UNIVERZITA PALACKÉHO V OLOMOUCI
KATEDRA INFORMATIKY

Martin Hořava

PŘEPIS PŘEDNÁŠEK PŘEDMĚTU OPERAČNÍ SYSTÉMY 2

zimní semestr 2012/2013

Abstrakt

Obsah

1. Operační paměť

- zásadní část počítače
- uložení kódu a dat běžících procesů i OS
- přístup k zařízením (DMA) //Direct Memory Access - přístup zařízení se namapuje do operační paměti a dá se poté přístupovat k zařízení přes operační paměť
- **virtuální paměť** umožňuje např. přístup k souborovému systému
- z HW pohledu může být operační paměť realizovaná různými způsoby (DRAM, SRAM, (EEP)ROM, ale i HDD, SSD, flash paměti)
- přístup k CPU k paměti nemusí být přímočarý (L1, L2 a L3 cache)
- pro jednoduchost budeme HW stránku věci zanedbávat
- Ulrich Drepper: What every programmer shoul know about memory - článek kde je popsaná jak funguje operační paměť atd do detailů

1.1. Požadavky na správu paměti(Memory managment)

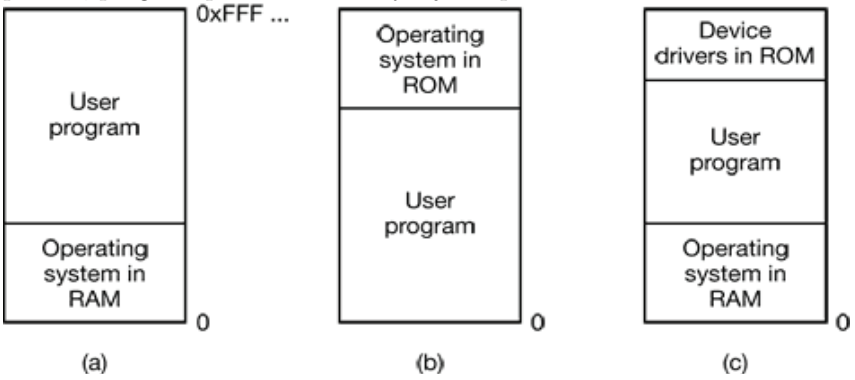
- evidence prostoru volného a přiděleného procesům
- přidělování a uvolňování paměti procesů
- přesunutí (přiděleního prostoru) - program by neměl být závislý na místě, kde se v paměti nachází (nutně k umožňování swapování)
- ochrana (přiděleného prostoru) - jednotlivé procesy by měly být izolovány
- sdílení - pokud je to žádoucí, mělo by být možné sdílet některé části paměti mezi procesy (2x spouštěný stejný program nebo knihovny, které jsou nahrané pouze jednou a poté jsou používány několika procesy)
- logická organizace - pamět počítače (spojitý prostor, "sekvence bytů") vs. typický program skládácjící se z modulů (navíc některých jen pro čtení nebo ke spouštění)
- fyzická organizace - paměť může mít více částí/úrovní (RAM? disk); program jako takový se nemusí vlézt do dostupné paměti RAM

1.2. Adresní prostory (logická organizace)

- v současných OS existuje několik různých nezávislých adresních prostorů, způsobů číslování paměťových buněk
- každý proces by měl mít vlastní pamětový prostor (izolace)
- různá zařízení mají odlišné adresní prostory
- ve spolupráci s hardwarem dochází k mapování fyzické paměti do adresního prostoru procesu
- např. grafická paměť může být namapována od adresy 0xD0000000

1.3. Správa paměti v jednoúlohových operačních systémech

- na správu paměti nejsou kladeny výraznější nároky //není kladen důraz na izolaci a nemusí se řešit sdílení paměti
- v jeden okamžik může běžet jenom jeden proces (MS-DOS, CP/M)
- (časté řešení) operační systém a ovladače zařízení se nachází na začátku nebo na konci paměti, program pak někde ve zbývajícím prostoru



1.4. Rozdělení paměti na souvislé bloky pevné velikosti

- předpokládejme, že OS je umístěn v nějaké části paměti
- nejjednodušší přístupu je rozdělit pamět na souvislé bloky stejné velikosti (např. dříve 8 kB, dnes spíše 8 MB)
- pokud program potřebuje víc paměti, musí se sám postarat o odsun/načtení dat do sekudnrání paměti
- pokud program potřebuje míň paměti dochází k neefektivnímu využití paměti - tzv. vnitřní fragmentaci
- výhodou je, že lze velice jednoduše vybrat umístění kam načíst proces
- problém přesunutí → relativní adresování
- v IBM OS/360 (historická záležitost)
- vylepšení: rozdělit paměť na bloky různých velikostí (např. 2, 4, 6, 8, 12 a 16 MB)
- jedna vs. více front (každá fronta pro procesy, které se vlezou do dané paměti)
- vzniká zde vnitřní fragmentace ⇒ bloky byly zabrány, ale každý např.: jen z 1/2

1.5. Přidělování paměti po blocích proměnlivé velikosti

- každý program dostane k dispozici tolik paměti, kolik potřebuje
- sníží se tak míra vnitřní fragmentace
- po čase dochází k vnější fragmentaci (paměť je volná, ale je rozkouskovaná a není možné alokovat větší blok)
- strategie přiřazování paměti
  - **first fit** - začne prohledávat paměť od začátku a vezme první vhodný blok

  - **next fit** - začne prohledávat paměť od místa kam se podařilo umístit blok naposledy
  - **best fit** - přiřadí nejmenší vyhovující blok (v paměti jsou nevyužité bloky, ale jsou co nejmenší)
  - **worst fit** - vezme se největší blok (v paměti njesou malé nevyužité bloky, ale nelze spustit větší program)
- informace o volném místu lze ukládat do bitmap nebo seznamů volného místa
- v MS-DOS (historická záležitost)
- **vylepšení: zahuštění (compaction)** - procesy jsou v paměti přesunuty tak, že vznikne souvislý blok volného místa (časově náročná práce)

1.6. Buddy alokace paměti

- kompresní řešení
- přiděluje paměť po blocích velikosti 2^K, kde L $\leq$ K $\leq$ U, kde 2^L je nejmenší blok, který lze přiřadit a 2^U je největší možný blok (celá paměť)
- algoritmus alokace paměti velikosti s
  1. najdi blok velikost $2^{k-1} \leftarrow$ s $\leq 2^k$, případně pokud je s $\rightarrow 2^L$, blok velikost 2^L
  2. pokud takový není, rozděl nejmenší blok větší než 2^k na půl (např.: pokud požaduje 3 kB a volný je pouze 32 kB, tak půli bloky, dokud nevznikne blok 4 kB a ten přiřadí)
  3. přejdi na krok jedna
- uvolnění paměti //rychlý algoritmus
  1. uvolni blok paměti
  2. je jeho soused volný?
  3. pokud ano, sluč je dohromady, pokračuj krokem 2
- snižuje míru vnější fragmentace, má střední míru vnitřní fragmentace
- spojení volných bloků je rychlé

1.7. Stránkování

- adresní (logický) prostor procesu je rozdělen na menší úseky - stránky (*pages*)
- fyzická paměť je rozdělena na úseky stejné délky - rámce (frame, page frames)
- provádí se mapování logický adres $\rightarrow$ na fyzické
- procesy už nemusí být umístěny v souvislých blocích paměti
- výpočet fyzické adresy musí být implementovaný HW (efektivita) //pokud by implementoval OS, nemohl by ho používat
- CPU si udržuje *stránkovácí tabulku*
- logická adresa má dvě části: *pd*, kde *p* je číslo stránky a *d* je *offset*
- fyzická adresa vznikne jako *fd*, kde *f* je číslo rámce v tabulce stránek příslušného stránce *p*

1.8. Adresář stránek

- velikost stránek je většinou mocnina 2 (typicky 4 KB)
- jednoduchý přesun na disk
- uvažujeme-li velikost stránky 4 KB a velikost adresního prostoru 32 bitů → velikost adresní tabulky je 1 milion záznamů
- nepraktické udržovat tak velkou tabulku (obzvlášť pro každý proces)
- používá se víceúrovňová tabulka
- část logické adresy udává tabulku, další část index v tabulce a další část offset
- např. pro 4 KB stránky může být toto rozložení 10-10-12 bitů //tabulku-index vybrané tabulky-offset
- tzn. systém k adresaci používá 1024 tabulek po 1024 záznamech
- značnou výhodou: nevyužité tabulky mohou být prázdné ⇒ nemusí se evidovat
- v praxi se používací i tří a čtyřúrovňové tabulky //pro desítky GB fyzické paměti
- ke stránkám jsou asociovány dodatečné informace (NX bit, modified bit)

1.9. TLB: Translation Lookaside Buffer

- cache procesoru obsahující hodně používané části stránkovacích tabulek
- pro danou stránku uchovává adresu rámce z důvodů efektivity
- pokud je adresa v cache (cache hit) je hodnota vrácena velice rychle (cca 1 hodinový cyklus)
- pokud hodnota není v cache (cache miss) načtení adresy trvá delsí dobu (10 - 100 cyklů), typický miss rate <1% → průměrný přístup k zjištění rámce je 1.5 hodinévho cyklu
- princip lokality - je vhodné programovat tak, aby data ke kterému se přistupuje byly na jednom místě (lokální proměnné na zásobníku, ne globální, které jsou rozesety po paměti programu)

1.10. Segmentace

- fotka Vasa

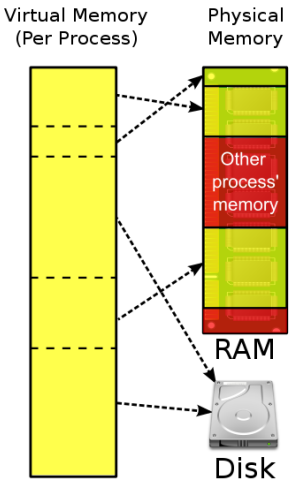
1.11. Copy-on-Write(COW)

- může být užitečné sdílet paměť (komunikace, úspora místa)
- dva stejné programy/knihovny v paměti
- stránky více procesů jsou navázaány na jeden rámec
- daná stránka má nastavený příznak CoW
- dojde-li k pokusu o zápis, vznikne vyjímka a procesu je vytvořena kopie rámce
- bude-li na rámec s příznakem CoW odkazovat jenom jedna stránka, příznak se odstraní
- `fork()` - vytvoří identickou kopii procesu; data jsou sice izolovaná, ale je možné sdílet kód
- úspora místa; úspora strojového času (není potřeba vytvářet kopie stránek/rámců); nízká penalizace pokud stránky přepíšeme
- virtuální paměť umožňuje další optimalizace (mapování souborů do paměti, atd.)

