1
Access: (0755/-rwxr-xr-x)  Uid: ( 0/  root)   Gid: (  0/  root)
Access: 2003-09-16 19:26:40.000000000 +0200
Modify: 2003-09-16 19:26:40.000000000 +0200
Change: 2003-09-16 19:26:40.000000000 +0200
```

Virtuální i-uzel v tomto případě obsahuje jen dva údaje převzaté přímo z adresářové struktury FAT — velikost a čas poslední modifikace (sekundy jsou vždy dělitelné dvěma). Ostatní údaje jsou buď implicitní (přístupová práva, vlastník a skupinový vlastník, počet odkazů) nebo jsou odvozeny z těch dostupných (ostatní časy jsou kopií času přístupu). Odvozeným údajem je i-číslo (označeno jako *Inode*), jenž může být libovolné avšak jedinečné (je odvozeno z umístění adresářového záznamu).

Jak bylo výše řečeno neobsahuje virtuální i-uzel univerzální sadu odkazů na bloky, neboť mechanismy odkazů se v jednotlivých souborových systémech výrazně liší. Každý jednotlivý podporovaný souborový systém místo toho používá vlastní specializovanou datovou strukturu, nad níž však implementuje jednotné rozhraní zajišťující překlad adres vztažených k souboru na adresy bloků. Vyšší vrstvy jsou tak odstíněny od implementačních detailů na úrovni jednotlivých souborových systémů a mohou používat jednotnou sadu funkcí pro přístup k virtuálnímu i-uzlu.

5.6 Adresářová struktura

Použití i-čísel spolu s označením svazku jednoznačně identifikuje soubory uložené na připojených svazcích, pro použití na uživatelské úrovni jsou však nepoužitelné, neboť se obtížně pamatují a potenciálně obrovský počet souborů nikterak nestrukturují.

Tvůrci Unixu proto nad i-uzly vybudovaly další vrstvu, jež poskytuje soubory opatřené srozumitelnými jmény (v původním Unixu omezené na 14 znaků) a především uspořádané do přehledné hierarchické podoby tzv. ***adresářové struktury***. Adresářová struktura je jedním z největších přínosů Unixu (i když jisté aspekty adresářové struktury se objevily již dříve) a byla přejata do mnoha dalších operačních systémů (i když mnohdy s drobnými změnami).

adresář

Vlastní implementace adresářové je překvapivě jednoduchá. Jejím základem jsou tzv. **adresáře**, což jsou v Unixu speciální soubory. Od běžného (regulárního) datového souboru se adresář liší příznakem v i-uzlu (v položce typ je uveden konstanta příslušná typu adresář), především však specializovaným obsahem, jehož formát je pevně definován již na systémové úrovni.

V klasickém Unixu obsahuje adresář tabulku se dvěma sloupci, první sloupec obsahuje i-číslo (2byty), druhý jméno souboru (14bytů).

Jako příklad vezměme zjednodušený obsah kořenového adresáře:

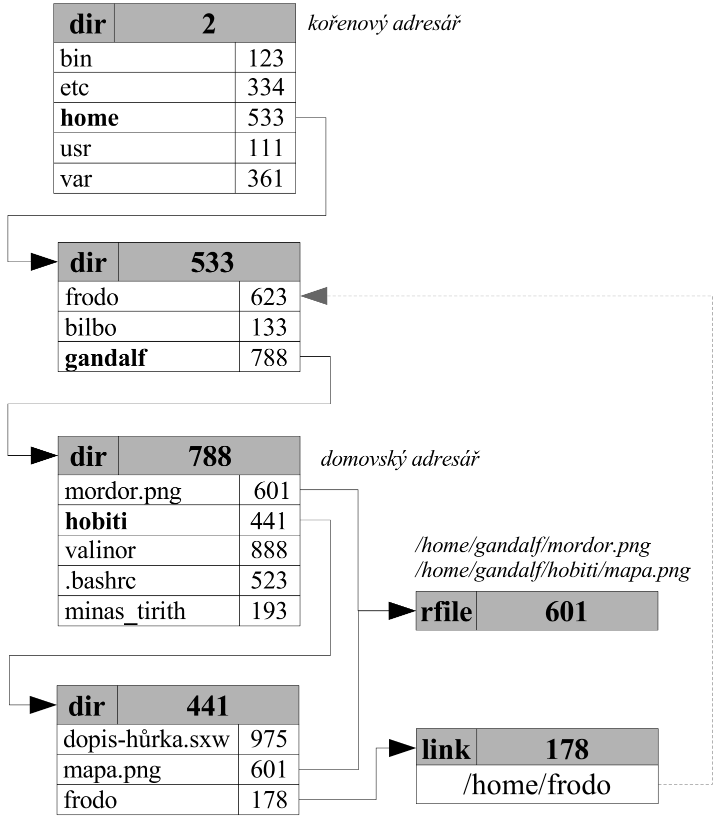
i-number	jméno souboru
2	.
2	..
10	bin
23	etc
5	usr

Prvotní funkce této adresářové tabulky je jednoduchá, mapuje jména souborů na čísla i-uzlů. Zde například určuje, že soubor (adresář) se jménem *bin* je representován i-uzlem s číslem 10. To však ještě nevytváří typickou stromovou strukturu adresářů. Podíváme-li se však na obsah adresáře *bin* (tj. do bloků odkazovaných i-uzlem 10) bude vše mnohem jasnější (opět výrazně zkráceno):

i-number	jméno souboru
10	.
2	..
54	cp
8	rm
21	zcat

Adresářová struktura je budována obousměrnými vzájemnými odkazy mezi adresářemi, rodičovský adresář obsahuje jméno adresáře podřízeného a jeho umístění (i-číslo), dětský adresář pak odkaz na adresář rodičovský (pod implicitním jménem „..“). Navíc každý adresář obsahuje odkaz na sebe sama (pod jménem „.“).

Jedinou výjimkou je adresář kořenový. Ten nemá rodiče, tj. neobsahuje na něj ani odkaz (obsahuje však položku „..“, jež však obsahuje odkaz na sebe sama). Navíc je vždy uložen v i-uzlu s pevným číslem 2 (pevné číslo je nutné, neboť jej nelze nalézt prohledáním rodičovského adresáře).



Z výše uvedené struktury adresářů vychází i základní algoritmus na této úrovni označovaný jako *namei* (***name** to **i**-node*), jenž převádí jména souborů na odpovídající i-čísla. Vstupem do tohoto algoritmu je jméno absolutní (=vztažené ke kořenovému adresáři) nebo relativní (=vztažené k pracovnímu adresáři, který si udržují nezávisle všechny procesy v systému). Předpokládejme např. že je potřeba převést absolutní jméno */bin/cp* na i-číslo, např. při otevírání souboru (jméno je parametrem služby *open*).

Algoritmus *namei("/bin/cp")*:

1. zjištění zda se jedná o cestu absolutní nebo relativní. V tomto případě začíná cesta lomítkem, tj. jméno je absolutní.
2. je otevřen soubor s i-číslem 2 (kořenový adresář) a lomítko je odstraněno
3. adresář je prohledán na výskyt jména *bin* (nyní na počátku cesty). V našem případě je nalezeno příslušná položka a je získáno odpovídající i-číslo 10.
4. kořenový adresář (i:2) je uzavřen
5. je otevřen soubor s i-číslem 10 (adresář *bin*) a je vyhledána položka obsahující jméno *cp*. Odpovídající i-číslo je 54.
6. je uzavřen adresář bin (i:10)
7. cesta je kompletně zpracována. Číslo 54 je vráceno jako výsledek (a je to skutečně i-číslo hledaného souboru)

Algoritmus *namei* funguje i pro zpětné odkazy (obsahující „..“) a po mírné modifikaci i pro relativní cesty. Počáteční odkaz na paměťový i-uzel pracovního adresáře je v tomto případě součástí každé položky tabulky procesů, je otevřen a na počátku přímo prohledáván (viz krok 3). Celý mechanismus však funguje pouze při použití jediného svazku (soubor je identifikován jen i-číslem nikoliv uspořádanou dvojicí s identifikací svazku). Tento problém je v Unixu řešen mechanismem připojovaní svazků, jenž bude vysvětlen později).

Základní podstata representace adresářové struktury a algoritmu *namei*, zůstává zachována i v současných Unixech (včetně Linuxu). Mezi změny patří 32-bitová i-čísla (současné systémy obsahují více než 64 tisíc souborů), delší jména souborů (± 256 znaků), sofistikovanější uspořádání záznamů adresářů (binární resp. digitální stromy apod.) a používání rychlých vyrovnávacích pamětí (cache) pro nejčastěji používané adresáře.

5.6.1 Odkazy (links)

Popis hierarchické souborové struktury prostřednictvím adresářových tabulek je založen na vícenásobném výskytu adresářových i-uzlů v tabulkách. Každý adresářový i-uzel je minimálně ve dvou tabulkách, ve své vlastní a v tabulce nadřízeného adresáře, další případné výskyty jsou v podadresářích.

Jaká situace však nastane pokud je vícenásobně odkazován i-uzel běžného souboru? Jeden fyzický soubor má v tomto případě dvě různá jména resp. je umístěn ve dvou různých (obecně nesouvisejících) adresářích (obecně se může objevovat pod různými jmény v několika různých adresářích).

pevný odkaz

Unix tyto situace připouští a prakticky využívá. Jednotlivé absolutní pozice souboru v adresářovém systému jsou označovány termínem **pevný odkaz** *(hard link)*. Tyto odkazy jsou zcela rovnocenné, to jest neexistuje žádný hlavní odkaz nebo hlavní jméno souboru. Při mazání souboru (jež se děje prostřednictvím jména) je nejdříve odstraněn odkaz (tj. řádek v adresářové tabulce) a pokud počet odkazů klesne k nule, je proveden výmaz fyzického souboru (tj. i-uzlu a případných datových bloků).

Pevné odkazy se dříve používaly například u souborů, jejichž obsah sdílelo více uživatelů a měli na něj tudíž odkaz ze svých domovských adresářů (resp. jejich podadresářů). Časté bylo jejich použití i u souborů užívaných několika aplikacemi, přičemž každá z nich hledala soubor pod jiným jménem (nevznikaly tak duplicitní kopie, resp. nebylo nutno používat konfigurační nástroje).

Pevné odkazy však mají jednu podstatnou nevýhodu, lze je používat pouze v rámci jediného svazku (adresářová tabulka obsahuje jen i-číslo ne číslo svazku). Pomocí mechanismu připojování svazků (viz následující podkapitola) lze sice dosáhnout vytvoření jediného virtuálního adresářového systému (s jediným kořenovým adresářem) a zakrýt tak rozdělení souborového systému na jednotlivé svazky před uživatelem, to se však netýká pevných odkazů. Uživatel, který je jinak od detailů ukládání dat odstíněn, nemůže pevné odkazy v některých případech vytvářet, nevěda proč.

symbolický odkaz

Z tohoto důvodu jsou dnes ve většině výše uvedených případů užívány tzv. **symbolické odkazy**, což jsou speciální soubory obsahující relativní nebo absolutní cesty, na nějž je přesměrováno otevření souboru (tj. místo odkazu je otevřen odkazovaný soubor). Symbolické odkazy se snadno používají, nejsou však již tak transparentní (lze rozlišit odkaz od původního jména souborů) a nejsou ani symetrické (smaže-li se odkaz, je smazán i odkazovaný soubor, v opačném případě zůstane odkaz-sirotek). Symbolické odkazy

jsou těsněji vázány na adresářovou strukturu, tj. nejsou omezeny na jediný svazek a lze se odkazovat pomocí relativního cesty (při kopírování podstromu, jenž obsahuje odkaz i jeho cíl zůstává relativní odkaz v platnosti).

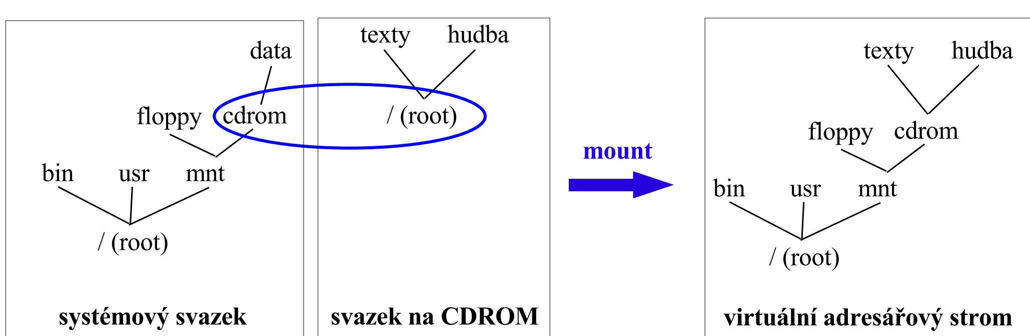
5.6.2 Připojování svazků (mounting)

Jak již bylo výše řečeno je pro Unix typické, že vytváří jediný virtuální adresářový systém, jenž vychází z jediného kořenového adresáře. Tento přístup, který byl použit již v nejstarších verzích Unixu zakrývá fyzické rozdělení svazků a vytváří jednotný pohled na souborový systém (bez nutnosti si pamatovat kryptická označení svazků). Aby však bylo možno dosáhnout tohoto stavu musí být adresářový model (a především algoritmus *namei* mírně rozšířen). Tento rozšiřující mechanismus se nazývá **připojování svazků** (nepřesně disků) anglicky *mounting* (mountování) a je ve své podstatě velmi jednoduchý.