2. Virtuální paměť

2.1. Motivace

- paměť RAM je relativně drahá → nemusí vždy dostačovat
- aktuálně používaná data (např. instrukce) musí být v RAM, nepoužívaná data nemusí (velké programy → nepoužívané funkce)
- je vhodné rozšířit primární paměť (RAM) o sekundární (např.: HDD)
- zvětšením dostupné paměti je možné zjednodušit vývoj aplikací (není potřeba se omezovat v množství použité paměti)
- sekundární pamět bávý řádově pomalejší
- k efektivní implementaci je potřeba spolupráce HW (MMU) a OS
- z pohledu aplikace musí být přístup k paměti transprentní
- virtuální paměť (VM) je součástí soudobých OS (swapování)
  - Windows NT - stránkovycí soubor (pagefile.sys)
  - Linux - swap partition (ale může být i soubor)
- bezpečnost dat v sekundárním paměti? (např.: po vypnutí počítače, řeší se např. kódováním)



2.2. Inicializace procesu a jeho běh

- můžeme načíst celý proces do primární paměti (může být neefektivní)
- demand paging (stránkování na žádost)
- do paměti se načtou jen data (stránky), která jsou potřeba (případně související → sekvenční čtení (prefetch))
- systém si eviduje, které stránky jsou v paměti a které ne (HW, stránkovací tabulka)
- přístup na stránku, která není v primární paměti → přerušení - výpadek stránky (page fault)

- přerušení načte stránku do paměti, aktualizuje stránkovací tabulku
- je-li primární paměť plná, je potřeba nějakou stránku přesunout do sekundární paměti (odswapovat)
- pokud je odsouvaná stránka sdílená (např.: CoW) je potřeba aktualizovat všechny tabulky, kde se vyskytuje
- potřeba efektivně převádět rámce na stránky

2.3. Vlastnosti stránek

2.3.1. Rezervovaná stránka

- existují v adresním prostoru, ale nezapisovalo se do ní
- každá stránka je nejdřív rezervovaná
- vhodná pro velké pole, ke kterým se přistupuje postupně
- zásobník

2.3.2. Komitovaná stránka (Commited)

- stránka má rámec v primární nebo sekundární paměti
- musí řešit jádro
- paměť je často současně komitovaná i rezervovaná

2.3.3. Další vlastnosti

- dirty bit - 0 pokud má stránka přesnou kopii v sekundární paměti; 1 nastaveno při změně (nutná podpora HW)
- present/absent bit - přítomnost stránky v paměti (HW, nutné k detekci výpadků stránek)
- mohou mít přístupná práva (NX bit)

2.4. Výměna stránek

- page fault
- pokud není stránka v primární paměti, načte stránku do ní
- není-li volný rámec v primární paměti, je potřeba odsunout nějakou stránku do sekundární paměti
  - získáme volný rámec v sekundární paměti (pokud není volný rámec v sekundární paměti, najde se takový, který má kopii v primární paměti, nastaví se dirty bit a daný rámec se použije)
  - vybere se „oběť“ - stránka v primární paměti, která bude uvolněna
  - pokud má stránka nastavený dirty bit, překopíruje se obsah rámce do sekundární paměti
  - načte se do primární paměti stránka ze sekundární
- zopakuje instrukci, která vyvolala page fault
- některé stránky je možné zamknout, aby nebyly odswapovány (nutné pro jádro, rámce sdílené s HW)

2.5. Výběr oběti

- hledáme stránku, která nebude v budoucnu použitá (případně v co nejvzdálenější budoucnosti)

2.5.1. FIFO

- velice jedodnoduchý algoritmus
- stačí udržovat frontu stránek
- při načtení nové stránky je stránka zařazena na konec fronty
- pokud je potřeba uvolnit stránku bere se první z fronty
- nevýhoda - odstraní i často užívané stránky
- Beladyho anomálie - za určitých okolností může zvětšení paměti znamenat více výpadků stánek

2.5.2. Lest Frequently Used (LFU)

- málo používané stánky $\Longrightarrow$ nebudou potřeba
- problém se stránkami, které byly nějaký čas intenzivně využívány (např. inicializace)

2.5.3. Most Frequently Used (MFU)

- práva načtené stránky mají malý počet přístupů

2.5.4. Least Recently Used (LRU)

- jako oběť je zvolená stránka, která nebyla nejdýl používaná
- je potřeba evidovat, kdy bylo ke stránce naposledy přistoupeno
- řešení:
  1. počítadlo v procesoru, inkrementované při každém přístupu a ukládané do tabulky stránek
  2. „zásobník“ stránek - naposledy použitá stránka se přesune navrchol
- nutná podpora hardwaru

2.5.5. LRU (přibližná varianta)

- každá stránka má přístupový bit (reference bit) nastavený na 1, pokud se ke stránce přistupovalo
- na počátku se nastaví reference bit na 0
- v případě hledání oběti je možné určit, které stránky se nepoužívaly
- varianta
  - možné mít několik přístupových bitů
  - nastavuje se nejvyšší bit
  - jednou za čas se bity posunou doprava
  - přehled o používání stránky $\rightarrow$ bity jako neznaménkové číslo $\rightarrow$ nejmenší = oběť

2.5.6. Algoritmus druhé šance

- založen na FIFO
- pokud má stránka ve frontě nastavený přístpuový bit, je nastaven na nula a stránka zařazena nakonec fronty
- pokud nemá je vybraná jako oběť
- lze vylepšit uvážením ještě dirty bitu

2.5.7. Buffer volných rámců (optimalizace)

- proces si udržuje seznam volných rámců
- přesun oběti je možné udělat se zpožděním
- případně, pokud je počítač nevytížený, je možné ukládat stránky s dirty bitem na disk a připravit se na výpadek (nemusí být vždy dobré)

2.6. Problémy stránkování

2.6.1. Minimální počet rámců

- každý proces potřebuje určité množství rámců (např. `movsd` potřebuje v extrémním případě 6 rámců)
- stránkovací tabulka(y) musí být opět v rámci
- přidělování rámců procesům
  - rovnoměrně
  - podle velikosti adresního prostoru
  - podle pririty
  - v případě výpadků stránek podle priority (globální alokace rámců)
- pokud počet rámců klesne pod nutnou mez, je potřeba celý proces odsunout z paměti
- horší trashing (proces začne odsouvat stránky z paměti, které právě potřebuje)

2.7. Trashing

- systém je ve stavu, kdy odvádí spoustupráce, ale bez rozumného efektu
- modelevá situace:
  - pokud klesne vytížení procesoru, systém spustí další proces
  - pokud je použitý alogritmus s globální alokací rámců, může odebírat rámce ostatním procesům
  - ostatní procesy můžou tyto rámce požadovat a brát je ostatním procesům
  - čeká se na sekundární paměť → sníží se využití CPU
  - procesor se pokusí spustit další proces, atd.
- viz. Keprt p. 105
- lokální alokace rámců může trashing omezit
- ideální je, aby měl proces tolik rámců, kolik potřebuje

2.8. Řešení trashingu

2.8.1. Pracovní množina rámců(working-set)

- vychází z principu lokality
- má-li proces tolik rámců, kolik jich v nedávné době (lokalitě) použil → OK
- má-li jich více → neefektivní využití
- má-li jich méně → hrozí trashing a je lepší celý proces odsunout z primární paměti
- hrozí vyhladovění velkých procesů
- náročný výpočet (např.: podobný algoritmu druhé šance)

2.8.2. Frekvence výpadků stránek

- sledujeme, jak často dochází u procesu k výpadku stránky
- je potřeba stanovit horní a dolní mez
- pokud proces je mimo tyto meze → přidat/ubrat rámce

2.9. Velikost stránek

- stránky mají velikost 2^n, typicky v intervalu 2^{12} - 2^{22}, i.e., 4 KB - 4MB
- závisí na HW architektuře (může být i víc nebo míň)
- z pohledu fragmentace je vhodnější mít stránky menší
- více menší stránek zabírá místo v TLB → časté cache miss
- při přesunu do swapu může být velká stránka výhodnější (přístupová doba)
- některé systémy umožňují používat různé velikosti
- Windows NT $\leq$ 5.1 & Solaris velké stránky pro jádro, malé pro uživatelský prostor
- Windows Vista a novější - large pages
- Linux (hugetblfs)

2.10. Převrácená tabulka stránek (Inverted Page Table)

- někdy je potřeba namapovat rámce zpět na virtuální stránky
- prohledávat tabulky stránek je neefektivní (miliony záznamů)
- slouží k tomu převrácená tabulka stránek (tabulka pevné velikosti podle počtu rámců)
- k vyhledávání slouží pomocná hash tabulka (Hash Anchor Table)
- mapuje se adresa (případně PID)

2.11. Mapování souborů I/O do paměti

2.11.1. Soubory

- operace `open`, `read`, `write` mohou být pomalé (systémové volání)
- mechanismus, který je použitý pro práci se sekundární paměti, lze použít pro práci se soubory
- soubor se načítá do paměti, po blocích velikosti stránky podle jednotlivých přístupů (demand paging)
- k souboru se přistupuje pomocí operací s pamětí (přiřazení, `memcpy`, ...)
- data se nemusí zapisovat okamžitě (ale až s odmapováním stránky/souboru)
- více procesů může sdílet jeden soubor → sdílená paměť (WinNT)
- možnost použít `copy-on-write`

2.11.2. I/O

- lze namapovat zařízení do paměti (specifické oblasti) a přistupovat k němu jako k paměti
- pohodlný přístup, rychlý přístup
- např.: grafické karty

2.12. Poznámky na závěr

- jádro může požadovat souvislý blok rámců
- stránkovací tabulky jsou opět jen stránky → můžou být odsunuty?
- spolupráce s cachí
- základní algoritmy
- realně se implementují složitější metody (heuristiky)

3. Implementace virtuální paměti

3.1. Úvodem

- x86/i386
  - budeme uvažovat jen 32-bitové procesory z rodiny i386 (kompatibilní a novější)
  - podpora starších módů pro kompatibilitu (budemem zanedbávat)
- AMD64
  - označení rodiny procesorů (Intel EM64-T)
  - rozšíření i386 na 64 bitů (long mode)
  - registry rozšířeny na 64 bitů (např.: EAX → RAX); větší počet
  - možnost adresovat rozsáhlejší paměť
- procesory se vyvíjely několik desetiletí (procesor 80386 uveden 1995) → relikty minulosti
- kombinují segmentaci i stránkování

3.2. Úrovně oprávnění

- čtyři úrovně oprávnění (rings)
- nelze používat některé instrukce
- zabraňují aplikacím poškodit systém
- ring 0 - nejvyšší oprávnění (jádro)
- ring 3 - nejnižší oprávnění (aplikace)
- nejčastěji se používá kombinace ring 0 + ring 3
- mikrokernel; virtualizace (XEN)
- přechod mezi úrovněmi přes brány (gates)

3.3. Typy adres na procesorech i386

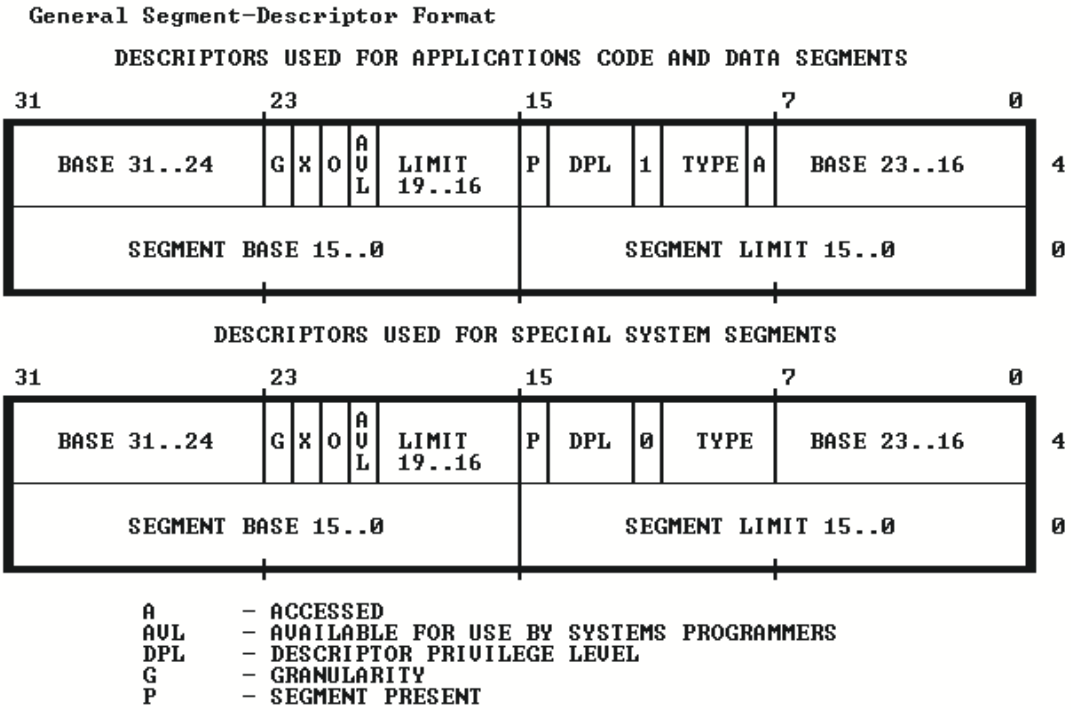
- logická adresa - vidí aplikace; 48 bitů (16 selektor segmentu; 32 offset); segment je často implicitní
- lineární (virtuální) adresa - v adresním prostoru procesoru (32 bitů)
- fyzická adresa - „číslo bytu“ přímo v primární paměti (32 bitů, s PAE 36); sdílená dalšími HW zařízeními; (nevyužité adresy mohou mít další použití SWAP)