připojování svazků

Mechanismus spočívá ve zdánlivém ztotožnění dvou adresářů (na dvou svazcích). Jeden adresář je tzv. přípojný bod (mount point) může ležet na libovolném svazku a není na tomto svazku kořenovým, druhý (připojovaný) musí být kořenovým adresářem (jiného) svazku. Po připojení se de facto zamění adresář – přípojný bod za připojovaný kořenový adresář. Tj. případný původní obsah přípojného bodu se stává neviditelným (není však ztracen je jen dočasně nedostupný) a místo něj se objeví obsah připojovaného adresáře (čímž se stane dostupným a stává se s celým adresářovým systémem připojovaného svazku

Vyžaduje pouze jediný (potenciálně jednobitový) příznak v i-uzlu adresáře – přípojného bodu, malou tabulku přípojných bodů a drobné rozšíření algoritmu *namei*.



Podívejme se například jak je modifikován algoritmus *namei* v případě použití absolutní cesty. Předpokládejme např. virtuální adresářový strom z obrázku (ten vznikl připojením svazku na CDROM v přípojném bodě */mnt/cdrom*) a cestu */mnt/cdrom/texty/g-hive.txt*. Počáteční kroky odpovídají klasickému algoritmu tj. je otevřen soubor kořenovího adresáře na systémovém svazku (číslo systémového svazku je uloženo v jádře, číslo i-uzlu je samozřejmě 2). Zde je vyhledána položka *mnt* a tak je zjištěno číslo i-uzlu dalšího adresáře. Ten je otevřen (leží stále na původním systémovém svazku). V něm se vyhledá i-číslo adresáře *cdrom* (je stále na systémovém svazku). Soubor tohoto adresáře však již není otevřen, neboť rutina při přístupu k i-uzlů, zjistí, že má nastaven příznak přípojného bodu (ten je nastaven při mountování). Proto dojde k vyhledání přípojného bodu v tabulce přípojných bodů (*mountpoint table*).

Tato tabulka má v zásadě jen tři sloupce: první obsahuje číslo svazku přípojného bodu (identifikuje svazek do nehož se připojuje), druhý číslo i-uzlu přípojného bodu (adresáře) a třetí číslo připojovaného svazku. Algoritmus *namei* zná v tomto okamžiku první

dvě hodnoty (první je číslo systémového svazku, druhé je v našem případě nalezené i-číslo adresáře *cdrom*). je proto snadné nalézt odpovídající číslo připojeného svazku. Nyní stačí otevřít i-uzel s číslem dva na tomto svazku (=kořenový adresář připojeného svazku) a algoritmus již pokračuje standardním způsobem (vyhledáním podadresáře *texty* a zjištěním jeho i-čísla), atd.

Mírně se modifikuje i algoritmus při zpětném průchodu stromem (v našem případě např. použití relativní cesty *../../floppy* v adresáři */mnt/cdrom/texty*), kdy se musí v přípojném bodě provézt traverz z připojeného do připojujícího svazku. Zde však stačí pouze pozměnit algoritmus při otevírání rodičovského adresáře u všech kořenových adresářů. Zatímco u algoritmu bez je moutování se otevře opět kořenový adresář (je sám sobě rodičem, což je uvedeno přímo v tabulce adresáře v řádce se jménem „..“), dojde v systému s mountováním k prohledání tabulky přípojných bodů (klíčem je zde číslo aktuálního svazku ve třetím sloupci) a následně k přesměrování na přípojný bod (první dva sloupce).

Mechanismus mountování v současných Unixech (včetně Linuxu) je mírně komplikovanější (např. z důvodů různých typů souborových systémů na svazcích) a umožňuje i připojení jiných než kořenových adresářů (je možné i vícenásobně připojit libovolný adresář na stejném svazku).

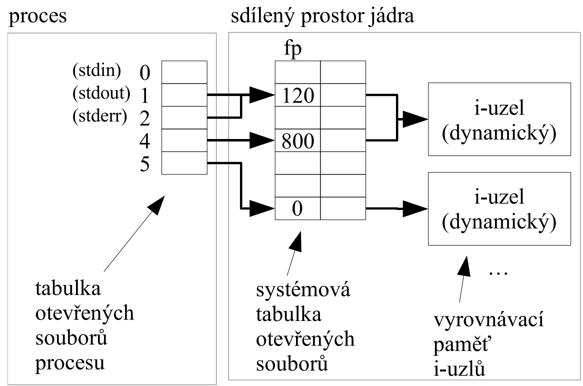
5.7 Otevřený soubor

Jak již bylo řečeno výše je soubor v Unixu representován tzv. i-uzlem tj. relativně malou strukturou, která je uložena na disku. To platí pro všechny soubory, ať jsou v daný okamžik využívány procesy či nikoliv. Pokud však chce proces přistupovat k datovému obsahu souboru musí soubor tzv. otevřít. Soubory, které jsou otevřeny alespoň jedním procesem (dále jen otevřené soubory) mají alokovány i další pomocné struktury, které přístup k souboru usnadňují a řídí.

Soubor je otevírán nejen při explicitní žádosti o jeho otevření, ale i v dalších případech:

1. spustitelný soubor je otevřen, pokud je vykonáván jako proces
2. dynamické knihovny jsou otevřeny, pokud jsou používány v alespoň jednom procesu
3. adresář je otevřen, pokud jej má některý z procesů jako aktuální domovský adresář
4. adresář je otevřen, pokud je kořenovým adresářem nějakého procesu (běžně to bývá skutečný kořenový adresář, ale může to být i jakýkoliv jiný adresář, pokud proces využil systémové volání *chroot*).
5. adresáře jsou krátkodobě otevírány v algoritmu *namei*

Nejnižší úroveň otevřeného souboru tvoří i-uzel ve formě dynamického i-uzlu (viz výše podkapitola 5.5 na straně 88). Dynamický i uzel je běžně vytvářen jen pro otevřené soubory.



Další úroveň tvoří tzv. systémová tabulka otevřených souborů (*system filetable*). Do této tabulky se přidá záznam při každém otevření souboru (v libovolném procesu). Tabulka obsahuje v zásadě dvě položky: ukazatel na příslušný dynamický i-uzel (je alokován jen při prvním otevření souboru, při dalším otevření se odkazuje na již existujícy i-uzel) a index aktuální pozice v souboru (*filepointer*). Index (méně přesně ukazatel) obsahuje absolutní pozici toho bytu v souboru nad nímž bude provedena následující operace čtení nebo zápisu. Po čtení nebo zápise se index automaticky posunuje (lze jej měnit i přímo prostřednictvím systémové služby *lseek*, ta je v knihovně jazyka C je index dostupná prostřednictvím funkcí *fseek*/*ftell*).

Poslední úroveň odkazů na otevřený soubor je *tabulka otevřených souborů procesu* (process filetable), která existuje v jádře pro každý proces. Položky této tabulky jsou opět zaplňovány při otevírání souborů, ale pouze v rámci daného procesu. Kromě několika méně významných atributů obsahují především referenci na příslušný záznam systémové tabulky otevřených souborů. Každý otevřený soubor procesu je jednoznačně identifikovatelný indexem v této tabulce. Toto malé přirozené číslo se označuje jako deskriptor souboru (a můžete se s ním setkat i na úrovni vyšších programovacích jazyků např. v C prostřednictvím funkce *fileno*).

Otázkou je proč je systém odkazů de facto čtyřúrovňový (program odkazuje deskriptorem do tabulky procesu, odkud vede reference do systémové tabulky a z ní na dynamický i-uzel, který teprve odkazuje na uzel statický, což je fyzický soubor). Jako zbytečná se jeví především systémová tabulka souborů (její funkci by mohla zastávat i tabulka v procesech).

Důvodem zdvojení odkazů je podpora dvou úrovní sdílení otevřených souborů mezi aplikacemi (procesy). Pokud dva procesy otevřou nezávisle na sobě stejný soubor, pak sdílejí pouze dynamický i-uzel a nikoliv záznam v systémové tabulce procesů. Oba procesy tak mají nezávislý index aktuální pozice v souboru a mohou na sobě nezávisle přistupovat k různým částem souboru (což si však na druhou stranu vyžaduje synchronizaci pomocí souborových zámků. Podobného efektu lze dosáhnout i v rámci jediného procesu (dvojnásobným voláním služby *open* nad stejným souborem).

Druhý typ sdílení se uplatňuje především při dědění otevřených deskriptorů souborů při vytváření procesů (tj. po volání systémové služby *fork*). V Unixu nový proces dědí tabulku otevřených souborů svého rodičovského procesu (a to i po zavolání služby exec, kdy začíná běžet podle nového spustitelného souboru). Oba procesy (rodič a jeho dítě) tak sdílejí soubory a toto sdílení je těsnější než tomu bylo u předchozího typu, neboť sdílen je i index aktuální pozice (v systémové tabulce). Pokud tedy např. jeden proces provede zápis či čtení souboru, pak je posunuta aktuální pozice i u druhého procesu. Tj. pokud jeden zapisuje sekvenčně (na konec souboru) a druhý taktéž, pak se ve výstupním

souboru střídají výstupy obou procesů a nikdy nedojde k žádnému přepsání a tím i ztrátě dat (výstupy však nemusí být snadno oddělitelné).

Tento princip se využívá např. pro výstup na terminál prostřednictvím standardního výstupu (resp. chybového výstupu). Vše začíná u shellu, který se spustí po přihlášení uživatele (v případě použití GUI terminálu pro každé okno terminálu) a otevře terminál jako speciální soubor (*/dev/tty*) jednou pro čtení (s deskriptorem 0) a podruhé pro zápis (s deskriptorem 1). Obě otevření jsou nezávislá (nesdílejí záznam v systémové tabulce). Pak provede duplikaci (=zkopírování) deskriptoru 1 na deskriptor 2 (oba deskriptory pak odkazují stejný záznam v systémové tabulce). Všechny tyto deskriptory mají v unixových aplikacích speciální význam, neboť deskriptor 0 je využíván jako standardní vstup, 1 je standardní výstup a 2 standardní chybový výstup. Standardní vstupy a výstupy jsou využívány v situacích, kdy chce program textově komunikovat s okolím (např. uživatelem) .

Ostatní interaktivní procesy jsou spouštěny ze shellu (ať již přímo či nepřímo např. u spouštění přes ikonu) a dědí tyto tří otevřené deskriptory a mohou je využívat ke své základní textové komunikaci. Ona výstupu jsou standadně do jediného souboru či terminálu, ale sdílení na úrovni systémové tabulky. tj. při výstupu nedochází k přepsání. To je zvlášť důležité pokud jsou oba výstupy přesměrovány do jediného souboru (přesměrování se děje opět na úrovni shellu a procesy přesměrování dědí).

OTÁZKY

1. Jaké jsou výhody přináší rozdělení dat svazků na clustery u pevných disků? Je výhodný i u flash pamětí (SSD disků?)
2. Jaká je hlavní funkce vyrovnávacích pamětí?
3. Proč se krátkodobě zamykají vyrovnávací paměti?
4. Proč je odkaz na otevřený soubor dvojúrovňový?

OTÁZKY K ZAMYŠLENÍ

1. Jak velkou diskovou paměť lze tříúrovňově adresovat při bloku o velikosti 4KiB, 32-bitové adrese?
2. Jak lze v Unixu/Linuxu poznat počet pevných odkazů na daný soubor? Jaké soubory mají největší počet pevných odkazů?
3. Jaká je nevýhoda původní representace adresářů pomocí tabulek o stejné délce řádků? Navrhněte lepší datovou strukturu.

ÚKOLY

1. Vypište v Unixu/Linuxu seznam všech přípojných bodů? (fstab, mount v čem se liší)
2. Vytvořte skript, který najde všechny pevné linky (= n-tice souborů se stejným číslem i-uzlu) ve vašem domovském adresáři? (s výjimkou adresářů)
3. Vyzkoušejte příkaz *namei*
4. Prostudujte dokumentaci příkazu *mount.*
5. Vypište počet souborů, který má otevřen daný proces (v Linuxu). (Rada: /proc/PID)

6 Subsystém vstupu a výstupu

CÍLE KAPITOLY

Subsystém vstupu a výstupu je klíčovou částí operačního systému, neboť se stará o správu všech zařízení kromě procesoru a paměti. Na druhou stranu je to velmi nízkoúrovňová část OS a aplikační programátor jej využívá jen pomocí rozhraní I/O služeb, Toto rozhraní je na jedné straně velmi jednoduché, na straně druhé závisí na konkrétním zařízení a zvolené platformě.