3.3.1. PAE: Physical Addrss Extension

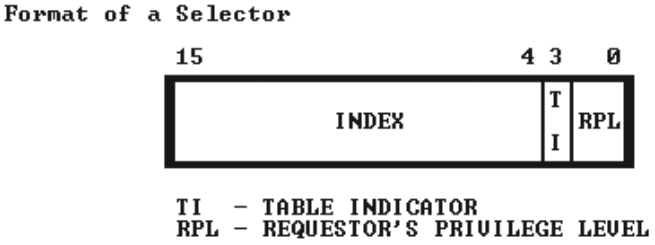
- umožňuje rozšířit využitelnou paměť RAM z 4 GB na 64 GB (Pentium Pro a novější)
- přesměruje část adresního prostoru do jiné části fyzické paměti
- změna formátu segmentových deskriptorů
- stránky 4 kB/2 MB
- změna na úrovni OS (případně ovladačů)
- bez úprav jednotlivé procesy stále omezeny na 4GB (AWE)
- mimochodem přidává podporu pro NX bit

3.4. i386: Segmentace

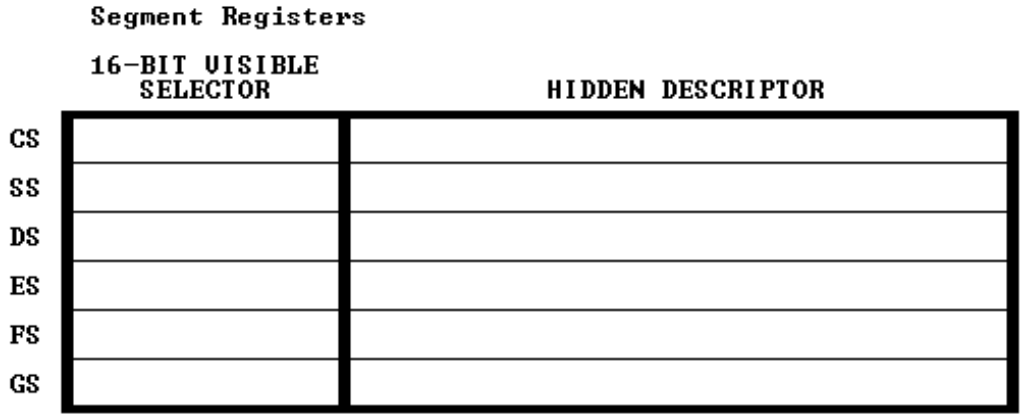
- paměť je možné rozdělit na segmenty (kód, data, zásobník, atd.)
- pro každý segment lze nastavit oprávnění (ochrana paměti)
- segmenty jsou popsány pomocí deskriptorů 8 B záznam
  - báze
  - limit (velikost segmentu)
  - požadované oprávnění (ring 0-4)
- deskriptory segmentů uloženy v:
  - Global Descriptor Table (GDT) - sdílená všemy procesy
  - Local Descriptor Table (LDT) - každý proces má vlastní
- každá může mít až 8192 záznamů
- přístupné přes registry GDTR, LDTR
- první záznam v GDT „null“ deskriptor
- granularita (stránky vs. byty)



- do segmentových registrů (CS, SS, DS) se ukládá selector (16 bitů) - ukazatel do GDT nebo LDT

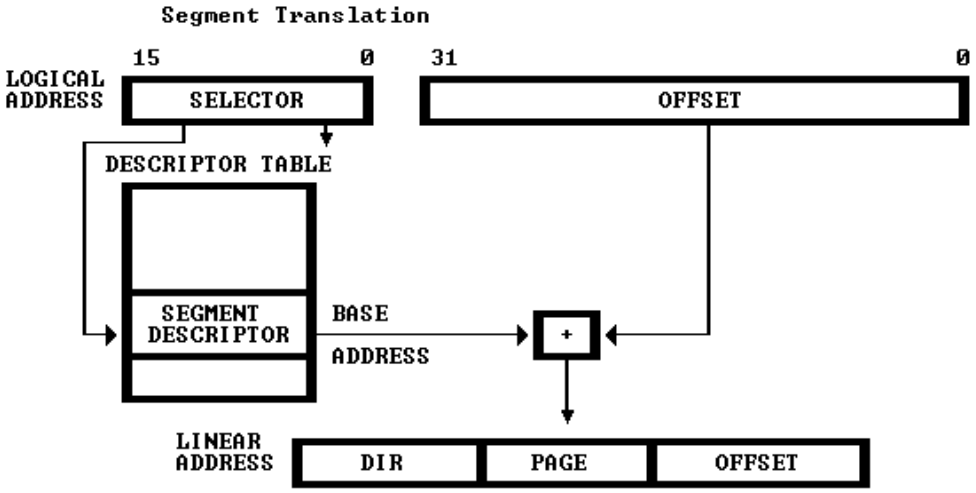


- při načtení se do segmentového registru načte i deskriptor (ale nejde k němu explicitně přistupovat



3.5. i386: překlad adres I. (segmentace)

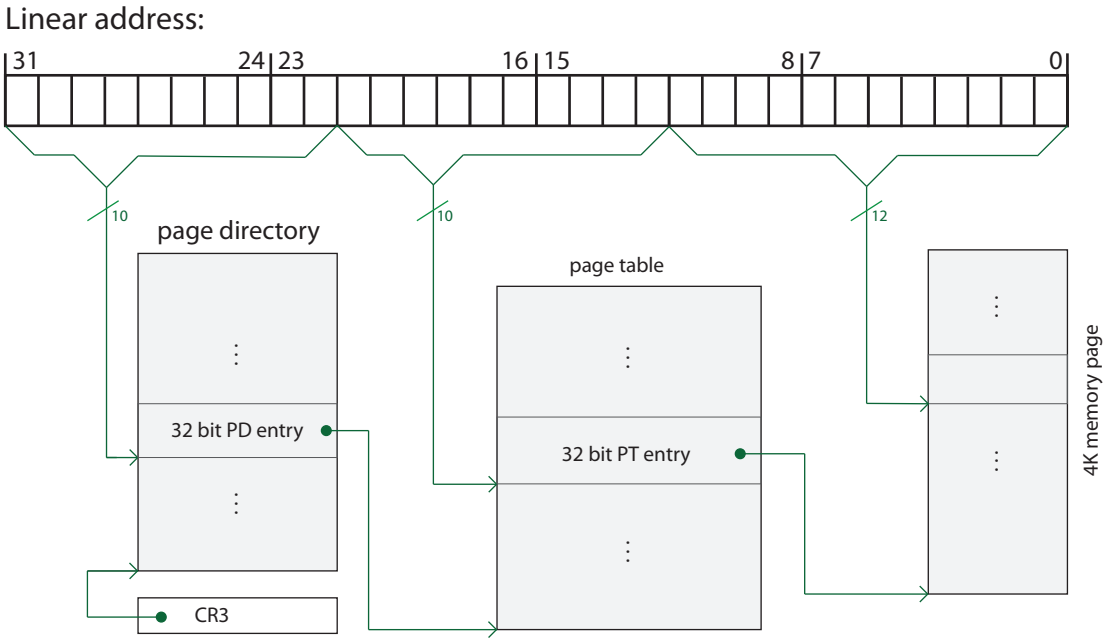
- logická adresa → lineární adresa (segmentace)
- ověří se oprávnění a limit (přístup za hranici segmentu) → neoprávněný přístup
- báze segmentu je načtena s offsetem → lineární



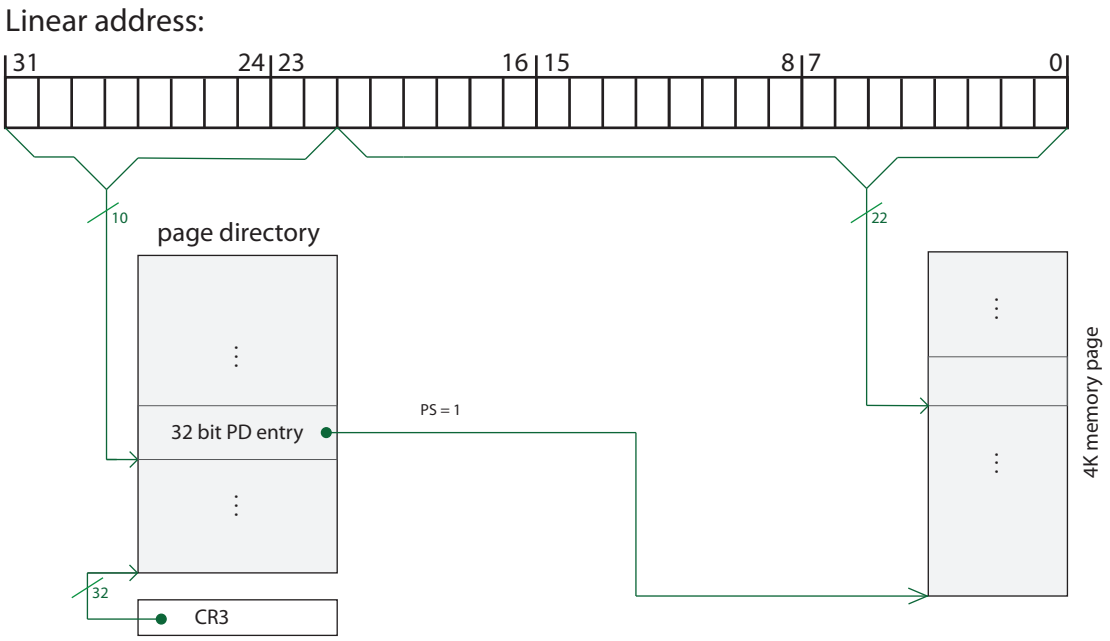
-
- lineární adresa → fyzická adresa

- standartní stránka/rámec: 4 KB
- hiearchická struktura
  - adresář stránkovacích tabulek (Page Directory)
  - adresář stránek (Page Tables)
  - offset
- adresáře mají velikost jedné stránky, každá položka 4 B → 1024 záznamů
- lineární adresa rozdělena na 10 + 10 + 12 bitů (PDI + PTI + offset)
- maximální kapacita 4 GB
- adresa v PDT v CR3 (zarovnané na celé stránky)
- nastavení příznaku v PDT (pro adresu rámce se používá jen 20 b), lze obejít přepočet přes PT a používat stránky velikosti 4 MB (zbytek adresy je offset)
- velikost stránek lze kombinovat

3.6. i386: Překlad adres II. (stránkování - 4 kB stránka)

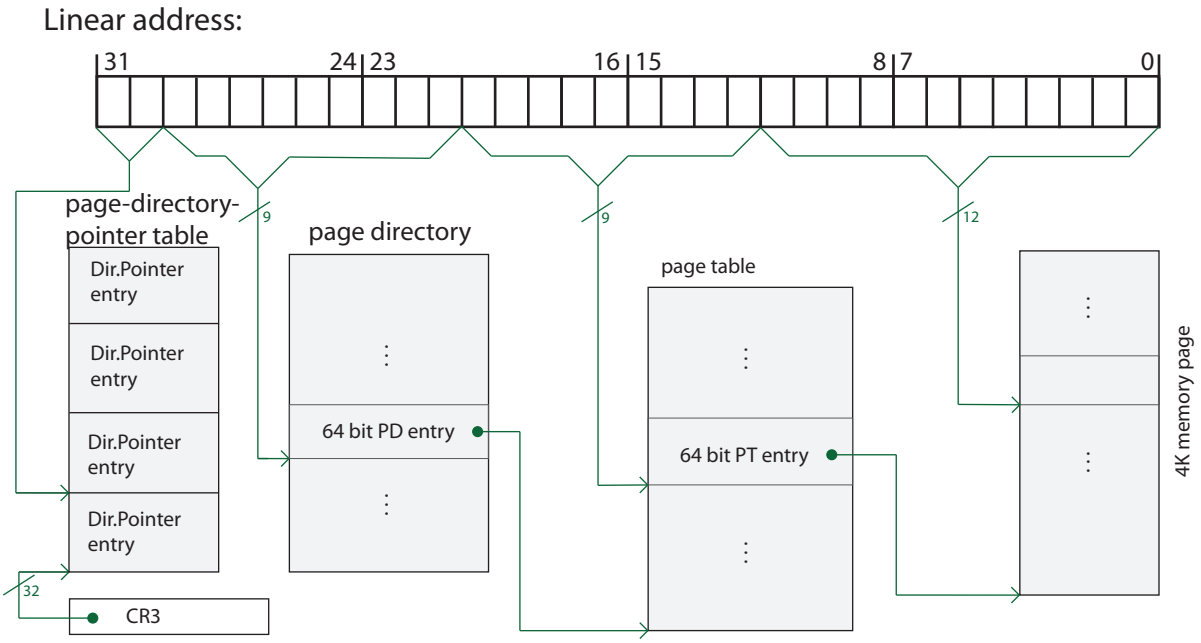


3.7. i386: Překlad adres III. (stránkování - 4 MB stránka)



3.8. i386: Překlad adres IV. (PAE)

- od Pentium Pro
- každá tabulka 4 kB, ale velikost záznamu 8 B → 512 záznamů
- stránkování trojúrovňové
- adresa rozdělena na 2 + 9 + 9 + 12 bitů
  - 2 b - ukazatel na adresář tabulek stránek (Page Directory Pointer Index)
  - 9 b - ukazatel na adresář tabulek stránek
  - 9 b - ukazatel v tabulce stránek
  - 12 b - offset
- velké stránky 2 MB → offset 21 bitů
- potencionální rošíření



3.9. AMD64: typy adres

- segmenty existují, ale nepoužívají se k adresaci, pouze ke kontrole oprávnění
- deskriptor kódového segmentu se používá k přechodu mezi 32 a 64 bitovým režimem
- současné procesory AMD64:
  - 40 bitové fyzické adresy (max. 52; způsob stránkování)
  - 48 bitové logické adresy (max. 64; velikost registrů)
- možnost rozšíření → kanonické adresy
- nejvyšší platný bit je kopírován do vyšších bitů
- dělí paměť na tři bloky

3.10. AMD64: stránkování

- používá se režim PAE
- zavedena čtyřúrovňová hiearchie
- záznamy v tabulkách stránek mají 8 B
- při 4 kB stránkách → $4 \times 9 + 12 = 48$ adresovatelných bitů
- pro 2 MB stránky vynechána jedna úroveň, možnost použít 4 rezervované bity → 52 bitů
- nepoužité bity: nejvyšší - NX bit, ostatní k dispozici OS

3.11. Ochrana paměti (segmenty)

- ochrana paměti na úrovni segmentů je zaplá a nejde vypnout (ale jde nastavit, aby nebyla účinná)
- prováděné kontroly:
  - kontrola typu segmentů (některé segmetny nebo segmentové registry můžou být použité jenom určitým způsobem)
  - kontrola velikosti segmentu (limitu), i. e., jestli program nesahá za hranice segmentu
  - kontrola oprávnění
  - omezení adresovatelné domény (omezení přístupu je k žádoucím segmentům)
  - omezení vstupních bodů procedur (brány)
  - omezení instrukční sady
- AMD64 v long mode neprovádí některé kontroly (báze a limit jsou ingorovány)
- stránkování funguje souběžně se segmentací
- bit pro systémové stránky (zákaz přístupu z ring 3) → volání funkce OS (přes bránu)
- bit pro zákaz zápisu
- AMD64 + PAE mají NX bit (zákaz spouštění) → viry
- možnost nastavit bity na jednotlivých stránkách i adresářích → efektivnější

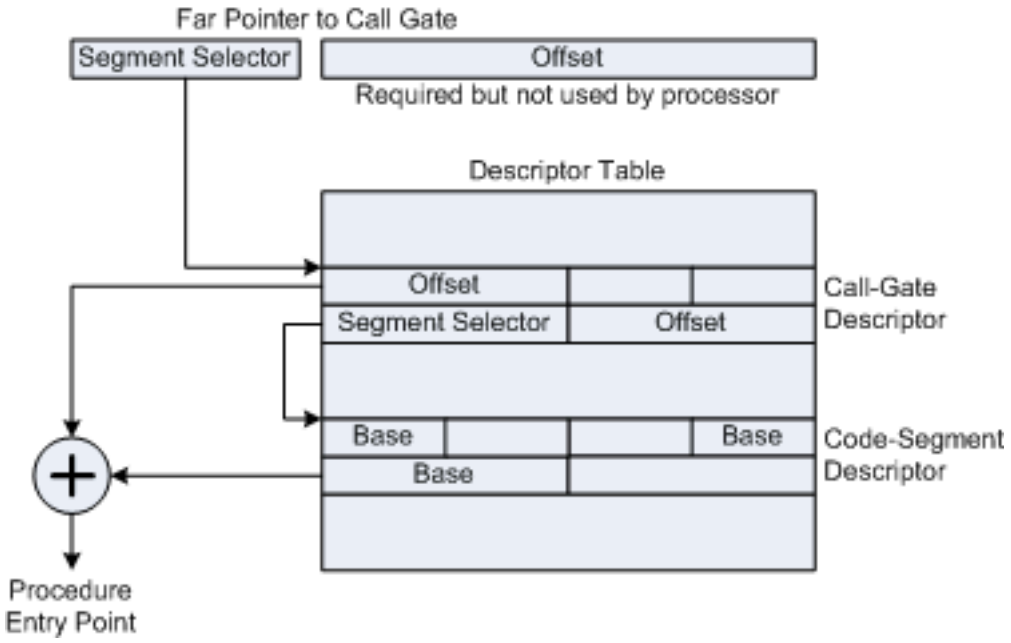
3.12. Úrovně oprávnění

- DPL (Descriptor Privilege Level) - úroveň oprávnění daného descriptoru
- CPL (Current Privilege Level) - aktuální úroveň oprávnění; odpovídá DPL v CS
- RPL (Requested Privilege Level) - úroveň oprávnění daného selektoru (požadovaná úroveň)
- pokud `max(CPL, RPL)` $\leq$ `DPL` je oprávnění v pořádku
- jinak procesor vyvolá vyjímku
- nižší hodnota → vyšší oprávnění

3.13. Ochrana u CS

- přechod mezi kódovými segmenty možný přímo, přes segment TSS, přes bránu
- různé druhy bran: call-, trap-, interrupt-, task- gate (popsané v GDT)
- deskriptor volací brány obsahuje selektor a offset volaného kódu
- ověřuje se DPL brány i DPL kódového segmentu
- podle nastavení „bitu konformity“ se případně použije zásobník pro každou úroveň oprávnění (adresy uloženy v TSS)
- nekomformní → změna oprávnění
- možné použít i k přepnutí režimu procesoru (např.: 16-, 32- bitů)
- ústup od používání → SYSENTER + SYSEXIT

3.14. Brány pro CS



3.15. Implementace v NT na i386

- paměť 2 GB : 2 GB (systém + proces); lze změnit na 1:3
- je možné používat Address Windowing Extension (AWE) - zpřístupní více než 2 GB
- rozdělení do stránek na volné, rezerovované, komitované → demand paging (nulování stránek)
- množina pracovních rámců (50 - 345); balance manager
- stránky se odswapovávají podle přístupového bitu
- načítá několik stránek současně (clustering)
- logical prefetcher umožňuje urychlit start stystému (sleduje přístup na FS a vytváří log)
- používá se segmentace i stránkování
- procesy jsou navzájem oddělené (LDT)

3.16. Implementace v Linuxu na i386

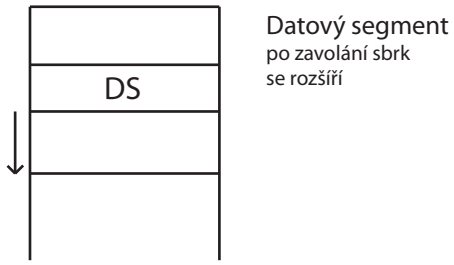
- paměť 1 GB : 3 GB (systém + proces)
- paměť rozdělena na zóny (procesy, DMA, highmem)
- podpora NUMA
- stránky v několika frontách (active, inactive)
- rozdělení stránek na volné, rezerovnané, kombinované → demand paging
- OOM killer

4. Aplikace a práce s pamětí

4.1. Motivace

- aplikace pracují s pamětí jinak než OS → stránky vs. objekty
- zásobník a halda (heap)
- rozdílené požadavky i rozhraní
  - C: `malloc, free, realloc,...`
  - C++: operátory `new, new[], delete, ...`
  - Java: `new`
  - List a Scheme: `cons, vector, ..., lambda?`
- není náležitost OS → běhové prostředí (JVM, CLR), standartní knihovna (libc - dlmalloc, ptmalloc, Hoard, TCMalloc [Google], jemalloc [FreeBSD])
- lze vyměnit (mohou být specifické pro OS i jednotlivé aplikace)
- záleží na nárocích (e.g., typické velikosti objektů, počtu vláken, tolerovaná režie, rychlost, míra fragmentace)
- potřeba řešit cache, lokalitu dat, TLB, atd.

4.2. Alokátor v GNU/(Linuxu): ptmalloc

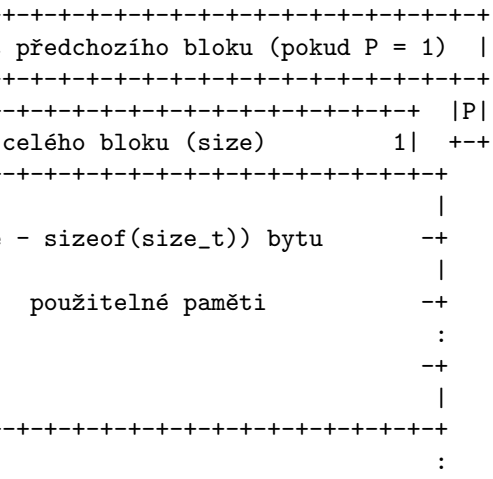
- ptmalloc v. 3 - knihovna v C
- založená na „Douf Leaś Malloc“ (dlmalloc, který je datovaný až do roku 1987)
- přidává podporu více vláken a SMP
- „optimalizovaný“ pro běžné použití
- dokumentace → zdrojový kód
- rozhraní poskytující operace `malloc, free, realloc, ...`
- získává souvislý blok paměti přes volání OS `brk/sbrk` (mění velikost datového segmentu o daný počet bytů)
- 


```
void * simple_malloc(int size) {
    return sbrk(size);
}
```

- uvolnění přes `sbrk(-velikost)` → jednoduchý, ale fragmentuje paměť
- (velké bloky) případně `mmap` (mapování anonymní paměti, `/dev/zero`)

4.2.1. ptmalloc: základní vlastnosti

- každé vlákno může mít svůj alokátor
- `malloc` přiděluje menší kousky paměti
- alokuje po blocích zaokrouhlených na dvojnásobek slova (slovo = 32/64 bitů, dle platformy), tj. 8 B (minimálně)
- některé platformy přímo vyžadují zarovnání paměti
- navíc hlavička
  - velikost předchozího bloku (pokud je volný)
  - velikost celého bloku
  - příznak jestli je předchozí blok volný (uložen ve velikosti - spodní bit; u nově alokovaného bloku je vždy 1)
- každý kus paměti začíná na sudém násobku slova
- vrácená paměť začíná na sudém násobku slova
- nejmenší alokovaný kus paměti na i386 je (16 B), i. e. 12 B pro data, 4 B pro hlavičku
- není rozdíl mezi malloc(1) a malloc(6)! //z duvodu alokace po nasobcich slovech



uvolnění paměti a free

- udržuje si informace o volných blocích
- `malloc` se nejdřív snaží najít první vhodný blok, který byl uvolněn
- volné bloky evidovány
  - do velikosti 265 B v 32 frontách (oboustraný spojový seznam) každá pro jednu velikost
  - větší bloky v 16 stromech (trie), každý pro bloky o velikosti $2^n - 2^{n+1}$, pro $n = <8, 23>$; (uspořádané velikosti)
  - speciální strom pro ostatní (větší bloky)
- při zavolání `free`
  - ověří se jestli je následující blok vonlá (a případně se sloučí)
  - ověří se jestli je předchozí blok volný (a případně se sloučí)
  - zařadí se do příslušného seznamu/stromu
- nikdy nejsou dva volné bloky vedle sebe
- pokud `malloc` vybírá z volných bloků, vybírá z nálsedujících strategií