Z tohoto důvodu je hlavním cílem kapitoly jen zcela základní seznámení se subsystémem vstupu a výstupu (s důrazem na Unix) .Jedinou praktickou dovedností je identifikace jednotlivých zařízení a orientace ve struktuře jejich služeb. Z interní implementace se pozornost zaměřuje pouze na spolupráci horní a dolní poloviny ovladače včetně zvláštního typu jejich synchronizace. I když jsou to jen elementární informace, jsou základem různých mechanismů, které používají ovladače ve všech operačních systémech.

6.1 Bloková a znaková zařízení

Subsystém vstupu a výstupu zajišťuje komunikaci procesů s hardwarovými zařízeními (s výjimkou operační a paměti a procesoru o něž se stará správce paměti resp. správce procesů). Subsystém vstupu a výstupu využívají navíc i další moduly jádra, a to především modul souborového systému, ale může být využíván například i správcem paměti (pro přístup k odkládacími zařízení).

Současné OS podporují velké množství periferních zařízení, ale v návrhu OS mají z praktických (i historických) důvodů největší význam tři třídy periferních zařízení:

- *vnější paměťová zařízení s blokovým (náhodným) přístupem* (např. pevné a optické disky)
- *vstupně/výstupní zařízení se sekvenčním přístupem* (terminál, myš)
- *počítačová síť* (problematika pročítačových sítí je mimo zaměření kursu a těchto skript)

bloková zařízení

Zařízení s náhodným přístupem se tradičně označují jako **bloková**. Kromě náhodného přístupu je pro ně typické využívání rozsáhlých vyrovnávacích pamětí sdílených mezi všemi blokovými zařízeními (viz souborový systém).

znaková zařízení

Zařízení se sekvenčním přístupem se označují jako **znaková** (dnes by se spíše hodilo označení *sekvenční*). Tato zařízení sice také využívají vyrovnávací paměť, ale ta má často jen omezenou velikost (maximálně jednotky KiB) a musí existovat pro každé zařízení zvlášť.

Struktura subsystému vstupu a výstupu je relativně jednoduchá – skládá se s ovladačů (driverů), které sice mohou sdílet podstatné části kódu, ale jinak jsou na sobě relativně nezávislé (určitá hierarchie zde sice existuje je však dána spíše existencí různých virtuálních zařízení). Velmi obdobná je také základní struktura jednotlivých ovladačů (především na vyšší úrovni abstrakce).

6.2 Ovladače

ovladač

Ovladače (*driver*) leží mezi hardwarem a vyššími vrstvy operačního systému. Ve své nejvyšší vrstvě nabízejí relativně jednotné rozhraní vůči vyšším vrstvám operačního systému (s omezenou množinou služeb). Ve své nejnižší vrstvě přistupují přímo k hardwaru (tj. modifikují přímo registry I/O zařízení nebo koprocesorů). Tato nejnižší vrstva je součástí HAL (*hardware access layer*). Tato část ovladačů bývá často naprogramována v assembleru (ale nutné to není) a jako jediná je přímo závislá na zvoleném hardwaru. Zbylá část ovladače je však odstíněna od drobných rozdílů jednotlivých hardwarových zařízení (např. jednotlivých typů). Počet ovladačů v systému je tak udržován na rozumné úrovni (i když dnes se jedná spíše o stovky ovladačů, což je však velmi málo ve srovnání s desetitisíci různých hardwarových zařízení).

6.2.1 Rozhraní ovladačů

Na rozhraní ovladačů, které využívají vyšší vrstvy OS jsou kladeny dva protikladné požadavky. Ne jedné straně výrazné rozdíly ve funkčnosti jednotlivých I/O zařízení vyžadují specializovaná rozhraní přizpůsobená funkcím daného zařízení (porovnejme například zvukovou kartu a dotykovou obrazovku), na straně druhé by vyšším rozhraním vyhovovalo rozhraní co nejjednotnější a nejjednodušší (rozdíly mezi zařízeními tak nemohou expandovat do vyšších vrstev systému a komplikovat tak celý OS.

Z tohoto důvodu se vnější rozhraní ovladačů rozděluje do dvou částí. Základní rozhraní poskytuje pouze základní operace, které lze definovat u většiny ovladačů (i když vnitřní implementace těchto operací se může výrazně lišit). Pokud není základní služba ovladačem podporována musí místo ní nabídnou nic nevykonávající *maketu* (tzv. *dummy* funkci).

Druhá skupina – úzce specializované operace – se poskytuje prostřednictvím jediné speciální služby obecného rozhraní, která přijímá číselný parametr určující konkrétní funkci/službu (a popřípadě blížeji nespecifikované parametry této služby). Jinak řešeno specializované operace tvoří podslužby v rámci formálně unifikovaného rozhraní.

Mezi základní obecně podporované operace patří (názvy se v jednotlivých OS mohou mírně lišit):

init – počáteční inicializace zařízení. Provádí se pouze jednou buď při startu systému (u stále připojených zařízení) nebo při připojení zařízení (u plug&play zařízení).

terminate – odpojení zařízení. Může být voláno při ukončování běhu systému nebo při softwarovém odpojení zařízení. Většinou nemá žádnou speciální funkci (tj. je dummy)

open – otevření zařízení procesem. Zařízení se stává součástí jeho kontextu a často bývá vytvoření tzv. sezení nebo komunikačního kanálu. Může být voláno i vícekrát (i když jen některá zařízení podporují bezkolizní sdílení)

close – uzavření zařízení. Zařízení přestává být součástí kontextu a v případě potřeby je uzavřeno i případné sezení nebo komunikační kanál.

get – vrací informace o daném zařízení (v obecně použitelné formě)

read – u znakových zařízení čte byte nebo blok bytů ze (vstupního) zařízení

write – u blokových zařízenízapisuje byte nebo blok bytů do (výstupního) zařízení

strategy – u blokových zařízení zajišťuje zápis nebo čtení bloků. Zápis/čtení není provedeno bezprostředně, ale podle určité zvolené strategie (přes vyrovnávací paměti).

control – vstupní bod ke specializovaným podslužbám. V Unixu jsou tyto operace přístupné přes systémovou službu *ioctl*(2)[1].

6.2.2 Identifikace zařízení a ovladačů

čísla major a minor

Zařízení jsou v rámci operačního systému Unix identifikována dvoúrovňovým identifikátorem, který je tvořen dvěma čísly, které jsou označovány jako *major (number)* a *minor (number)*. Číslo major identifikuje ovladač a je to ve skutečnosti index do tabulky ovladačů. Tato tabulka obsahuje v jednotlivých řádcích odkazy na rutiny jednotlivých služeb daného ovladače (tj. z pohledu jazyka C ukazatele na funkce). Například v prvním sloupci mohou být odkazy na rutiny operace *init*, ve druhém *terminate* atd. Při volání určité operace nad zařízením tak stačí nalézt v řádku opovídajícím major číslu a sloupci odpovídajícím dané službě odkaz na rutinu, kterou lze následně zavolat.

Volané rutině se pak předá jako parametr číslo minor, které identifikuje zařízení v množině zařízení, které jsou aktuálně obsluhovány daným ovladačem. Je to pět ve skutečnosti index do tabulky zařízení, která je uložena v rámci daného ovladače (tato tabulka obsahuje údaje o fyzických adresách jednotlivých zařízení, jejich stavu apod.). Rutina tak může dané zařízení adresovat a může se i přizpůsobit jeho aktuálnímu stavu.

Použití číslených identifikátorů však není příliš vhodné pro uživatele na aplikační úrovni. Proto jsou (zcela v duchu Unixu) mapovány do adresového prostoru souborového systému. Jednotlivá zařízení (resp. uspořádané dvojice ovladač:zařízení) jsou totiž dostupné jako tzv. speciální soubory integrované do souborového systému (a jeho adresářové struktury). Navenek jsou to běžné soubory, do nichž lze zapisovat a/nebo číst, mohou má nastavena přístupová práva apod. Nejsou však fyzicky propojeny se žádným souborem souborového systému, ale jedná se pouze o odkazy na fyzická zařízení (která tak lze identifikovat).

Původně byly speciální soubory representovány i-uzly, které byly uloženy na fyzických svazcích. Od i-uzlů fyzických souborů se formálně odlišují uvedením jiného bitového příznaku, skutečný rozdíl však spočívá v chybějících odkazech na datové bloky (ty samozřejmě neexistují) a naopak v uvedení dvou číselných identifikátorů zařízení (major:minor). Mapování i-uzlů na jména v adresářové struktuře je shodné s běžnými soubory (viz algoritmus namei).

Zápis do znakového zařízení, např. sériového portu má proto následující tvar:

[1] je nutno odlišovat službu (operaci) ovladače a systémovou službu. Operace ovladače je mnohem hlouběji v jádře OS a je využívána i dalšími vyššími vrstvami OS. Systémová služba (v tomto případě *ioctl*) je v rozhraní jádra (tj. je přímo užívána aplikacemi) a je pouze jakousi zkratkou k službě ovladače (má proto omezenější funkci)

```
dsc = open("ttyS0", O_WRONLY);
write(dsc , buffer, 1);
```

Nejdříve je otevřen speciální soubor, tj. je aktivován algoritmus *namei*, a po nalezení čísla i-uzlu je vytvořen dynamický i-uzel a záznamy v systémové a procesové tabulce otevřených souborů (a je vrácen index do tabulky procesové jako tzv. deskriptor). Vše stejně jako u fyzického souboru. Navíc je však otevřeno vlastní zařízení a to voláním operace OPEN ovladače, jenž je identifikován číslem *major* a na zařízení označené číslem *minor* (obě čísla jsou uložena v otevřeném i-uzlu). Při zápisu přes deskriptor, pak dojde na úrovni i-uzlu ke zjištění, že zapisováno není do souboru ale znakového zařízení a dojde proto k k přímému zápisu na zařízení (nikoliv tedy k zápisu do svazku přes vyrovnávací paměti). Zápis se děje operací READ příslušného ovladače (opět s využitím major a minor).

Klasický Unix dovoloval existenci speciálních souborů na libovolném svazku a v libovolném adresáři. Pro vytvoření speciálního souboru postačovalo zavolat službu OS s názvem *mknod* (existuje i přikaz stejného jména, který tuto službu volá). Při vytvoření stačí předat jméno speciálního souboru a obě čísla (major a major). Čísla bylo možno zjistit z dokumentace příslušného ovladače.

V praxi se však speciální soubory vytvářely jen v jediném adresáři : */dev* (a na systémovém svazku) a to většinou již při instalaci systému (ovšem opět přes *mknod*). Při instalaci byly vytvořeny speciální soubory pro všechna nalezená hardwarová zařízení a pro všechna softwarová zařízení poskytovaná jádrem systému (včetně toho nejdůležitějšího tj. */dev/null*). Toto řešení byly dlouhou dobu dostatečné, neboť zařízení byl jen omezený počet a žádná se nepřipojovala za běhu systému.

U modernějších OS však začaly objevovat problémy. Kvůli možnosti dynamického připojování zařízení musely být vytvářeny stovky speciálních souborů (z nichž jen minimum označovala skutečně existující zařízení). Proto nejnovejší Unixovské systémy (včetně Linuxu) vytvářejí speciální soubory až ua běhu prostřednictvím virtuálních FS (tj. obsah */dev* již není vytvářen fyzicky na disku, ale je emulován jádrem podle skutečného stavu zařízení). Navenek se tedy nic nezměnilo, ale implementace je již zcela rozdílná.

Následující výpis obsahuje prvních 15 souborů z výpisu adresáře */dev* (pomocí příkazu *ls -l*)

```
crw-rw----+ 1 fiser root    14,  12 2007-06-17 11:23 adsp
crw-------  1 root  root    10, 175 2007-06-17 11:22 agpgart
crw-rw----+ 1 fiser root    14,   4 2007-06-17 11:23 audio
drwxr-xr-x  3 root  root         60 2007-06-17 11:22 bus
lrwxrwxrwx  1 root  root          4 2007-06-17 11:23 cdrom -> scd0
lrwxrwxrwx  1 root  root          4 2007-06-17 11:23 cdrom-sr0 -> scd0
crw-------  1 fiser root     5,   1 2007-06-17 11:23 console
lrwxrwxrwx  1 root  root         11 2007-06-17 11:22 core -> /proc/kcore
drwxr-xr-x  6 root  root        120 2007-06-17 11:22 disk
crw-rw----  1 root  root    14,   9 2007-06-17 11:23 dmmidi
drwxr-xr-x  2 root  root         60 2007-06-17 11:24 dri
crw-rw----+ 1 fiser root    14,   3 2007-06-17 11:23 dsp
lrwxrwxrwx  1 root  root         13 2007-06-17 11:22 fd -> /proc/self/fd
brw-rw----  1 fiser floppy   2,   0 2007-06-17 11:23 fd0
```

V tomto výpise je jen 7 speciálních souborů (ostatní jsou symbolické odkazy na speciální soubory zavádějící alternativní jména zařízení a adresáře s dalšími speciálními soubory).

Znaková zařízení začínají znakem „c“ a pátém a šestém sloupci se vypisují major a minor čísla (např. většina znakových zařízení ve výpise má major 14, které odpovídá ovladači zvukové karty (o lze zjistit výpisem souboru */proc/device*), zařízení *dsp* má pak např. minor 3). Výpis blokových zařízení má stejnou strukturu, jen začíná znakem „b“ (zde je jen jedno odpovídající první disketové mechanice).