    * v seznamech: FIFO
    * ve stromech best-fit

```
– chunk ->     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
               |   Size of previous chunk                            |
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  'head:'      |   Size of chunk, in bytes                           |
  nem ->       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
               | Forward pointer to next chunk list                  |
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
               |Back pointer to previous chunk in list               |
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
               | Unused space (may be 0 bytes long                   .
               .                                                     .
               .                                                     |
  next chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  'foot:'      |   Size of chunk, in bytes                           |
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ostatní operace

– **realloc(ptr, size)**

    * pokud je požadovaném bloku místo, nemění velikost
    * jinak kopíruje data do nového bloku a starý uvolní
    * není dobrý nápad:

```
int i = 0;
char * buf = malloc(sizeof(char));
while((c = getc(stdin)) != EOF) {
    buf = realloc(buf, (++i) * sizeof(char));
    buf[i] = c;
}
//blok se bude neustále stěhovat po paměti a bude docházet stále ke kopírování
```

– **Další související operace**

    * `mallopt` - nastavení vlastností alokátoru
    * `posix_memalign` - získaná paměť zarovnana na stránky

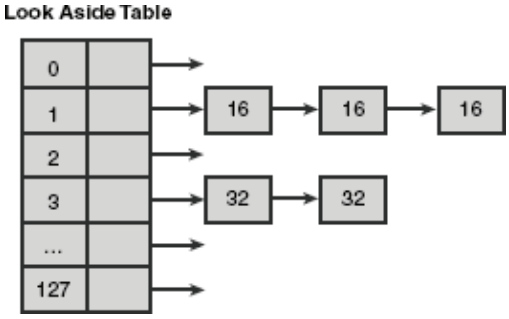
4.3. Alokátor ve Windows

- API + (Frontend) + Backend; jeden proces může mít několik heapů
- viz SolRu p. 731
- každý objekt obsahuje hlavičku (8 B) jako v `ptmalloc`

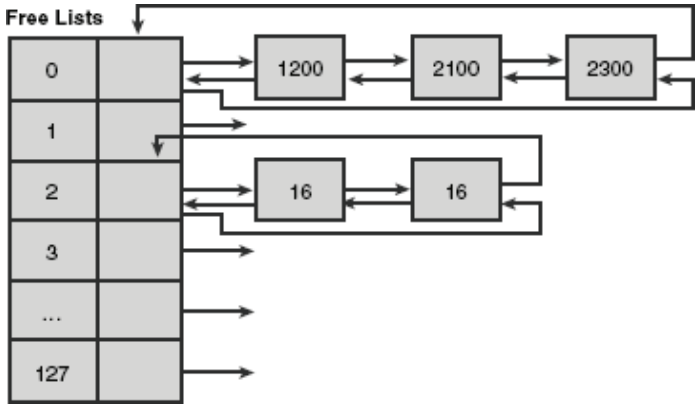
Frontend

- optimalizace alokací menších bloků
- „přichystané bloky použití“
- LAL (Look-Aside-Lists): 127 seznamů s objekty velikost $n \times 8$ B //do XP
- LF (Low Fragmentation): jako LAL, ale až pro objekty do 16 kB (větší objekty → větší granularita)

  - objekty 1 B - 256 B: zarovnány na 8 B
  - objekty 257 B - 512 B: zarovnány na 16 B
  - objekty 513 B - 1024 B: zarovnány na 32 B
  - objekty 1025 B - 2048 B: zarovnány na 64 B
  - ...
- používá se LAL i LF (od Vista implicitní)



backend

- podobný `ptmalloc`
- velké bloky se alokují přímo přes VM
- pokud není k dispozici malý blok, rozdělí se větší
- 

4.4. Automatická správa paměti

- Garbage Collector: John McCarthy vyvinul pro LISP (1958)
- renesance (main stream) v průběhu devadesátých let (Java)
- zjednodušení vývoje aplikací na úkor režie počítače
- jinak přítomna v ostatních jazycích i dřív
- centrum pozornosti → Java, .NET, skriptovací jazyky
- Garbage Collector lze doplnit i do C/C++
- Boehm(-Demer-Weiser) garbage collector (okamžitá náhrada GC_MALLOC)

Počítání odkazů

- každý objekt obsahuje počítablo, kolik na něj odkazuje objektů
- pokdu je na něj vytvořený/zrušený odkat, zvýší/sníží se počítadlo o 1
- v případě, že je počítadlo 0 → objekt je uvolněn
- problém: režie (paměť i CPU); cyklické závislosti

- Delphi, PHP, Python, COM

4.4.1. Mark and Sweep

- populární typ GC
- každý objekt má příznak, jesli se používá (nastavený na 0)
- v průběhu „úklidu“
  - objekty odkazované na zásobník, statických proměnných (popř.: registrů) → označeny jako používané
  - objekty odkazované z označených objektů → označeny
  - neoznačené objekty uvolněny

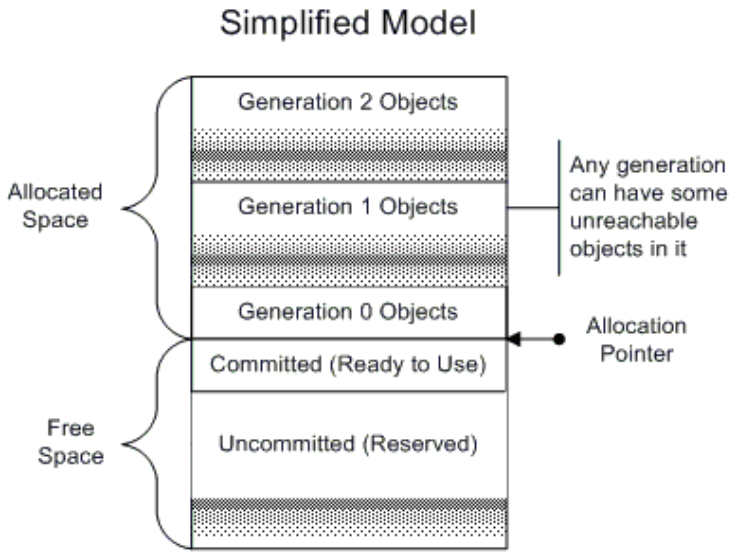
Abstrakce: tři barvy

- objekty rozděleny do tří množin (černé, šedé, bílé)
- na počátku:
  - bílé kandidáti na uvolnění //na smazání
  - šedé - obejkty odkazované z „kořenů“ (určené k prověření) //musime prověřit, zdali jsou používané
  - černé - nemají odkazy na bílé objekty //jsou používané
- krok:
  - vezme se jeden šedý obejkt a je přebarven černě
  - z něj odkazované objekty přebarveny šedě
  - opakuje se, dokud existují šedé objekty

4.4.2. Varianty GC

- přesunují vs. nepřesunují (moving/non-moving)
  - lze přeskupit obejkty, aby vytvořily souvislý blok
  - režie přeskupení/snížení fragmentace → rychlejší alokace
- generační
  - předpokládá se, že krátce žijící objekty budou žít krátce

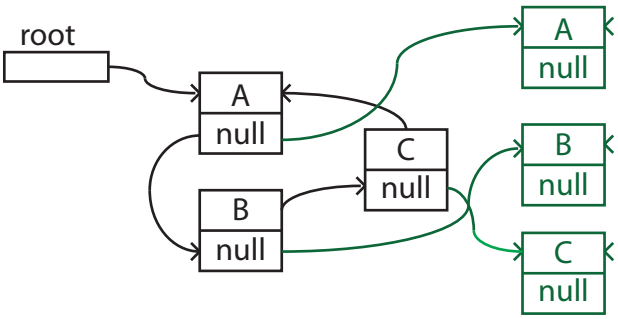
– paměť rozdělena na generace → nad staršími objekty nemusí probíhat sběr příliš často



- přesné vs. konzervativní
  - existují informace o uložení ukazatelů? (JVM)
  - alternativně lze určit, jestli daná část objektu je ukazatel (Boehm GC)
- inkrementální vs. stop-the-world
  - problém se zastavením běhu programu, aby mohlo dojít k „úklidu“ (změny odkazů)
  - inkrementální redukují výše zmíněný problém
  - možné vylepšit paralelním zpracováním

stop-the-world - musí se program zastavit a garbage collector mezi tím provede úklid

4.4.3. Kopírující GC

- mark-and-sweep - problém s fragmentací
- alternativní řešení:
  - paměť rozdělena na dva stejně velké bloky (aktivní, neaktivní)
  - při úklidu se objekty přesunují z aktivní části do neaktivní (nové místo)
  - po přesunu všech živých objektů dojde k prohození aktivní a neaktivní části
  - problém: přeuspořádání ukazatelů
  - výhoda: odstranění fragmentace + pracuje se jen s živými objekty (cache, TLB)
  - 

- kombinace obou: makr-and-compact

4.4.4. Manuální vs. automatická správa paměti

Manuální

- režie (CPU): malloc //náročná operace - alokuje paměť + se ještě dívá na volné bloky, které by mohl použít
- uniklá paměť (Valgrind)
- používání uvolněné paměti, dvojité uvolnění
- možná finalizace objektů //můžeme při smazání objektu uzavřít např.: síťové spojení atd.
- neexistuje problém stop-the-world

Automatická

- rychlejší: malloc //rychlejší - často jen posune pointer dopředu, místo prohledávání
- pomalejší uvolnění paměti (někdy může být výhodné uvolňovat obejkty naráz)
- paměť může taky „unikat“ (statické proměnné)
- problém s finalizací obejktů //může se stát, že objekt není používaný a stejně je v paměti stále alokovaný
- nedeterminismus
- `private static List log` - GC bude očekávat, že všechna data jsou živá a tudíž žádná neodstraní → únik paměti

4.4.5. Zásobník, Halda, Regiony, Pooly, atd.

- práce se zásobníkem je rychlejší než s haldou
- hodnotové typy C++, C#; možné ovlivnit umístění (zásobník × heap)
- např.: v Javě místo vytvoření objektu nelze kontrolovat (Heap?)

Escape analysis

- analýza jestli daný objekt/ukazatel může „uniknout“ z daného kontextu (např.: funkce)
- záležitosti překladače
- možnost převést alokaci heapu na zásobník
- možnost rozbít obejkt hodnot na primitivní datové typy

Regiony/Arény/Pooly

- region/aréna - alokuje se velký blok paměti
- přiděluje se místo z tohoto bloku (může být požadovaná jednotná velikost objektu)
- předpokládá se, že objekty budou uvolněny společně
- menší režie na alokaci i uvolnění než v případě malloc/free

Pozor na předčasné optimalizace !!!

4.5. Atomatický přístup do paměti

- vícevláknové aplikace/víceprocesorové počítače (+ out-of-order provádění aplikací)
- obecné přístupy do paměti nemusí být atomické (záležitost CPU)
- klíčové slovo `volatile` - často záleží na překladači; zarovnání přístupů
- *memory barriers* umožňují vynutit si synchronizaci (záležitost CPU)

Atomické operace

- Compare-and-Swap (CAS): ověří jestli je daná hodnota rovná požadované a pokud ano, přiřadí ji novou (CMPXCHG)
- Fetch-and-Add: vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna (XADD)
- Load-link/Store-Conditional(LL/CS): načte hodntou pokud během čtení nebyla změněna, uloží do ní novou hodnotu
- umožňují implementovat synchronizační primitava (mutex, kritické sekce, atd.)

4.6. Transakční paměť

- atomické operace umožňují implementovat bezzámkové (lock-free) datové struktury → netriviální
- souběžný přístup k paměti → vývoj vícevláknových aplikací komplikovaný
- vlákna vs. procesy
- problémy se synchronizací (zámky nejsou reentrantní) //zámek je jednou zamčený a nejde zamknout třeba 3 krát
- ale: dtabázové systémy zvládají paralelně pracovat s daty bez vážnějších problémů!!!
- rozdělení programu na části, které jsou provedeny „atomicky“ (ACI)

```
void transfer(Account a1, Account a2, int amount) {
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
    }
}
```

// rozdeleni do transakci, buďto se provede cela transakce, nebo se neprovede vůbec, pokud jedna transakce provadi zmenu, jina transakce nevidi zmenu, dokud neni provedena

- změny se neprovádí přímo, ale ukládají se do logu, v případě „commitu“ se ověří, jestli došlo ke kolizi
- zjednodušení vývoje - na úkor režie //transakce se provádí vícekrát
- omezení: transakci musí být možné provést vícekrát //pokud prevedeme penize a odpalime rakety, muze dojit ke kolizi a je nutno provest danou akci znovu

```
void transfer(Account a1, Account a2, int amount) {
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
        launchTheMissiles();
    }
}
```

- možné explicitně ovládat transakce `retry, orElse`
- problém: jak se vypořádat s hodnotami mimo transakční paměť (např.: existující knihovny)
- různé varianty a implementace
- podpora HW pro transakční paměť → omezená až žádná (processor ROCK? [SUN])
- podporovat na straně softwaru (Software Transactional Memory - STM)
- podpora jako knihovny /rozšíření Javy/C#/C++
- výzkum Haskell [Microsoft], Fortress/DSTM [SUN] - projekt již zrušen
- jazyky zaměřené na paralelní zpracování, např.: Clojure

5. I/O zařízení

- zásadní složka Von Neumanovy architektury
- různé pohledy na I/O zařízení: inženýrský (dráty, motory) vs. programátorský (rozhraní)
- různé rychlosti od 10 B/s (klávesnice) po 1 GB/s (PCI Express)
- různé druhy přístupu (bloková, znaková, ostatní)

5.1. Rozdělení zařízení dle přístupu

Bloková zařízení

- data jsou přenášena v blocích stejné velikosti (typicky 512 B až 32 kB)
- možné nezávisle adresovat (zapisovat/číst) data po jednotlivých blocích //pokud chceme číst např. 1 B, musíme přečíst celý blok
- HDD, SSD, CD, DVD, páska, ...?

Znaková zařízení

- proud znaků/bytů (nelze se posouvat)
- klávesnice, myš tiskárna, terminál

Ostatní

- nespadají ani do jedné z kategorií
- hodiny (přerušení), grafické rozhraní (mapovaná paměť)

5.2. Přístup k zařízením

Port-Mapped I/O

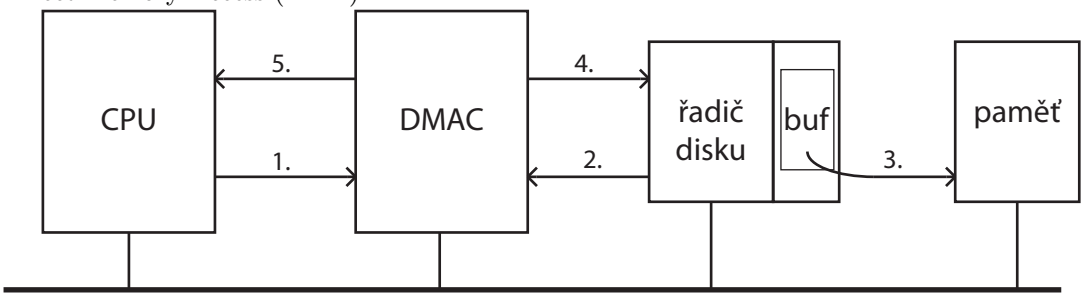
- výhodné pro jednoduchá zařízení, kde se nevyžaduje příliš komunikace s zařízením
- registry jednotlivých zařízení mají samostatný adresní prostor (oddělený od paměti)
- přístupné přes operace `in, out` - zápis/čtení hodnoty z portu
- nevýhody: omezení na speciální operace (jen zápis/čtení), omezené řízení přístupu

Memory-Mapped I/O

- registry jednotlivých zařízení jsou namapovány v paměti
  např.: `out 0x01, eax` ⇒ roztočí se disk
- data se čtou přímo na sběrnici, zapisuje je na sběrnici zařízení, ne do paměti
- výhoda: k zařízení se přistupuje jako k paměti (možné používat všechny instrukce), řízení přístupu - lze použít to, co se používá pro paměť
- problém: cache, oddělená sběrnice pro paměť
- rozdělení paměti na oblasti: 640 kB, 3 GB //zařízení jsou mapovány v určitých částech pamětí - z historických důvodů

5.3. Přístup k zařízením: přenos bez účasti CPU

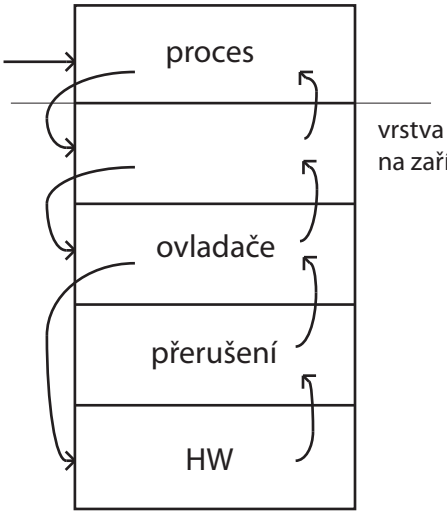
- přesun dat v přechozích případech vyžaduje účast CPU ⇐ neefektivní
- čtení z disku
  - řadič disku dostane požadavek: čtení
  - dist načte do interního bufferu data
  - řadič disku vyvolá přerušení
  - CPU čte postupně data z bufferu a ukládá do paměti //hrubě neefektivní, protože vyžaduje čas CPU
- Direct Memory Access (DMA)



  1. řadič DMA (DMAC) dostane požadavek: čtení + cílovou adresu
  2. předá požadavek řadiči disku
  3. zapisuje data do paměti
  4. dokončení je oznámeno řadiči DMA
  5. DMAC vyvolá přerušení
- různé varianty (např.: přenos přes řadič DMA)
- spolupráce DMA s MMU
- PCI zařízení nepotřebující DMA (PCI Bus Master)
- část paměti, která je strankovaná se zdá být jako souvislá, což fyzicky tak být nemusí

5.4. Přístup k I/O z aplikace

- OS by měl zajisti přístup k zařízením uniformním způsobem bez ohledu na zařízení (IDE, SCSI, CD)
- např.: zápis bloku dat, přečtení znaku //specifikované rozhraním OS
- API, zápis do speciálních souborů (Linux, např.: `/dev/hda`) //zápisem nebo čtením souborů přistupujeme přímo k datům na disku, jak jsou uložena fyzicky je záležitost OS
- Linux: major (ovladač) a minor (zařízení) číslo zařízení

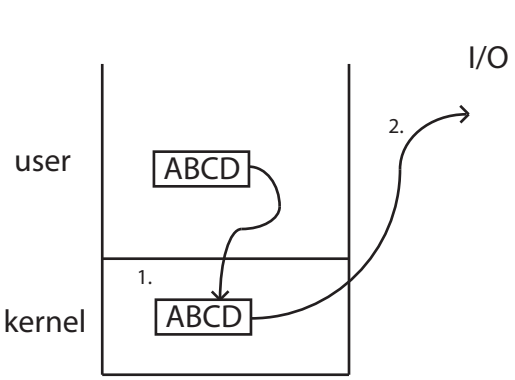


- proces žádá o přístup k HW
- 2. vrstva se postará o vyrovnávací paměť, zařízení přístupu komunikuje s ovladačem
- ovladač konkretizuje požadavek a pošle HW
- po vyřízení požadavku HW vyřídí přerušení
- ovladač předá vrstvě 2 výsledek požadavku
- vrstva předá výsledek procesu

- synchronní/asynchronní (blokující/neblokující) přístup k zařízení //neblokujíc přístup - operace čtení se podívá na vstup, jestli jsou na vstupu data, pokud ano přečte, u blokujícího je operace zablokována a čeká na vstupu na čtení
- výlučný/sdílený přístup //u sdíleného přístupu je potřeba zajistit synchronizaci
- rozlišujeme 3 přístupy:

Aktivní čekání

- data se kopírují z bufferu do registru
- podle stavového registru se čeká až budou přenesena
- jednoduchá implementace, ale neefektivní



I/O s přerušeními

- není nutné čekat na přenos dat
- v průběhu přenosu může procesor provádět další činnost
- data předána jádru, aplikace zablokována
- přenos dat řídí obsluha přerušení

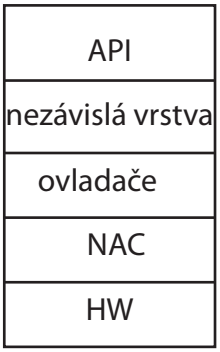
I/O přes DMA

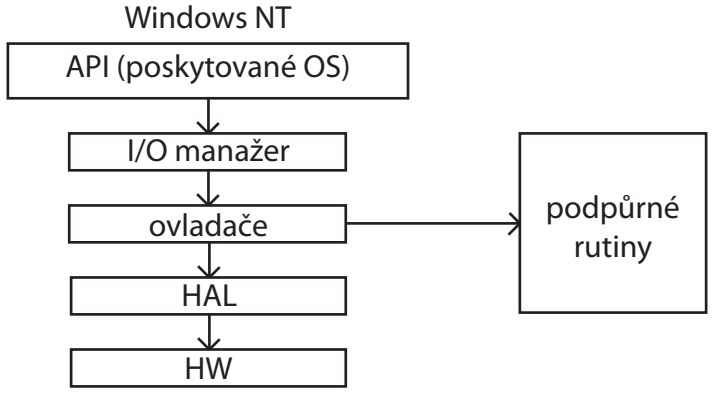
- analogické přerušením, ale přenos dat řídí řadič DMA ⇒ míň přerušení

Bufferování

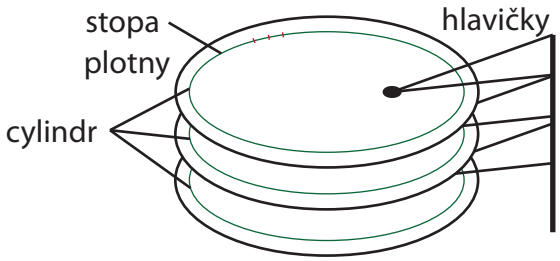
- optimalizace přenosu dat - zpoždění zápisu/čtení //aby se nemuseli přenášet jednotlivé b nebo B

5.5. Ovladače zařízení



- zajišťují přístup k zařízení ⇒ zápis, čtení, inicializace, správa napájení, logování
- typicky součástí jádra OS (může být i v uživatelském prostoru ⇒ oddělení ovladačů)
- zkompilování do jádra vs. dynamické načítání
- měl by být dodán výrobcem HW ⇒ definovaný model ovladačů (např.: bloková zařízení)
- ⇒ spolupráce s ostatními částmi OS, sdílení funkcionality
- zjednodušení vývoje ovladačů, jednotný přístup aplikací
- Hardware Abstraction Layer (HAL) - stará se o odstínění konkrétního disku (od všech disků), grafické karty atd
- Windows, v Linuxu (per se není)
- 

5.6. Bloková zařízení: HDD



- disk - plotny, stopy (⇒ cylindry), sektory (typicky 512 B)
- původně se adresovaly sektory ve formě CHS (praktická omezení, mj. velikost disku) //Cylidnr, Head, Sector
- nahrazeno LBA (logical block addressing) //navenek jeden velký spojitý blok dat, vevnitř se řeší podrobněji
- low-level formát ⇒ hlavička + data + ECC
- přípojené typicky přes (P)ATA, SATA
- rychlost přístupu ovlivňuje:
  - nastavení hlavičky na příslušný cylindr (seek time, nejzásadnější, řády ms)
  - rotace (nastavení selektoru) pod hlavičkou
  - přenosová rychlost komunikace s daným zařízením
- nezávislá cache (hromadí požadavky ⇒ eliminuje přesuny hlavičky)

5.6.1. Optimalizace

- více požadavků se bude řešit najednou
- místo sektorů se pracuje s clustery sektorů (velikost podle velikosti disku)
- cache disku
- cache OS ⇒ společně s VM; cachuje se na úrovni FS
- odpovídající algoritmy - LRU, LFU, . . . , jejich kombinace
- zjednodušení OS ⇒ otevření souboru ⇒ namapování do cache, demand paging
- write-through cache: data se po zapsání zapisují přímo na disk
- write-back cache: data se zapisují až po čase (pozitivum: možnost optimalizací zápisu, negativum: závislá na zdroji napájení, pokud dojde napájení data zapsáná v cache jsou ztracena)
- vynucení uložení cache (flush)
- sekvenční čtení
  - read-ahead - data se načítají dopředu
  - free-behind - proaktivně uvolňuje stránky při načítání nových
- „spolupráce“ - OS a HW (spoon-feeding), databáze