6.2.3 Horní a dolní polovina ovladače

Ovladače většinou přistupují k hardwarovým zařízením, která pracují asynchronně vzhledem k procesoru (tj. i vzhledem k rutinám jádra). Výjimkou jsou pouze zařízení, který umožňují přímý synchronní přístup (např. hodiny reálného času) a samozřejmě i všechna zařízení softwarová.

horní polovina ovladače
dolní polovina ovladače

U asynchronních zařízení je se ovladač skládá ze dvou částí (označovaných jako poloviny), které se výrazně liší a to jak svým umístěním v jádře OS, tak svou funkcí. Tzv. **horní polovina ovladače** je běžnou rutinou jádra (tj. je volána jako běžná procedura) a zajišťuje rozhraní vůči vyšším vrstvám (viz operace ovladače výše). **Dolní polovina ovladače** je však *obslužnou rutinou přerušení*, jež je vyvoláváno daným zařízením. Není proto přímo vyvolávána zbytkem jádra (tj. ani dolní polovinou) a běží vůči zbytku jádra asynchronně (tj. může být vyvolána mezi jakýmikoliv dvěma instrukcemi jádra, bez ohledu na to jaká rutina resp. jaký proces právě běží). Přerušení je ve většině případů vyvoláno v případě, že zařízení provedlo výstup (u výstupních zařízení), resp. na zařízení se objevila vstupní data (u zařízení vstupních).

Pro zjednodušení si přestavme vstupně-výstupní zařízení, které umožňuje přímý vstup a výstup jediného bytu a které je plně ovládáno hlavním procesorem a využívá jediné přerušení (tomuto zjednodušení odpovídá např. sériový port).

Předpokládejme, že horní polovina ovladače byla zavolána v rámci jistého procesu a to prostřednictvím služby READ ovladače (vstup jednoho bytu). Horní polovina ovladače se podívá do vyrovnávací paměti daného zařízení a pokud tato obsahuje alespoň jeden byte, tak jej ihned vrátí. Pokud je vyrovnávací paměť prázdná pak se proces zablokuje (přesune do stavu *sleep* a opustí procesor). Po jisté době se data na vstupním zařízení objeví (tj. např. něco zašle byty na sériový port), je vyvoláno přerušení. V rámci přerušení se vykoná dolní polovina ovladače, která byte přesune do vyrovnávací paměti a odblokuje proces (ten se přesune do stavu *waiting*). Poté se provede návrat z obslužné rutiny přerušení a dále běží proces, ve kterém k přerušení došlo. Po jisté době je naplánován původní proces (čtenář) a ten převezme přečtený byte a vrátí jej.

Toto komunikace mezi horní a dolní polovinou ovladače je v zásadě komunikací mezi producentem a konzumentem (nad sdílenou vyrovnávací pamětí). Není to však zcela pravda, neboť systém není zcela symetrický. Zamysleme se např. co se stane pokud dojde k přeplnění vyrovnávací paměti (horní polovina resp. procesy, které jí volají nestačí odebírat došlé byty). V klasickém vztahu producent-konzument by muselo dojít k zablokování producenta dat tj. zde dolní poloviny ovladače (ta by se zablokovala dokud by nedošlo k uvolnění nějakého místa ve vyrovnávací paměti). Zde však k zablokování dojít **nesmí**!

Důvod je jednoduchý — dolní polovina ovladače se může vykonat v kontextu libovolného procesu tj. i procesu, který s daným zařízením vůbec nekomunikuje (měl jen to neštěstí, že běžel právě v okamžiku vzniku přerušení). Pokud by se dolní polovina měla zablokovat, muselo by dojít k zablokování tohoto (právě běžícího procesu). Proces by tak

čekal na událost, se kterou nemá nic společného a ani ji nemůže nějak ovlivnit. Navíc zde není ani jistota zda vůbec dojde k odblokování (proces čtenář se může zastavit v jiném čekání resp. uvázne) resp. k odblokování v rozumném čase. Pokud proces vyžaduje rychlou odezvu (např. jedná se o internetový server) může být toto čekání neakceptovatelné. V ještě horším případě by se mohlo jednat o proces systémový (např. zloděje stránek), u něhož může prodleva vést k pádu celého systému (a nemůže tomu zabránit ani vysoká či dokonce realtimová priorita procesu). Zablokování obslužní rutiny přerušení je proto zcela vyloučené a obslužná rutina musí v tomto případě příchozí data zahazovat (např. u klávesnice to doprovází často pípáním). Je to sice nepříjemné, ale vstupní data si lze ve většině případů vyžádat znovu (např. u sítě novým zasláním paketu).

Zákaz blokování dolních polovin ovladačů však ještě neřeší všechny problémy, neboť obslužná rutina může probíhat relativně dlouho i v případě, že se neblokuje. Pokud by byla funkce dolní poloviny příliš komplexní, vedlo by to k náhodným a mnohdy již neakceptovatelným prodlužováním odezvy u všech programů (včetně služeb na pozadí a systémových procesů). Proto musí být dolní polovina co nejjednodušší (v zásadě jen několik málo instrukcí pro převzetí a kopírování dat v rozsahu pár bytů). Všechny složitější operace (např. filtrování a předzpracování dat) musí provádět horní polovina.

Většina těchto problémů je spojena se vstupem dat. Při výstupu také dochází ke spolupráci horní a dolní poloviny, ta však má poněkud jiný charakter. Horní polovina vloží při výstupu data do vyrovnávací paměti (pokud je plná pak se zablokuje). Pokud je naopak prázdná musí po vložení svých dat iniciovat začátek přenosu (přímým přístupem k hardwaru). Dolní polovina je aktivována v okamžiku, kdy je dílčí výstup dat hotov. Zkontroluje zda nejsou další data k přečtení (pokud ano iniciuje další přenos) a ukončí se. Pokud je již vyrovnávací paměť prázdná, ihned skončí. I zde se tedy dolní polovina nikdy nezablokuje, toto chování je však přirozenější (nikdy nedochází ke ztrátě dat).

V reálném systému mohou být vztahy mezi horní a dolní polovinou komplexnější, ale v zásadě se vždy jedná o nesymetrický vztah producent–konzument. Tak je tomu i u blokových zařízení (ty nevyužívají jedinou vyrovnávací paměť, ale sdílenou předem alokovanou pro každý blok a k blokování dolní poloviny nedochází již z podstaty)

OTÁZKY

1. Jaký je rozdíl mezi službou INIT a OPEN v rozhraní ovladačů?
2. Co je přerušení?

OTÁZKY K ZAMYŠLENÍ

1. Jaký je rozdíl v mechanismu sdílení vyrovnávací paměti mezi blokovými a znakovými zařízeními?
2. Kde najdete informace o major číslech ovladačů v Linuxu?

7 Bezpečnost na úrovni OS

CÍLE KAPITOLY

Bezpečnost na úrovni OS je jen malou částí informační bezpečnosti, přesto by však vydala na samostatný kurz. Navíc mnohé aspekty bezpečnosti jsou součástí jiných kursů studia oboru Aplikovaná informatika.

Z tohoto důvodu se tato kapitola v zásadě omezuje jen na dva mechanismy, které jsou úzce spojeny s moderními informačními systémy:

- hodnocení (evaluace) úrovně bezpečnosti
- PKI infrastruktura a certifikační autority

Po dokončení získáte základní přehled o problematice (včetně základní terminologie) a ověříte si svou schopnost uvažovat o různých aspektech informační bezpečnosti z pohledu využívání možností moderních operačních systémů.

V praxi budete moci interpretovat například následující informace:

- operační systém je certifikován na úrovni EAL4 (dříve podle TCSEC na C2)
- certifikát použitý na WWW serveru je označen jako nedůvěryhodný
- SSH umožňuje bezpečnou komunikaci se serverem, i když komunikace probíhá po nezabezpečeném komunikačním kanálu (pozor *umožňuje* nikoliv *zajišťuje*).

7.1 Co je bezpečnost

Bezpečnost je jedním z nejčastěji skloňovaných slov v oblasti informačních technologií. S různými aspekty bezpečnosti se setkávají i běžní uživatelé počítačů, živí se jí desetitisíce specialistů a byly o ní napsány stovky či tisíce knih (a ještě více článků). Na otázku co je bezpečnost, však není ani v případě že se omezíme na informační technologie, jednoduchá odpověď. Proto musíme začít nejdříve alespoň částečným vymezením pojmů a teprve potom se můžeme zaměřit na jednotlivé dílčí problémy (omezíme se jen na ty nejdůležitější, neboť problematika je tak široká, že ji lze v rámci těchto skript jen nastínit).

Počítačovou bezpečnost (přesněji bezpečnost IT) lze pro účely definice rozdělit na tři základní koncepty (CIA triáda):

důvěrnost (confidentiality) – data (a obecněji i služby) jsou dostupné pouze oprávněným osobám (přístup neoprávněných osob je narušením bezpečnosti)

integrita (integrity) – data (resp. jiné) služby mohou modifikovat pouze oprávněné osoby (u dat typicky například pouze autor). V mnoha případech je narušení integrity maskováno (u dat).

dostupnost (availability) – služby (a také data) musí být (oprávněným osobám) přístupné v okamžiku kdy tyto služby (data) potřebují. Opakem dostupnosti je *odepření* (služeb) [denial (of service)]

Při různých útocích na informační systémy (a při snaze jim účinně bránit se všechny tři koncepce (aspekty) bezpečnosti vzájemně provázány (i kdyý často jeden aspekt převažuje).

Příklady útoků:

ukradení počítače: narušena je primárně dostupnost, pokud nemáme data šifrována pak může dojít k narušení důvěrnosti (nění to však zcela nutné), integrita však přímo narušena není (nepočítáme-li případnou úplnou ztrátu dat, ale to je spíš odepření.

napadení virem: nejčastějším a nejviditelnějším projevem je odepření služeb (včetně např. zpomalení běhu OS a aplikací), lze uvažovat i o narušení integrity služeb (aplikace zobrazující stav, nemusí zobrazovat skutečnost). Nekteré viry narušují i integritu dat (drobné modifikace datových souborů) a relativně časté je i narušení důvernosti (typicky například využitím vašeho e-mailového adresáře)

odposlech (hlasu) – zde je přirozeně nejvíce narušena důvěrnost. Dostupnost je naopak zcela neovlivněna (resp. jen minimálně). Narušení integrity se týká částečně služeb, a nikoliv dat (u datových přenosů, však může docházet i k online změnám přenášených dat).

cenzura: zde je typické narušení integrity dat. Méně podstatné je narušení důvěrnosti (cenzurovaná data nebývají důvěrná)

Oblasti bezpečnosti

Kromě základních aspektů je pro popis bezpečnosti důležitá i oblast IT technologií k níž se bezpečnost vztahuje (tj. vůči čemu se vede útok). Mezi základní oblasti bezpečnosti patří:

- bezpečnost hardwarová
- bezpečnost na úrovni OS
- bezpečnost síťová (především ve WAN sítích)
- bezpečnost aplikační (včetně např. bezpečnosti databázové)
- bezpečnost personální (útoky na IT vedené skrze osobu)
- bezpečnost organizační (organizační zabezpečení v rámci organizace)

Příklad (umělý):

Bezpečnostní manager je špatně placen (problém organizační bezpečnosti) je podplacen další osobou (= personální bezpečnost) a vydá mu své bezpečnostní heslo do systému. Ten je špatně nakonfigurován (bezpečnost OS), což umožní útočníkovi vzdálené přihlášení (síťová bezpečnost) získaným heslem a modifikaci programu zajišťujícího kontrolu osob přicházejících do technologického centra firmy (aplikační bezpečnost). Toho útočník využije pro vstup do tohoto centra a odcizení zálohovacích médií (hardwarová bezpečnost).

Stejně jako v případě aspektů se jednotlivé oblasti bezpečnosti prolínají a nelze je mechanicky oddělovat (jak lze vidět i z umělého příkladu). Celková bezpečnost (informačního) systému je dána jeho bezpečností v dílčích oblastech, přičemž přirozeně platí, že nemůže být větší než bezpečnost nejslabšího článku (může však být i nižší). Z tohoto hlediska je proto často úzkým hrdlem bezpečnost na úrovni personální a organizační (která často stojí mimo okruh zájmů IT bezpečnostních managerů). Proto i když se těmito oblastmi nebudeme dále podrobněji zabývat, je nutné je mít stále na zřeteli.

7.2 Kvantifikace bezpečnosti

Bezpečnost resp. úroveň bezpečnosti je jen obtížně kvantifikovatelná (tj. není např. snadné porovnat bezpečnost dvou operačních systémů apod.). V zásadě existuje je jediná skutečně univerzální míra bezpečnosti — peníze. Bezpečnost systému lze ohodnotit mírou peněžních prostředků, které byly investovány na zajištění bezpečnosti (a ta by přirozeně měla odpovídat finančním možnostem potenciálních útočníků).

Peníze jako míra bezpečnosti mají však i mnoho nevýhod, jejich hodnota (relativní) se mění s časem i místem, cena může být ovlivňována i dalšími (nebezpečnostními) aspekty (trh u systémů s vyšší úrovní bezpečnosti je relativně malý a mohou se v něm projevovat monopolistické tendence). Hlavní nevýhodou je však skutečnost, že často hodnotí bezpečnost per partes nikoliv bezpečnost celkovou. Organizace si například mohla zakoupit drahé bezpečnostní prvky či know-how, ty však spolu nespolupracují nebo spolupracují jen obtížně. I přes enormně vydané finanční prostředky tak může být jako celek jen minimálně bezpečný.

evaluace bezpečnosti

I z tohoto důvodů se již od osmdesátých let provádí **evaluace (hodnocení) bezpečnosti** informačních systémů. Tuto evaluaci provádí nezávislá firma (nezávislá na uživateli IS) a jejím výsledkem je certifikát bezpečnosti, jehož součástí je i kvantifikace úrovně bezpečnosti (často to nebývá jen jediná úroveň, ale soubor více kvantifikovaných charakteristik). Podstatné ja skutečnost, že bezpečnost systému se vyhodnocuje jako celek (tj. ve všech oblastech i jejich vzájemných interakcích).

Tento přístup však má i několik nevýhod:

- evaluace systému jako celku je velmi náročná (časově a personálně) a tudíž velmi drahá (dovolit si ji mohou jen velké podniky či státní orgány)
- metodika různých firem a následně jejich kvalitativní i kvantitativní charakteristiky se mohou lišit (a jsou tudíž mnohdy téměř neporovnatelné)

Z tohoto důvodu začaly vznikat národní (a později i mezinárodní) standardy bezpečnostních certifikací, které jsou navíc schopné hodnotit i nasazení univerzálně navržených informačních systémů. Jádrem těchto systémů jsou nejčastěji systémy operační a tak se běžně zaměňuje evaluace operačního systému za evaluaci informačního systému jako celku (do níž patří i pravidla správy, organizační a personální metodiky, interakce s okolními systémy). Předběžná evaluace obecných systémů (bez ohledu na konkrétní nasazení) sice není tak dokonalá jako evaluace konkrétního systému in situ (tj. v konkrétní instalaci v konkrétní organizaci s konkrétními lidmi), ale je nesrovnatelně levnější.

V současnosti existuje několik standardů bezpečnostního hodnocení, z nichž nejdůležitější jsou dva. Systém *Trusted Computer System Evaluation Criteria* (TCSEC, známý také jako *Orange Book*), byl vytvořen v osmdesátých letech na ministerstvu obrany USA (DoD), je tak už poněkud zastaralý, ale relativně jednoduchý (a vhodný především pro orientační klasifikaci bezpečnosti informačních systémů. Výrazně komplexnější (a také novější)

je mezinárodní standard Common Criteria (CC), který se začíná šířeji prosazovat (a na rozdíl od TCSEC je i standardem pro ČR).

7.2.1 Trusted Computer System Evaluation Criteria (TCSEC)

Tento standard zavádí pouze jednorozměrnou charakteristiku úrovně bezpečnosti a nehodnotí ani jednotlivé konkrérní bezpečnostní prostředky – místo toho se soustřeďuje na metodiku a různé obecné bezpečnostní charakteristiky. Celková úroveň bezpečnosti v tomto standardu je ohodnocena jednou ze stupňů D, C2, C1, B6, B2, B1 až A1 (se vzrůstající bezpečností).

D = systém, jež neprošel evaluací (tím se ale nikdo nechlubí), nebo nebyl vůbec evaluován. U systémů, které neprošli evaluací se často místo označení D používá tzv. pravděpodobně nejvyšší dosažitelný stupeň bezpečnosti (což je de facto nepřípustné, ale nese to určitou informační hodnotu). Například operační systém MS DOS lze tímto stupněm ohodnotit, neboť nebyl certifikován a ani v případě certifikace by nedosáhl lepší stupeň bezpečnosti. Naopak většina distribucí Linuxu má de facto bezpečnostní stupeň D (nebyly z finančních důvodů certifikovány), ale reálně (po správné konfiguraci by mi mohly dosáhnout úrovně maximálně C2).

C1 = systém s tzv. Discretionary Access Control (DAC) tj. s volitelným řízením přístupu k jednotlivým informačním prostředkům (včetně dat). V tomto systému tak může mít každý prostředek definovány omezení, kdo k němu může přistupovat resp. jak k němu může přistupovat, přičemž oprávnění jsou vázána na konkrétní a systémem ověřitelnou identitu (např. fyzickou osobu, aplikaci nebo vzdálený systém). Volitelnost spočívá v tom že identifikovatelný subjekt může, pokud má k tomuto právo, nastavovat libovolná omezení včetně omezení nulových (tj. není systémem nucen udržovat jistou minimální úroveň bezpečnosti). Operačních systémů přesně na této úrovni není mnoho, neboť mohou relativně snadno dosáhnout úrovně C2.

C2 = kromě požadavků převzatých z úrovně C1, zde přibývá nutnost vést bezpečnostní audit (tj. provádět logování událostí důležitých z hlediska bezpečnosti a především je nutno takto získané informace analyzovat. Mezi další požadavky patří nutnost poskytovat příslušnou systémovou dokumentaci a manuály. Složitější jsou i požadavky na přihlašovací procedury. Této úrovně mohou jako nejvyšší dosáhnout informační systémy založené na běžných operačních systémech (Unix, Linux, MS Windows XP, Mac OS X) a některé jsou pro tuto úroveň i oficiálně certifikovány. K dosažení teéto úrovně však nestačí jen běžná instalace těchto systémů, ale i příslušná konfigurace a správa (ta je však např. u akademických systémů jen stěží dosažitelná).

povinná kontrola pístupu

B1 = tato úroveň přináší závažnou změnu v bezpečnostní politice systému tzv. **Mandatory Access Control** (MAC). Ta na rozdíl od DAC spočívá v explicitním vynucení dané úrovně bezpečnosti. Tento přístup lze ukázat (na zjednodušeném) příkladě převzatém ze světa tajných služeb. Subjekty v systému (uživatelé, data, protředky OS) musí být označeny bezpečnostním návěštím, které je zařazuje do určité úrovně oprávnění (např. v pořadí důvěrné, tajné, přísně tajné). Uživatel má přístup jen k prostředkům na stejné nebo nižší úrovni oprávnění (pokud má navíc i oprávěné dané DAC, které může být užíváno společně s MAC). Pokud nějaký

prostředek vytváří (např. pořizuje text), pak jej musí vytvořit na své úrovni bezpečnosti (tj. nikoliv na menší). Jinak řešeno (opět jsme u tajných služeb), agent prověřený na úroveň tajný (tj. mající toto bezpečnostní návěstí) může číst tajné a důvěrné dokumenty (nikoliv přísně tajné) a pokud nějaký dokument vytvoří pak musí mít úroveň tajné (ne důvěrné, neboť by tak mohl prozradit tajné informace osobám s nižší úrovní oprávnění). Opatřování dokumentů bezpečnostními návěštími si vyžaduje i export těchto návěští při zálohování a výměnně dat mezi systémy (data musí být opatřena příslušnými daty a zašifrována). Samozřejmostí je formální specifikace bezpečnostních politik. Této úrovně dosahují i některá bezpečnostní rozšíření klasických operačních systémů (např. některé Unixy nikoliv však Linuxy). Tyto systémy jsou většinou úzce specializované (běžné aplikace na nich nespustíte) a jsou používány v oblastech, které tuto bezpečnost vyžadují (bankovnictví, vojenství apod.)

B2 = tato úroveň se v základních rysech příliš neliší od úrovně B1 (kterou obsahuje). Rozšíření jsou spíše formálního charakteru jako např. model bezpečnostní politiky čistě definovaný a formálně dokumentovaný, DAC a MAC omezení musí mít důsledně všechny objekty v systému, kontrola skrytých kanálů, silnější autentifikace, strukturální odlišení prvků kritických pro bezpečnost od ostatních (právě toto odlišení je nejtypičtějším znakem této úrovně), separace bezpečnostních administrátorů a systémových operátorů (nemůže existovat nějaký superuživatel). Od této úrovně je nezbytné intenzivní a dlouhodobé testování jednotlivých částí systému.

B3 = i na tomto stupni se jedná spíše o evoluci. Nejtypičtějším rysem je tzv. nutnost implementace tzv. referenčního monitoru, což je malý systémový modul, který kontroluje veškerý přístup k prostředkům. Tento softwarový modul je dostatečně malý, aby byl intenzívně testován a byly v něm odhaleny všechny bezpečnostní nedostatky. Dalším charakteristickým rysem je stálá detekce případných průniků (s případnou bezprostřední odpovědí). Ostatní požadavky lze shrnout pod ještě intenzívnější testování a ještě dokonalejší formální specifikaci bezpečnostního modelu. Systémy této úrovně se již běžně nepoužívají a jejich fukčnost je relativně omezená (či lépe výrazně specializovaná)

A1 = tato úroveň je formálně shodná s úrovní B3, vyžaduje však formální matematický důkaz jednotlivých aspektů bezpečnosti. Jen několik málo systému bylo evaluováno na tuto úroveň bezpečnosti (např. SCOMP v roce 1984)

Jak již bylo řečeno, je tento standard již zastaralý a v současnosti se již na evaluaci nových systému nepoužívá. Existuje však možnost jak na něj mapovat souhrn jednotlivých úrovní novějších standardů.

7.2.2 Common Criteria

Mezi novějšími standardy je nejdůležitější standard *Common Criteria* (formální název *Evaluation criteria for IT security*), což mezinárodní standard (ISO/IEC 15408), který nahradil několik starších standardů národních. Prosazuje se od počátku 21. století a v současnosti je vůči němu evaluována většina operačních systémů.

Tento standard je oproti TCSEC mnohem komplexnější a především umožňuje evaluaci vztáhnout k bezpečnostním požadavkům a různým typům bezepečnostních mechanismů (neposkytuej však už žádné absolutní jednodimenzionální hodnocení úrovně bezpečnosti).

Pro posouzení bezepčnosti cílového sysytému (ve standardu je označován je jako TOE = *Target Of Evaluation*) musí existovat několik úzce provázaných formálních dokumentů, jejichž některé části jsou formalizované, jiné lze sdílet mezi producenty dané třídy produktů (např. operačních systémů), a některé jsou specifické pro daný produkt. Mezi nejdůležitější patří:

Protection Profile (PP) – dokument, vytvořený uživateli resp. komunitou daného systému (resp. dané třídy systémů), který identifikuje bezpečnostní požadavky. Výrobci mohou své produkty certifikovat vzhledem k těmto požadavkům a ty tak mohou být navíc zohledněny i v jednotlivých bezpečnostních cílech (ST).

Security Functional Requirements (SFR) – specifikuje jednotlivé bezpečnostní funkce, které mohou být poskytovány produktem [převzato z]. Standard obsahuje katalog těchto prostředků a jejich případné vzájemné vztahy.

Security Target (ST) – nejdůležitější dokument při konkrétní evaluaci. Formálně definuje bezpečnostní cíl, a to prostřednictvím jednotlivých SFR specifikací (ty jsou vybrány na základě PP). To umožňuje volit takový bezpečnostní cíl, který je požadován uživateli daného produktu a jen ten evaluovat (evaluace tedy není absolutní). Dokonce i produkty stejného zaměření mohou být evaluovány oproti zcela různým bezpečnostním cílům. Tento dokument musí být při evaluaci vždy zveřejněn.

Evaluation Assurance Level (EAL) – hlavní výsledek evaluace. Kvantativní hodnocení úrovně bezpečnosti vztažené k danému bezpečnostnímu cíli (!) založené na základě ohodnocení kvantity a kvality testování jednotlivých SFR i jejich vzájemných vztahů (tj. hodnotí záruky dosažení nikoliv bezpečnost samu). Standard rozeznává sedm úrovní s následujícími označeními :

- EAL1 Functionally Tested
- EAL2 Structurally Tested
- EAL3: Methodically Tested and Checked
- EAL4: Methodically Designed, Tested and Reviewed
- EAL5: Semiformally Designed and Tested
- EAL7 Semiformally Verified Design and Tested
- EAL7: Formally Verified Design and Tested

Již z označení je zřejmé, že se nehodnotí absolutní bezpečnost, ale záruky získané testováním a (semi)formální verifikací. Proto nejsou jednotlivé dosažené úrovně přímo porovnatelné. V případě běžných operačních systémů (užívajících bezpečnost založenou na uživatelích tj. DAC) však existuje jistý společný bezpečnostní a tak lze provést alespoň částečné porovnání. Většina OS dosahuje úrovně EAL 4 (MS Windows XP, Suse Linux Enterprise Server 9, apod.), specializovanější systémy (které by v TCSEC dosahovali úrovně B2-B3) pak EAL 5. Úrovně EAL 4 dosahují i bežně používané PKI tokeny (viz např. iKey 3000)

7.3 Kryptografie a PKI

Klíčovou roli v zabezpečení operačních systémů (především při vyšších požadavcích na bezpečnost) hraje kryptografie tj. mecahnismy šifrování a spřízněných oborů (včetně

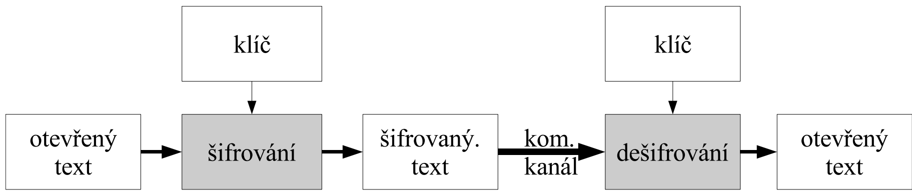
velmi důležité technologie digitálního podpisu). V těchto skriptech je pozornost zaměřena především na problematiku praktického využívání kryptografie (např. na technologii PKI). Teoretické východiska (i když jsou velmi důležitá) jsou zde zmíněna pouze okrajově.