5.6.2. Algoritmy přístupu

Varianty

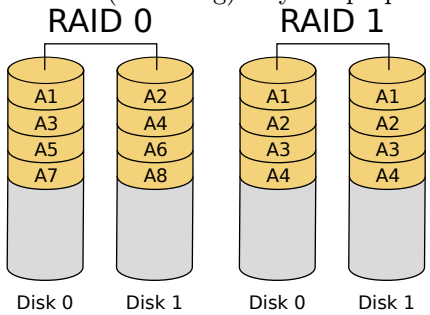
- dnes již přesunuto do řadiče disků
- FCFS (First Come First Serve)
- SSFT (Shortest Seek Time First) - vybrán ten, kam bude bude nejrychlejší přesun (má tendenci zůstat uprostřed)
- SCAN (výtahový algoritmus) - raménko je systematicky přesouváno od jednoho okraje k druhému postupně zapisuje data (data uprostřed zapisována rychleji)
- C-SCAN - jako SCAN, ale data se zapisují jen v jednom směru
- PLUG a C-LOOK - jako SCAN a C-SCAN, ale nepřesunují hlavičky až na konec, když tam není požadavek

Poznámky

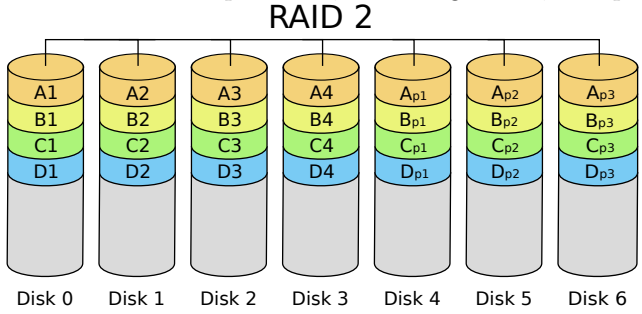
- SSFT vhodný pro sekvenční práci, SCAN pro zatížené systémy
- zmíněné algoritmy počítají s „virtuální reprezentací“ disku

5.7. Bloková zařízení: RAID

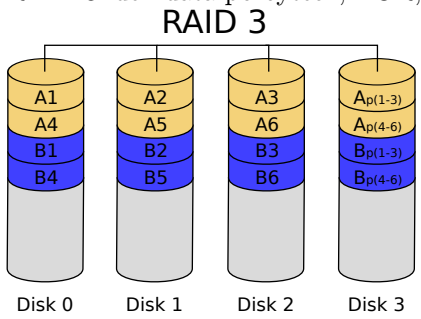
- SLED: Single Large Expensive Disk
- RAID: Redundant Array of Inexpensive/Independent disks
- Mean Time to Failure: $MTTF_{pole} = MTTF_{disk}/N$ //střední doba času poruchy děleno počet
- hardware vs. software RAID
- RAID-0 (stripping): zvýšení propustnosti, problém selhání pořád existuje
- RAID-1 (mirroring): zvýšení propustnosti (kopie), řeší problém selhání



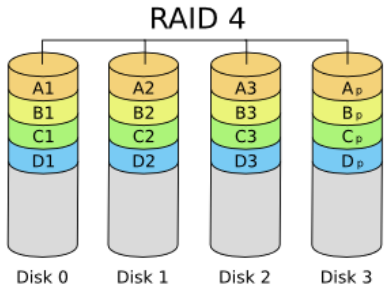
- RAID-2: dělí data po bitech, Hammingův kód, disk pro paritu (nepoužívá se)



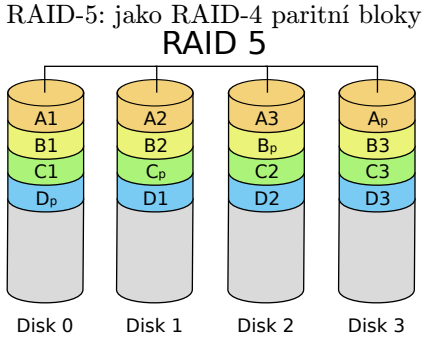
- RAID-3: dělí data po bytech, XOR, disk pro paritu (zátěž), zvládne výpadej jednoho disku



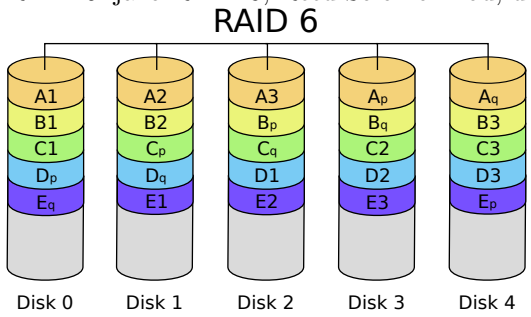
- RAID-4: jako RAID-3 používá bloky (zátěž)



- RAID-5: jako RAID-4 paritní bloky jsou, ale distribuovány



- RAID-6: jako RAID-5, Reed-Solomon kód, dva paritní bloky, výpadek až dvou disků



- kombinace RAID-0+1, RAID-1+0

5.8. Bloková zařízení: SSD, CD, DVD

Solid-state Drivers

- flash paměti, popř.: rozhraní jako HDD
- bez rotujících částí ⇒ rychlý přístup (výrazně víc IOPS) //Input Operations Per Second

- problématický zápis
  - omezení na počet přepsání jednoho místa
  - paměť musí být nejdřív vymazána před zápisem
  - často lze zapisovat po stránkách (řádově kB), ale mazat je nutno po blocích (řádově v MB) ⇒ rychlejší zápis než přepis
- wear leveling //kontrola obnošení
  - žádný - data se přepisují na místě
  - dynamický - změněné bloky označeny jako neplatné a data zapsány jinam (USB)
  - statický - jako dynamický, ale přesouvá i nezměněné data (SSD)
  - softwarová vs. hardwarová implementace (JFFS2, LogFS)
- garbage collection a TRIM

5.9. Compact Disc (CD)

- data umístěna na spirále ⇒ pomalé vyhledávání, rychlé sekvenční čtení
- vysoká redundance dat //pomáhají číst po mechanickém poškození
- symbol - k zakódování 8 b se používá 14 b
- 42 symbolů tvoří rámec o velikost 588 b (192 b data, zbytek ECC) //ECC - samoopravny kod
- jeden sektor obsahující 2048 B dat je tvořen 98 rámci (zahrnuje 16 B hlavičku a 288 B pro ECC)
- efektivita 28 %!

DVD

- analogicky jako CD

5.10. Bloková zařízení: SAN, NAS, NBD

SAN(Storage Area Network)

- umožňuje připojit disky přes síť, aby se jevily jako lokální
- SCSI přes Fiber Channel
- ATA over Ethernet
- iSCSI
- virtualizace a konsolidace úložných zařízení
- možnost řešit výpadky HW

NAS(Network Attached Storage)

- poskytuje zařízení na úrovni souborového systému

NBD(Network Block Device)

- zpřístupňuje blokové zařízení přes síťové spojení

5.11. Znaková zařízení

Terminál

- většina počítačů: klávesnice + monitor $\Longrightarrow$ terminál
- osobní počítače
- síťové terminály
- samostatné terminály (RS-232) $\Longrightarrow$ převod znaků na sériovou linku a zpět //jediná možnost jak komunikovat s terminálem, dokud není spuštěné síťové rozhraní
- vstup z klávesnice a výstup řeší odlišné ovladače
- možnost předávat znaky přímo aplikaci (RAW mode, data jsou přímo posíláná na seriovou linku) nebo počkat (backspace; cooked mode - data se ukladaji na buffer a po čase se pošlou na seriovou linku) $\Longrightarrow$ ovladač musí mít buffer (echoing)
- speciální znaky pro speciální chování (Ctrl+D $\Longrightarrow$ EOF, Ctrl+H $\Longrightarrow$ backspace, Ctrl+ $\Longrightarrow$ SIGQUIT
- aplikace často vyžadují sofistikovaný přístup k výpisu textu (editory)
- $\Longrightarrow$ escape sekvence (rozdíly mezi terminály, ANSI)
- speciální znak ESC (0x1B)
- např.: ESC [nA $\Longrightarrow$ posun kurzoru o n řádků nahoru
- další operace: vkládání/mazání řádků, posun doleva/doprava, změna barvy

5.12. Hodiny(časovače)

- krystal generující pravidelné pulzy (např.: 1000 MHz)
- programovatelný časovač
  - nastavení registru na určitou hodnotu se inicializuje
  - při každém pulzu snížená hodnota o jedna
  - při nula vygeneruje přerušení a zastaví se
- různé funkce
- může jich být víc (příp. možnost emulovat jedním)
  - evidence reálného času
  - plánování procesů (proces nesmí využít víc času, než mu bylo přiděleno)
  - uložení cache
  - systémové volání `alarm`

5.13. Shrnutí I/O

- blokující (synchronní) - aplikace vydá požadavek a je uspána do doby, než je vyřešen
- neblokující - nedochází k uspání, požadavek je vyřešen okamžitě pokud to jde, např.: `read` vrátí dosputná data
- asynchronní - požadavek je předán, v momentě, kdy je vyřešen, je o tom aplikace informována
- buffer - paměť určená k přenosu dat mezi zařízeními (příp. zařízením a aplikací)
  - vyrovnání se s odlišnými přenosovými rychlostmi
  - vyrovnání se různými velikostmi přenášených dat
  - sémantika kopírování (copy semantics) - jádor vs. aplikace
- cache - rychlá paměť, která umožňuje zrychlit přístup k jinak pomalejším zařízení
- cache a buffer - odlišné funcke, i když stejná paměť může být použita pro oba účely

6. Souborové systémy

6.1. Motivace

- potřeba uchovávat větší množství dat (primární paměť nemusí dostačovat)
- data musí být perzistentní (musí přežít ukončení procesu)
- k souborům musí být umožněn souběžný přístup
- $\Longrightarrow$ řešení v podobě ukládání dat na vnější paměť (např.: disk)
- $\Longrightarrow$ data ukládána do souborů tvořících souborový systém (File System/FS)
- soubor jako proud bytů (doprovázen doplňujícími informacemi)
- souborový systém jako abstrakce (odstínění od implementačních detailů)
- $\Longrightarrow$ Unix(dotaženo v „Plan 9 form Bell Labs“)
- zajímavý problém: pojmenování objektů (souborů) & a jejich organizace

6.2. Operace se soubory

- `create` - vytvoření souboru
- `write/append` - zápis do souboru (na konec, popř. přepis); souvislý blok vs. postupný zápis
- `read` - čtení ze souboru (do přichystaného bufferu)
- `seek` - změna pozice
- `erase` - odstranění souboru (uvolnění místa); `link & unlink`
- `truncate` - zkrátí daný soubor na požadovanou velikost
- ne všechny souborové systémy a zařízení podporují všechny operace; např.: CD + ISO9660 (FS) //nejsou k dispozici erase atd.
- operace dostatečně obecné $\Longrightarrow$ přístup k zařízením (disk, klávesnice, terminál); ovládání systému
- $\Longrightarrow$ lze používat existující nástroje pro práci se soubory //`cat /proc/cpuinfo` - zobrazí informace o dostupných CPU
- např.: využití při práci `procfs, devfs, sysfs`
- roury, FIFO
- `open` - otevře soubor, aby s ním šlo manipulovat přes popisovač (file descriptor, file handle) //proč musíme otevírat soubor? potřebujeme pracovat s určitým souborem, tudíž aby jsme nemuseli pokaždé vyhledávat daný soubor na disku
- popisovač - po otevření souboru se vytvoří ukazatel na strukturu v jádře
- přístup přes jméno neefektivní
- „soubory“ nemusí mít jméno
- jeden soubor může být otevřen vícekrát (více ukazatelů na pozici v souboru)
- `close` - uzavře soubor

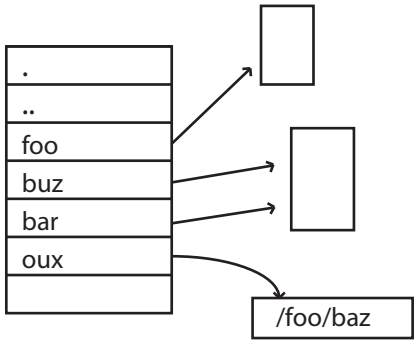