7.3.1 Symetrická a asymetrická kryptografie

Z hlediska distribuce klíčů a celkového mechanismu šifrování jsou rozeznávány dvě základní oblasti kryptografie: *kryptografie se symetrickým klíčem* a *kryptografie s asymetrickým klíčem* (často označovaná také jako kryptografie s veřejným klíčem).

symetrický klíč

Kryptografie se symetrickým klíčem má již úctyhodnou tisíciletou historii a její základní principy jsou známé (mnozí je začínají chápat již na základních školách v rámci různých dětských organizací). Princip symetrické kryptografie shrnuje obrázek.



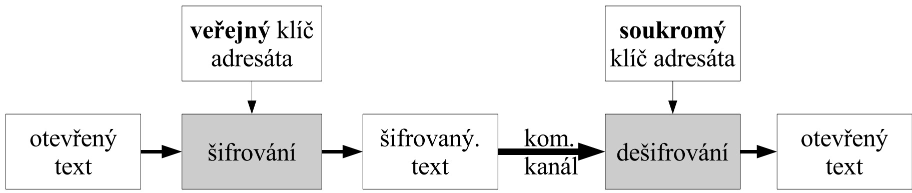
Typická je existence jediného klíče, který musí znát obě komunikační strany a který musí být utajen. Šifrování i dešifrování se provádí za pomoci tohoto jediného (sdíleného klíče). Zašifrovaný text může dešifrovat kdokoliv kdo zná klíč, tj. za normálních okolností adresát (příjemce) šifrované zprávy (resp. několik adresátů) a také její odesílatel.

Hlavní nevýhodou kryptografie se symetrickým klíčem je nutnost distribuce klíčů mezi všemi komunikačními partnery. Tato distribuce se musí dít bezpečným komunikačním kanálem, které musí být odlišný od kanálu, který je užíván pro přenos šifrovaných zpráv. Tj. pokud se pro přenos šifrovaných zpráv použije síť (např. Internet), pak se musí klíče vyměnit např. pomocí osobního předání, bezpečného kurýra (při vysokých požadavcích ne bezpečnost) resp. poštou nebo telefonicky (při nižších požadavcích, je jen málo organizací, které jsou schopny kontrolovat více komunikačních kanálů najednou a nepřetržitě). Navíc při vzrůstu počtu komunikujících partnerů roste počet používaných klíčů

Symetrická kryptografie však má i několik výhod, šifrování a dešifrování může být velmi rychlé (lze online šifrovat toky v jednotkách Mb nebo Gb za sekundu), existuje velké množství kvalitních šifer (a v neposlední řadě šifrování resp. dešifrování mohou provádět i lidé bez pomoci počítačů).

veřejný klíč

Krytpografie s veřejným klíčem je relativně mladá (veřejně je k dispozici od 70. let minulého století) přinesla však do kryptografie skutečnou revoluci. Z hlediska klasické symetrické kryptografie se její možnosti zdají neuvěřitelné a těžko pochopitelné (s jejími principy se většina lidí seznámí až na vysoké škole).



Pro asymetrickou kryptografie je typická existence paru klíčů. Tyto klíče vznikají v rámci jediného procesu (zdrojem jsou data z kvalitního generátoru náhodných čísel) a úzce spolu souvisí. Jejich použití se však výrazně liší: veřejný klíč je vskutku veřejný, a soukromý musí být utajován. Osoba, která oba klíče vygenerovala, veřejný klíč nabídne potenciálním komunikačním partnerům (např. zveřejněním na Internetu), soukromý klíč si bezpečně uschová (nejlépe na nějakém bezpečnostním zařízení).

Při šifrování odesílatel šifruje zprávu veřejným klíčem adresáta (který musí mít k dispozici) a odešle ji komunikačním kanálem adresátovi. Ten jako jediný zná odpovídající veřejný klíč (neboť oba klíče generoval a soukromý bezpečně uložil) a může tak zprávu svým soukromým klíčem dešifrovat (dešifrovat ji na rozdíl od symetrické šifry nemůže ani odesílatel!).

Výhody jsou zřejmé, neboť správa klíčů je výrazně jednodušší (každý potenciální účastník komunikace má jen jediný klíčový pár, který nemusí být jako celek distribuován jiným zabezpečeným komunikačním kanálem). Asymetrické šifrování má samozřejmě i své nevýhody, šifrování a dešifrování je relativně pomalé (to je dáno velikostí klíčů, které jsou běžně cca desetkrát delší než symetrické, i časovou náročností použitých algoritmů), a hlavně při distribuci veřejných klíčů může dojít k jejich podvržení (útočník pak může dešifrovat zprávy určené adresátovi bez znalosti jeho soukromého klíče).

Obě nevýhody však lze výrazným způsobem eliminovat. Pro zrychlení je možno kombinovat asymetrické šifrování se symetrickým. Při tomto tzv. hybridním šifrování jsou pomocí asymetrického šifrování distribuovány dočasné klíče pro symetrickou kryptografii (jeden z účastníku komunikace pošle druhému náhodné heslo zašifrované veřejným klíčem druhého partnera) a ty jsou pak použity pro (rychlé) zašifrování datového toku.

Nebezpečí podvržení veřejného klíče lze pak minimalizovat zavedením mechanismu tzv. certifikačních autorit, které budou popsány dále.

Už nyní by mělo být zřejmé, jakou revoluci přineslo používány veřejných klíčů do praktické kryptografie. Hlavní výhoda však ještě nebyla zmíněna. Tato výhoda souvisí s nutností **autentifikace** komunikujících partnerů. Použití šifrování předpokládá, že odesílatel si může být jist identitou příjemce (jen příjemce je schopen zprávu dešifrovat). Opačná autentizace (tj. zaručená identifikace odesílatele příjemcem) je také ve většině případů nutná (pokud by nebyla zaručena, mohl by kdokoliv zprávu podvrhnout). Je zřejmé, že v případě symetrické šifry je tato i tato autentifikace v mnoha případech relativně snadná, neboť spoluvlastník klíče je schopen zprávu zašifrovat. U asymetrické šifry to však není již tak snadné, neboť zprávu může zašifrovat kdokoliv (přesněji kdokoliv kdo zná veřejný klíč adresáta, což však je de facto kdokoliv). Odesílatel sice může uvést ve zprávě své identifikační údaje, ale ty lze podvrhnout (samozřejmě nikoliv přímo v existující zašifrované zprávě). Identifikátor však musí být v tomto případě sdíleným tajemstvím, jehož distribuce má přináší stejné problémy jako distribuce symetrických klíčů (po zachycení tajného identifikátoru není problém posílat adresátovi podvržené „tajné" zprávy, u nichž je příjemce záměrně uveden v omyl).

autentifikace

Naštěstí existuje v rámci kryptografie s veřejným klíčem řešení, které tyto chyby nemá a navíc přináší i další výhody (které symetrická kryptografie nenabízí). Toto řešení označované jako **digitální podpis** spočívá v drobné obměně šifrovacího schématu.

digitální podpis

Při použití digitálního podpisu odesílatel „šifruje" zprávu svým soukromým klíčem. Kdokoliv kdo zná jeho veřejný klíč může zprávu „dešifrovat" a buď její verifikací (tj. ověřením zda dává smysl) nebo prorovnáním s původní zprávou (pokud ji má k dispozici) může snadno ověřit, že byla „šifrována" příslušným odesílatelem (neboť je ten má k dispozici

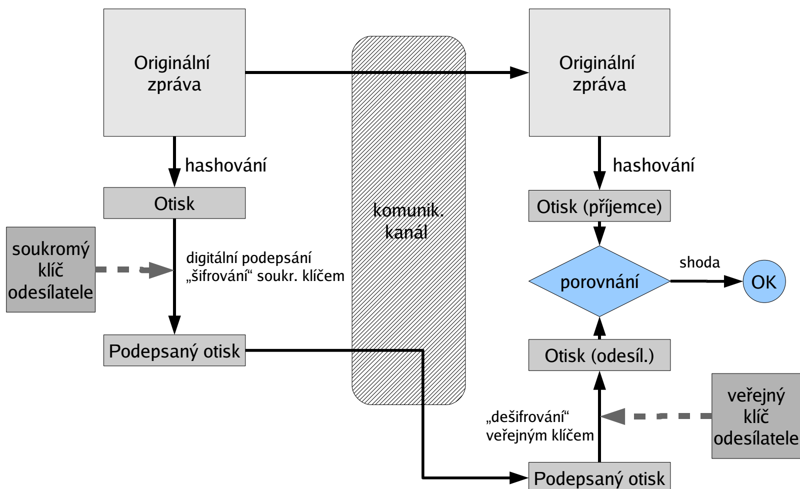
párový soukromý klíč). V předchozí větě, byla použita slova „šifrovat" resp. „dešifrovat", ale jedná se pouze o použití původních algoritmů šifrování a dešifrování pro zcela jiný účel (rozhodně nedojde ke vzniku šifrovaného textu, neboť jen stěží tak lze označit text, který může „dešifrovat" kdokoliv). A lze dokonce říci, že právě v oddělení šifrování a autentifikace odesílatele je hlavní přínos digitálního podpisu. U symetrické šifry totiž může identitu ověřit jen příjemce a nikdo jiný. Pokud by však chtěl např. příjemce dokázat před soudem, že zprávu obdržel od žalované protistrany musel by u symetrické šifry poskytnout sdílené heslo i soudu (resp. obecně státu) a dále dokázat, že si ji neposlal sám (což je obtížné). U digitálního podpisu tyto komplikace nejsou, každý kdo zná veřejné heslo odesílatele (včetně např. soudu) může ověřit, že odesílatelem byl skutečně příjemce (a ten nemůže tuto skutečnost popřít). Mechanismus tak lze použít k obecné autentifikaci (i zde je však nutné zabránit podvržení veřejného klíče). Navíc, pokud se podepsaná zpráva ještě zašifruje, získáváme mechanismus podobný symetrické šifře (identitu může ověřit jen příjemce a nikdo jiný, což je také někdy užitečné).

Drobným nedostatkem tohoto řešení je skutečnost, že pro ověření platnosti podpisu musí mít ověřující strana k dispozici i vlastní (otevřený) dokument (lze sice provést verifikaci i na základě např. smysluplnosti resp. formální správnosti „dešifrovaného" textu, ale toto řešení není obecně použitelné). Tento nedostatek lze vyřešit zasíláním původní zprávy spolu je její podepsanou podobou, ale v tomto případě se zpráva přenáší de facto dvakrát (navíc musí být celá „zašifrována", následně dešifrována a porovnána s otevřeným textem, což může být velmi pomalé). Proto se v praxi nepodepisuje celá zpráva, ale jen tzv. digitální otisk.

digitální otisk

Digitální otisk je několikabytová (16-64 bytů) hodnota, která vznikne aplikací hashovací funkce na text zprávy. Hashovací funkce (funkčně obdobná hashovací funkci používanou hashovacích tabulkách) vrací hodnoty, které jsou rovnoměrně rozptýleny v oboru hodnot a jsou citlivé na změnu hahovaného textu (tj. drobná změna v textu zprávy se projeví v zásadní změně hodnoty otisku). Při použití hashovavací funkce musí nutně docházet k tzv. kolizím tj. situaci, kdy dva různé texty mají stejný otisk (množina mnohabytových textů je vždy výrazně větší než množina otisků).

Použití otisků v procesu tvorby a ověřování digitálního podpisu je jednoduché. Nejdříve je k původní zprávě (OT) získán digitální otisk (DT). Potom je poslána zpráva sestávající se z původní zprávy a výsledku digitálního podpisu otisku (otisk je „šifrován" soukromým klíčem). Druhé straně stačí pro ověření odesílatele (= často tvůrce) získat vlastní digitální otisk zprávy a ten porovnat s otiskem získaným „dešifrováním" podepsaného otisku (za pomocí veřejného klíče). Pokud jsou shodné je odesílatel ověřen tj. nikdo se za něj nemohl vydávat a on sám nemůže odeslání (= často autorství) zprávy popřít.



Algoritmy asymetrické kryptografie používané v současné praxi jsou založeny pouze na třech základních přístupech: obtížnosti rozkladu prvočísel (RSA), obtížnosti diskrétního logaritmu (ElGamal, DSA) a problematika eliptických křivek (nejkratší klíče, nejsložitější matematika). To je výrazně menší nabídka než u symetrické kryptografie (navíc všechny tři problematiky jsou na sobě dosti závislé), což činí asymetrickou kryptografii poněkud zranitelnější (jediný objev může vést ke zpochybnění současné asymetrické kryptografie).

7.3.2 Certifikační autorita

Praktické použití šifrování s veřejným klíčem (včetně digitálního podpisu) vyžaduje pro zajištění dostatečné úrovně bezpečnosti i další pomocnou infrastrukturu (*PKI – Public Key Infrastructure*). Ta kromě bezpečného provádění základních operací (šifrování a autentizace prostřednictvím digitálního podpisu) zajišťuje i další služby založené na použití asymetrické kryptografie jako jsou např. ověřitelná časová razítka.

Jádrem rozšiřující infrastruktury však musí být mechanismus, který zabraňuje největšímu nedostatku základní kryptografie s veřejným klíčem — možnosti podvržení veřejného klíče. Veřejnost klíče sice na jedné výrazně usnadňuje jeho distribuci (ta se může dít i nezabezpečenými kanály), na straně druhé však nebrání jeho podvržení. Předpokládejme, že dvě osoby Anička a Bedřich chtějí spolu komunikovat prostřednictvím asymetrické kryptografie (tak by žádná další osoba nemohla narušit důvěrnost ani integritu jejich komunikace). Anička a Bedřich sídlí na různých kontinentech a tak si oba své veřejné klíče pošlou nezabezpečeným e-mailem (to je v zásadě bezpečné). Anička pak může napsat zprávu, digitálně ji podepsat svým soukromým klíčem a zašifrovat veřejným klíčem Bedřicha a poté poslat. Vše se zda v pořádku, zprávu může přečíst jen Bedřich (důvěrnost) a odeslat ji může jen Anička (integrita).

Co se však stane pokud třetí osoba např. Cyril na svém SMTP serveru zachytí e-mail s veřejným klíčem Aničky. Pak si může tento klíč uschovat a místo něj Bedřichovi pošle svůj veřejný klíč C1V (v rámci původného dopisu). Bedřich uvěří, že se jedná o veřejný klíč Aničky (z klíče nelze přímo určit jeho vlastníka). Zlotřilý Cyril poté podobným způsobem zachytí a zamění Bedřichův klíč poslaný Aniččce (za jiný svůj veřejný klíč C2V, může si jich vygenerovat téměř libovolné množství).

Anička nyní chce poslat tajný dopis Bedřichovi. Podepíše je svým soukromým klíčem a následně zašifruje za pomoci klíče o němž se domnívá, že je Bedřichův (je však Cyrilův jak mi vševědoucí víme). Cyril dopis zachytí, dešifruje jej svým soukromým klíčem (C2S) a následně ověří že je od Aničky (má Aniččin veřejný klíč). Pak si dopis bez problémů přečte (a ještě si je jist, že je od Aničky a může ho i změnit. Pak jej podepíše svým C1S a zašifruje veřejným klíčem Bedřicha a pošle Bedřichovi. Ten dopis dešifruje svým soukromým klíčem a za pomoci veřejného klíče C1V (o němž se domnívá, že je Aniččin, i když je Cyrilův si ověří, že je od Aničky (což je omyl, je ve skutečnosti od Cyrila). Smutný příběh končí konstatováním, že i když si jsou oba aktéři (Anička i Bedřich) jisti důvěrností a integritou své komunikace, je pravda zcela jiná (ovšem ani Cyril si nemůže být jist, protože důvěrnost a integritu zpráv, které čte a vytváří, může narušovat Dominik).

Pokud si tuto situaci rozebereme, zjistíme, že bez znalostí dodatečných informací nemohou Anička ani Bedřich průnik do své komunikace odhalit (a mohou je věřit, že se nikomu nevyplatí nepřetržitě zachytávat jejich komunikaci, aby zachytil jejich veřejné klíče).

Řešení však přirozeně existuje. Nejednodušší je možnost přímé výměny klíčů (osobní setkání) nebo přenos klíčů zabezpečeným kanálem (např. pokud jim klíče letecky dopraví jejich společná přítelkyně - letuška). Obecně je však toto řešení nepraktické, neboť bezpečný kanál nemusí být dostupný (všichni se nepřátelíme s letuškami nebo kurýry), a i kdyby jej bylo možno použít, vedlo by jeho obecné používání ke stejným problémům jako distribuce sdílených klíčů symetrické kryptografie (počet by rostl kvadraticky s počtem komunikujících) a tak by použití veřejných klíčů oproti tajným sdíleným nepřinášelo žádnou výhodu.

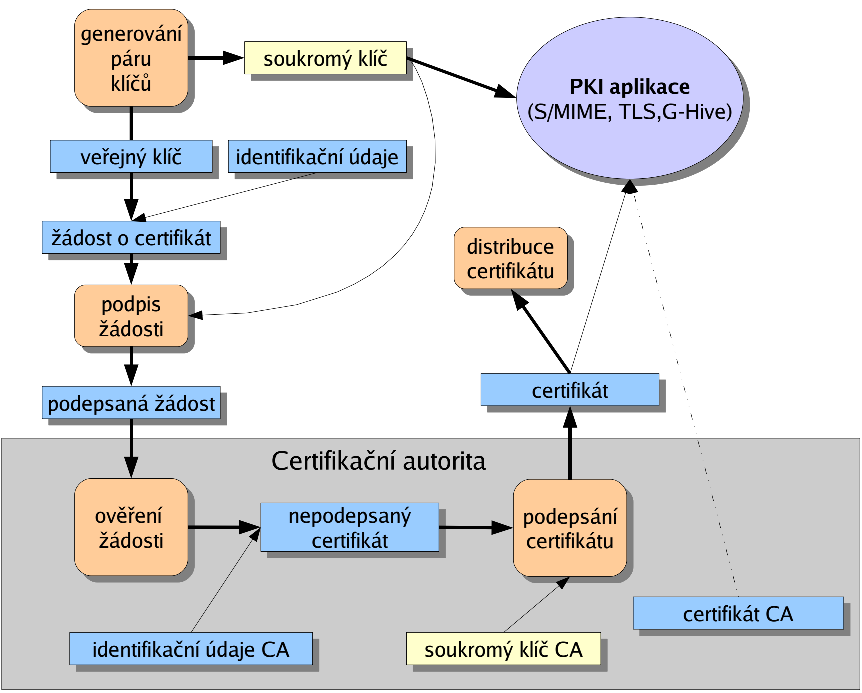
Díky neschopnosti většiny narušitelů paralelně odposlouchávat a především modifikovat dva nebo dokonce více nezávislých kanálů, lze situaci mírně zjednodušit použitím nezávislého kanálu pro potvrzení klíčů (Anička např. může po obdržení Bedřichova veřejného klíče zavolat Bedřichovi telefonem a nechat si nahlas přečíst jeho klíč resp. otisk jeho klíče, a poté zkontrolovat, že ji došel nemodifikován). Situace je však zjednodušena pouze pro dvě komunikující osoby (nemusí otravovat letušku) a nikoliv obecně (stále je zde kvadratický růst využití záložních spojení).

Zásadní řešení spočívá v možnosti podepisování veřejných klíčů třetími osobami. Pokud tedy má Anička přítelkyni Žofii, jejíž veřejný klíč z jistou vlastní a Žofie vlastní ověřený veřejný klíč Bedřichův, pak může Žofie poslat Bedřichův klíč Aničce podepsaný vlastním klíčem. Nyní stačí aby Anička důvěřovala Žofie, že nedopouští takových věcí jako ke podepsání neověřeného klíče či dokonce podvržení klíčů, a problém je vyřešen: Anička prostřednictvím Žofie může důvěřovat pravosti Bedřichova veřejného klíče (a to bez jakéhokoliv přímého spojení s Bedřichem). Řešení lze ještě dále zobecnit, neboť důvěra podepřená podpisy klíčů může být i tranzitivní (Žofie sama může důvěřovat klíči díky podpisu pro ni důvěryhodné osoby, atd.). Navíc klíč může být podepsán i větším počtem osob a pak stačí důvěřovat jen jediné z těchto osob. Výsledkem je v tomto případě jakási síť důvěry, která spojuje blízké osoby a kterou lze díky tranzitivnosti a vícenásobným podpisům rozšířit i na dosti velké komunity uživatelů (existují i celosvětové sítě tohoto druhu). Toto řešení má však i zřejmé nedostatky, neboť síť nemusí být dostatečně hustá (ne vždy se podaří najít cestu od jedné osoby ke druhé) a s rostoucí délkou cest jejich důvěryhodnost klesá (důvěra není ve skutečnosti, až tak úplně tranzitivní). Na druhou stranu je plně decentralizovaná a tudíž obtížněji napadnutelná (navíc může například existovat i více cest důvěry a infiltrace tak může být dosti obtížná). Toto řešení je využíváno především u programu PGP a jeho komunity.

Toto řešení je však dosti těžkou uplatnitelné u komunity, která není těsněji vázána ani není propojena dostatečným počtem vazeb, a kde je především požadavek na mnohem větší důvěryhodnost (např. při komunikaci občanů se státními orgány nebo pracovníku se svým vedením). V tomto případě je dobré vyčlenit jedinou dílčí organizaci, která je důvěryhodná a má možnost ověřovat identitu uživatelů a jejich veřejných klíčů. Ta jediná podepisuje veřejné klíče všech potenciálních účastníků komunikace (a zodpovídá tak za za jejich věrohodný).

certifikační autorita

Tento přístup k distribuci věrohodných klíčů se používá u tzv. **certifikačních autorit**, které tvoří páteř standardního PKI. Certifikační autority (dále jen CA) vydávají tzv. certifikáty, což jsou zaručené veřejné klíče klientů CA (autorita musí ověřit identitu klientů i jejich klíčů), k nimž se připojí identifikační údaje jejich držitelů (jméno, pozice, apod.) a jež jsou následně podepsány soukromým klíčem CA. Pokud osoba důvěřuje CA a má její věrohodný certifikát (je to v zásadě veřejný klíč CA podepsaný jí samou) pak může důvěřovat všem certifikátům vydaným CA (a tím i příslušným veřejným klíčům jejích klientů). Získání důvěryhodného certifikátu CA není sice zcela jednoduché (při přenosu internetem může být snadno podvržen), ale je rozhodně jednodušší než ověřování certifikátů u všech potenciálních komunikačních partnerů (certifikáty CA lze získat přímo u dané CA, od důvěryhodných přátel a organizací resp. spolu s komunikačním produktem).



Hlavní funkcí certifikační autority je vytvářené certifikátů. Zájemce o certifikát (tj. potenciální uživatel PKI infrastruktury) nejdříve vygeneruje oba své klíče. K veřejnému klíči předá své identifikační údaje (o jejich rozsahu rozhoduje CA, ale vždy obsahují např. jméno) a vytvoří tzv. žádost o certifikát (ten musí mít přesně definovaný formální tvar). Pak žádost podepíše svým soukromým klíčem (aby dokázal certifikační autoritě, že jej vlastní tj. nenechává si podepisovat cizí veřejný klíč). Podepsanou žádost pošle certifikační autorita.

Ta nejdříve ověří jeho správnost, tj. správnost formálních údajů (např. při osobním kontaktu s žadatelem a na základě například občanského průkazu), formální správnost a úplnost údajů a nakonec i digitální podpis na žádosti. Je-li vše v pořádku, přidá identifikační

údaje CA a vytvoří tak certifikát (prozatím nepodepsaný). Následuje podpis tohoto certifikátu soukromým klíčem CA (to je z bezpečnostního hlediska ten nejkritičtější krok procesu a musí být zabezpečen na všech úrovních od hardwarové k personální). Výsledkem je (podepsaný) certifikát, který je předán zájemci. Ten pak může certifikát distribuovat (to může udělat i CA např. prostřednictvím svých WWW stránek). Certifikát je často také importován spolu se soukromým klíčem (a mnohdy i certifikátem CA) do PKI aplikace (jen soukromý klíč je nezbytný pro dešifrování a podepisování, oba certifikáty však jeho použití usnadňují především při používání klíčů). Všimněte si také, že certifikační autorita nezná soukromý klíč odpovídající certifikátu (a nemůže ho tak za žádných okolností zneužít).

Certifikáty (a žádosti o certifikát) jsou binární soubory se strukturou odpovídající obecnému standardu ASN.1 (ve kódování DER). Jsou proto přímo nečitelné, ale jejich obsah lze zobrazit ve většině PKI aplikací (např. ve Firefoxu) nebo pomocí specializovaných nástrojů.

Kromě podepisování má CA i další důležité funkce, které většinou souvisí s distribucí klíčů a především jejich další správou během jejich života. Certifikáty totiž nemají nekonečnou životnost, ale jsou platné jen v určitém časovém rozpětí. Tato časové rozpětí je stanoveno CA (je součástí informací přidaných CA) a nelze jej změnit (stejně jako všechna metadata certifikátu jsou podepsána CA). Certifikát se běžně vydává s platností od okamžiku je ho vytvoření po dobu jednoho nebo dvou let. Omezená platnost je dána bezpečnostními důvody (pokud má útočník dostatečně dlouhou dobou může nalézt soukromý klíč k veřejnému hrubou silou), ale pravděpodobně i komečními (za vystavení certifikátu se platí). Omezenou životnost má i certifikát certifikační autority (ovšem delší, běžně 5-10 let). CA proto musí zajišťovat obnovu certifikátů klientů (zpoplatněnou) tak i obnovu svého certifikátu (CA musí používat několik svých certifikátů, jejichž platnost se překrývá v čase, neboť platnost vydaných uživatelských certifikátů nemůže být delší než platnost příslušného certifikátu CA).

revokace certifikátu

Certifikát však může ztratit svou platnost i před vypršením původní doby platnosti. Děje se tak například při zcizení nebo ztrátě soukromého klíče (tj. platnost je narušena ze strany držitele certifikátu), nebo při změně osobních dat uložených v certifikátu (např. při změně pozice ve firmě u firemních certifikátů. Ve všech těchto případech musí CA certifikát zneplatnit (odborněji revokovat). To se přirozeně neprojeví na samotném certifikátu (ten se už může vyskytovat v tisících kopií, které nelze najednou změnit). Místo toho musí CA revokovaný certifikát zveřejnit v tzv. ***seznamu revokovaných certifikátů*** (CLR = Certificate revocation list). Každá aplikace využívající certifikát by měla tento seznam konzultovat v okamžiku kdy ověřuje digitální podpis (nebo šifruje pro držitele certifikátu). To je dost problematické, neboť je vyžadován on-line přístup k seznamu tj. jisté centrální databázi (certifikát už tak není sám o sobě dostatečným potvrzením autenticity). Navíc i když je při získávání CLR snadno dosáhnout integrity (CLR je samozřejmě digitálně podepsáno certifikační autoritou), hrozí útoky typu DOS (*denial of service*, tj. odepření služby).

Jak lze vidět je funkce CA relativně dosti komplexní a u velkých CA si vyžaduje vytvoření několika dílčích organizací, z nichž každá se zabývá jen určitou speciální funkcí (je to dobré i z hlediska bezpečnosti, neboť například je malá část osob by měla mít přístup k soukromému klíči). Časté je např. vynětí ověřování identity osob (tzv. registrační autorita) nebo WWW presentace (včetně distribuce certifikátů a CLR).

Pro detailnější studium PKI doporučuji knihu:

DOSTÁLEK, Libor, Marta VOHNOUTOVÁ a Miroslav KNOTEK. *Velký průvodce infrastruk-*

turou PKI a technologií elektronického podpisu. 2., aktualiz. vyd. Brno: Computer Press, 2009, 542 s. ISBN 978-80-251-2619-6.

7.3.3 Vytvoření bezpečného kanálu

Kryptografie s veřejným klíčem se zatím nestala viditelnou součástí našich životů. Jen velmi malá menšina lidí vlastní (a používá) certifikát nebo používá jiný systém asymetrické kryptografie (a i ti co certifikát vlastní jej často používají jen párkrát za rok pro komunikaci s úřady).

bezpečný (šifrovaný) kanál

Přesto však existuje aplikace asymetrické (resp. přesněji hybridní) kryptografie se kterou se setkáváme každodenně a v mnoha podobách. Je to tzv. **bezpečný (šifrovaný) kanál** mezi dvěma vzdálenými zařízený a musí splňovat následující požadavky:

1. je jej možno vytvořit i na nezabezpečené lince (a bez existence sdíleného tajemství)
2. alespoň jeden komunikační partner je vůči druhému autentifikován (tj. je ověřena jeho identita)
3. komunikace je šifrována (včetně výměny jakýchkoliv informací citlivých na útok v postranním kanálu). Komunikace je tedy důvěrná
4. je zajištěna integrita přenášených dat (nelze např. provést tzv. vkládací útok, kdy je odposlechnutá komunikace použita útočníkem, tím že je použita v následné komunikaci s jedním z partnerů za účelem zopakování původně zaslaného požadavku). Tj. jinak řečeno, přenášená (šifrovaná) data musí být různá i v případě, kdy jsou originální (nešifrovaná) data shodná.

Bezpečné kanály jsou užívány v následujících běžně používaných technologiích (seznam nemusí být úplný):

- vytvoření bezpečného kanálu mezi mobilem (přesněji SIM kartou) a ústřednou operátora (oba účastníci se musí autentifikovat).
- vytvoření bezpečného spojení mezi bankomatem a bankou (i zde je nutná oboustranná autentifikace)
- vytvoření bezpečného kanálu mezi WWW klientem a serverem (zde většinou postačuje autentifikace serveru). V současnosti je pro tento účel (a další typy bezpečného internetového spojení) používán obecnější standard TLS (dříve SSL) vytvářející bezpečný kanál přímo nad TCP/IP socketem.
- vzdálené přihlašování přes SSH (zde je opět nosičem nezabezpečený TCP/IP kanál) není však využíváno SSL

Vytvoření bezpečného kanálu si můžeme ukázat na příkladu protokolu SSH1 (SSH2 a TLS používají poněkud složitější protokol, který je však principiálně shodný).

Na počátku máme dva komunikační partnery, z nich každý vlastní svůj dvojici klíčů (včetně soukromého) a navíc veřejný klíč partnera (i zde je často Achillovou patou distribuce veřejných klíčů). Dále mezi partnery existuje obousměrný datový kanál umožňující přenášet data (zde je proudový socket). I když obecně mohou být partneři plně symetričtí, je běžnější rozdělení rolí tj. jeden z partnerů je označován jako server druhý jako klient (tato asymetrie se však týká spíše podkladového proudu a navazování spojení, po vytvoření je kanál plně symetrický).

Prvotní komunikace začíná na nezabezpečeném (nešifrovaném) kanálu.

1) server pošle klientu verzi SSH protokolu, kterou podporuje.

2) klient ověří zda danou verzi podporuje, a pokud ano pak ji potvrdí (může poslat i jinou verzi a pak se může naopak přizpůsobit server)

pokud se oba komunikační partneři shodnou na verzi následuje vlastní kryptografická relace:

3) server pošle svůj veřejný klíč a dočasný veřejný klíč (ten je generován jednou za hodinu), dále pošle osm náhodných kontrolních bytů (anti-spoofing cookie), které zajistí jedinečnost dále přenášených dat (zabrání vkládacímu útoku). Dále posílá informace o šifrách a dalších kryptografických algoritmech (otisky, hashe), které podporuje (tzv. kryptografická svita = suite)

4) klient si ověří zda zná klíč serveru (tj. má jej v databázy známých serverů), pokud ne zeptá se uživatele zda jej chce potvrdit (SSH nepoužívá certifikační autority a klíč musí být zkontrolován jiným způsobem)

5) klient vygeneruje jedinečný klíč symetrické šifry pro danou relaci (dále označovaný jako klíč relace). Ten spolu s kontrolními byty serveru. zašifruje veřejným klíčem serveru a dočasným veřejným klíčem a pošle serveru. Navíc posílá seznam kryptografických algoritmů, které hodlá používat (především algoritmus symetrické šifry), jenž jsou vybrány ze svity

6) server dešifruje klíč relace pomocí obou soukromých klíčů, které vlastní tj. svým trvalým klíčem a (jednohodinovým) klíčem dočasným.

Nyní již probíhá komunikace na šifrovaném (bezpečném kanále), k čemuž server používá dešifrovaný klíč relace a algoritmus zvolený klientem

7) server pošle potvrzovací správu (již šifrovanou)

8) klient ověří zda je potvrzovací zpráva šifrována (správným klíčem), pokud ano tak je tím dokončeno vytvoření bezpečného kanálu, ale prozatím jen s jedinou autentifikací (klient zná identitu serveru, ale naopak to neplatí). To v některých případech stačí (např. při připojení k zabezpečené WWW stránce přes SSL), nikoliv však u SSH (mobilu, bankomatu).

Identitu klienta lze pak ověřit např. jednoduchým heslem (který posílá klient serveru). Předávání hesla je bezpečné, neboť heslo se posílá zabezpečeným kanálem autentifikovanému serveru (tj. nemůže jej zachytit případný útočník).

Lepší je však použití asymetrické kryptografie za použití dvojice klíčů klienta (ty prozatím nebyly použity).

9) klient pošle požadavek na autentifikaci a v ní uvede identifikaci daného klíče (např. jeho digitální otisk)

10) server najde odpovídající klíč ve své databázi (a ověří jeho omezující podmínky)

11) server vygeneruje náhodný 256bitový řetězec (tzv. výzva), zašifruje jej veřejným klíčem klienta (ze své databáze) a pošle jej klientovi

12) klient výzvu dešifruje a poté zkombinuje s klíčem relaci (viz bod 5) a s výsledku vypočítá digitální otisk. Otiskl pošle jako odpověď na výzvu.

13) server, vypočítá z výzvy a klíče relace (obě zná) otisk a porovná je s otiskem s odpovědí klienta. Jsou-li stejné je klient autentifikován.

Zajímavá je skutečnost, že klient nevrací jako odpověď na výzvu dešifrovanou výzvu samotnou. Důvod je jednoduchý — pokud by tak učinil, mohl by server podstrčením

skutečného šifrovaného (nenáhodného) textu dosáhnout jeho dešifrování (což by mohl využít k útoku na klíč klienta). Podstrčit by mohl i prostý text a dosáhnou tak jeho podepsání. Proto klient vrací jen otisk, jenž je navíc počítán i s přidanou informací (klíčem relace), což brání útokům spočívajícím ve vícenásobném odeslání téže výzvy. Ano, ani server nemusí být na té správné straně síly (resp. může být hacknut).

OTÁZKY

1. Jaký důsledek má odhalení souboru s nešifrovaným textem hesel pro bezpečnost OS. Zhodnoťte podle různých aspektů CIA triády?
2. Jaký je rozdíl mezi DAC a MAC?
3. Jaký je základní rozdíl mezi evaluací podle TCSEC (Orange Book) a Common Criteria?
4. Zhodnoťte výhody a nevýhody symetrické a asymetrické kryptografie?
5. Jaká je funkce certifikační autority? (co a proč dělá)
6. Co zajišťuje resp. nezajišťuje bezpečný komunikační kanál?

OTÁZKY K ZAMYŠLENÍ

1. Diskutujte situaci, kdy certifikovaný OS obsahuje skrytou bezpečnostní vadu, která je odhalena až po bezpečnostní evaluaci.
2. Diskutujte bezpečnostní funkci těchto opatření při interakci s certifikační autoritou (včetně možných útoků, problematických scénářů)
   a) žádost o certifikát musí být podepsána párovým soukromým klíčem
   b) žádost o revokaci musí být podepsána párovým soukromým klíčem
   c) certifikáty vydané CA nesmí být (bez svolení) použity jako certifikáty (podřízené) CA
   d) certifikáty mají omezenou dobu platnosti (včetně toho jak hodnotit dokument podepsaný prošlým párovým klíčem)
3. Co je útok postranními kanály? Jaký může mít vliv na bezpečnost bezpečného kanálu?

Literatura

[1] Bach, Maurice. *Principy operačního systému Unix.* 1. vyd. Praha : SAS, 1993. 514 s. ISBN 80-901507-0-5

[2] Čada, Ondřej. *Operační systémy.* 1. vyd. Praha : Grada, 1994. 384 s. ISBN 80-85623-44-7

[3] Custer, Helen. *Windows NT.* 1. vyd. Praha : Grada, 1994. 424 s. ISBN 80-85424-87-8

[4] Goodheart, Berny, Cox, James. *The magic garden Explained.* 1. vyd. London : Prentice Hall, 1994. 664 s. ISBN 013 098138 9

[5] Hansen, Per B.. *Principy operačních systémů.* 1. vyd. Praha : SNTL, 1979. 356 s. ISBN neuvedeno

[6] Kerrisk M. *The Linux Programming Interface: a Linux And UNIX System Programming Handbook* [e-book]. San Francisco: No Starch Press; 2010. Available from: eBook Collection (EBSCOhost), Ipswich, MA. Accessed October 23, 2013.

[7] *Linux – Dokumentační projekt.* 3. vyd. Brno : Computer Press, 2003. 1020 s. ISBN 80-7226-761-2

[8] Lukáš J. *Jádro systému Linux.* ComputerPress, 2008. ISBN: 9788025120842

[9] Plášil, František. *Operační systémy.* Praha : SNTL, 1992. 439 s. ISBN 80-03-00269-9

[10] Skočovský, Luděk. *Principy a problémy systému Unix.* 1. vyd. Veletiny : Science, 1993. 288 s. ISBN 80-901475-0-X

[11] Skočovský, Luděk. *Unix,Posix,Plan 9.* 1. vyd. Brno : Skočovský Luděk, 1998. 394 s. ISBN 80-902612-0-5

[12] Stallings, William. *Operating systems: internals and design principles.* Ninth edition. Harlow: Pearson, 2018, ISBN 978-1-292-21429-0