- `get/set attribute` - práce s atributy (metadaty)

Typy souborů

- běžné soubory, adresáře (Unix: speciální soubory pro blokové a znakové zařízení)
- binární vs. ASCII soubory (ukončení řádků)

6.3. Organizace souborů

- soubory jsou rozlišované podle názvů (často specifikované pro daný OS nebo FS)
- rozlišování velkých a malých písmen (Unix vs. MS-DOS a Windows)
- MS-DOS: požadavek na jméno souboru ve tvaru 8+3: jméno + přípona //zůstalo zachováno ve FAT
- rozlišení obsahu souboru
  - podle přípony (Windows-asociace s konkrétní aplikací)
  - magická čísla (Unix - podle úvodních bytů je možné identifikovat typ)
  - podle metadat (informace o souboru jsou uloženy vedle souboru jako součást FS)
- typicky se soubory organizují do adresářů (složek)
- každý adresář může obsahovat běžné (popř. speciální) soubory i další adresáře $\Longrightarrow$ stromová struktura (používaly se i omezenější systémy - žádné, jedna nebo dvě úrovně)
- v zápisu cesty ve stromě se používá lomítek
  - Unix: `/usr/local/bin`
  - Windows: `\usr\local\bin`
- k přístupu k souboru se používají:
  - absolutní cesty: `/foo/bar/baz.dat`
  - relativní cesty: `foo/bar.dat` $\Longrightarrow$ každý proces má aktuální adresář
- operace s adresáři: `Create, Delete, OpenDir, CloseDir, ReadDir, Rename`
- speciání adresáře v každém adreáři „.“ a „..“ (aktuální a nadřazený adresář) $\Longrightarrow$ nutné pro navigaci v hiearchii adresářů
- struktura nemusí být hiearchická $\Longrightarrow$ obecný graf (acyklický, cyklický)
  - hardlink - ukazatel na soubor (jeho tělo/obsah) //výhoda - transparentnos, nevýhoda - odkaz pouze v jednom FS
  - symlink - soubor jako ukazatel na jiný soubor (specifikovaný cestou) //lze používat odkaz na jiný FS, ale nejsou plně transparentní - může vzniknout odkaz na neexistující soubor
  -



- **Unix:** soubor má 2 počítadla, s počtem ukazatelů na soubor, které adresáře se na něj odkazují a které mají daný soubor otevřený, jakmile oba 2 počítdla klesnou na 0, tak lze fyzicky smazat
- v Unixech běžné, ale ve Windows delší dobu (ale chybělo rozhraní)

6.4. Dělení disku

- každý fyzický disk se skládá z jedné nebo více logických částí (partition, oddíl); popsané pomocí partition table daného disku
- v každé partition může existuovat souborový systém (označovaný jako svazek)
- v Unixech je každý svazek připojen (mounted) jako adresář (samostatný svaze pro `/, /home, /usr`)
- $\Longrightarrow$ Virtual File System (VFS)
  - využití abstrakce $\Longrightarrow$ umožňující kombinovat různé FS do jednoho VFS
  - specializované FS pro správu systému (`procfs, sysfs`) $\Longrightarrow$ API OS
  - možnost pripojit běžný soubor jako svazek (i svazek jako soubor)
  - síťové disky (NFS, CIFS)
  - (jako důsledek lze snadno portovat jednotlivé FS)
- ve Windows jednotlivé svazky označeny (a:, b:, c:, . . .); ale funguje i mountování (preferovaný jeden svazek pro vše)

6.5. Struktura souborů

- často je soubor chápán jako proud bytů
- sekvenční vs. náhodný přístup
- někdy může být výhodná i jiná struktura
- rozdělení jednoho souboru na více proudů (např.: spustitelný soubor - kód + data)
- potřebná podpora ze strany FS i OS $\Longrightarrow$ streamy v NT
- společně s daty jsou k souboru připojena metadata (atributy)
  - vlastník souboru
  - přístupová práva
  - velikost souboru
  - příznaky (skrytý, archivace, spustitelný, systémový)

6.6. Přístup k souborům

Zamykání

- sdílený přístup
- omezení přístupu - současné čtení více procesů (zabránění zápisu)

Sémantika konzistence

- chování systému při současném přístupu

- **Unix**: změny okamžitě viditelné
- **Windows**: immutable-shared-file - pokud je soubor sdílený, nejde jej měnit (jednoduchá implementace)

Práva přístupu

- ACL(access control list) //seznam uživatelů s právami k přístupu k souboru
  - seznam uživatelů a jejich přístupových práv (3 bity pro každý přístup)
  - udržovat seznam může být netriviální (možnost doplnit role)
  - Denied ACL - seznam uživatelů, kteří nemají práva s daným souborem nakládat; z důvodu netriviálnosti ACL
- Unix
  - „ACL“ - pro vlastníka, skupinu a zbytek světa
  - př.: rwx-rx-r: vlastník může číst, zapisovat, spouštět, skupina číst a spouštět, zbytek světa jen číst

7. Implementace souborových systémů

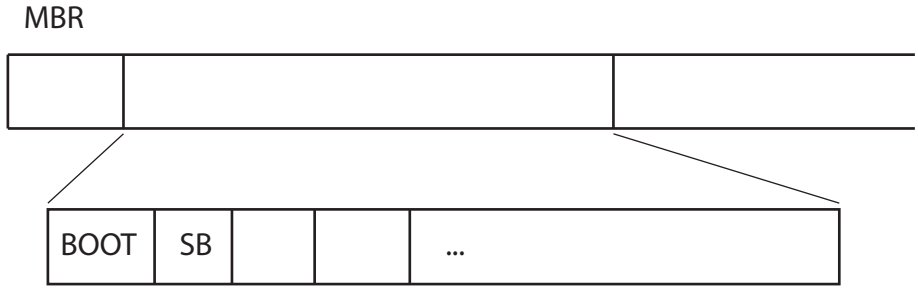
7.1. Očekávané vlastnosti

- (budeme předpokládat souborové systémy pro práci s disky s rotujícími částmi)
- schopnost pracovat se soubory a disky adekvátní velikosti
- efektivní práce s místem (evidence volného místa, nízká fragmentace // nejedná se o rozházení dat na disku, ale fragmentace jako u pamětí)
- rychlý přístup k datům
- eliminace roztroušení dat na disku
- odolnost proti poškození při pádu systému (výpadku napájení) $\Longrightarrow$ rychlé zotavení

Další vlastnosti

- snapshoty // obraz disku v jednom okamžiku ke kterému jsme schopni se vrátit
- komprese dat
- možnost zvětšovat/zmenšovat FS za běhu
- kontrolní součty
- defragmentace za běhu
- správa oprávnění

7.2. Struktura disku

- pro jednoduchost předpokládáme, že struktura disku je lineární
- MBR - master boot record: informace o rozdělení disku na svazky + zavaděč
- 

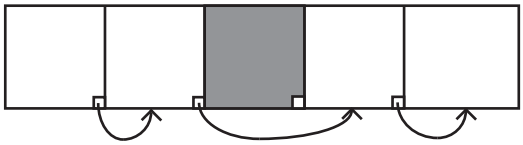
- sektor disku - obvykle velikost 512 B
- $\Longrightarrow$ pracuje se s většími bloky 1 - 32 kB (často 4 kB nebo 8 kB //výhoda, že má stejnou velikost, jako velikost stránky operačního systému) // malé bloky - velké adresy, je výhodnější číst větší blok paměti (vzhledem k seek)
- $\Longrightarrow$ optimální velikost bloku? (rychlost vs. úspora místa)
- jednotlivé svazky obsahují souborový systém (vlastní organizace dat)
- VFS - virtuální souborový systém
- je potřeba si udržovat informace o jednotlivých souborech (FCB, inody)
- cache

7.3. Alokace diskového prostoru

Alokace souvislých bloků

- soubory jsou ukládany na disku za sebe v souvislých blocích
- rychlé sekvenční čtení, problém s odstraněnými soubory //způsobuje vnější fragmentaci
- ideální pro CD, protože jsou určené pouze pro čtení

Soubor jako spojový seznam



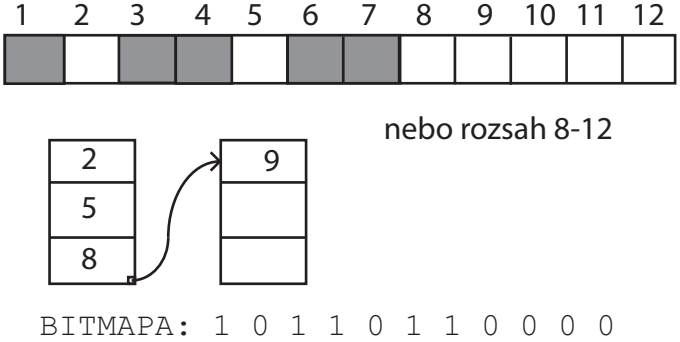
- FS je rozdělený na bloky
- každý blok má ukazatel na následující
- rychlé sekvenční čtení, problématický náhodný přístup (nutnost číst vše //musíme jít po odkazech) + poškození disku
- varianta: uchování odkazů na bloky ve speciální tabulce (FAT)

Indexová alokace

- soubor si nese informaci o svém uložení v blocích (struktura na začátku souboru)
- problém s velkými soubory $\Longrightarrow$ víceúrovňové tabulky

7.4. Evidence volného místa

- je potřeba udržovat informace o volném místě
- použitím spojového seznamu volných bloků (možné použít volné bloky), jako u souborů
- vylepšení $\Longrightarrow$ rozsahy volných bloků (problém při fragmentaci)



- bitmapy - každý bit udává, jestli je daný blok volný (nutné místo - obvykle méně než promile kapacity) //pomocí binární aritmetických operací lze rychle najít např.: první volný blok

Přidělování volného místa

- je žádoucí zapisovat data do souvislých volných bloků $\Longrightarrow$ eliminace přesunů hlavičky

- heuristické algoritmy (složitější na testování), problematické zaplnění disku nad 95 % //problém najít volné místo a současné zapisování více velkých souborů //zapisuje napřeskáčku data souborů za sebe při kopírování dvou souborů
- algoritmus v ext2: //hledání volného místa
  - zvolí se cíl zápisu (první volný blok, případně první volný blok za souborem)
  - hledá se v bitmapě prvních volných 8 bitů (8 bloků), ležících za cílem; pokud takové nejsou, hledá se po bitech
  - (prealokace) alokátor se snaží zabrat 8 bloků za sebou
- clusterování - souvislý zápis více bloků (možnost je přeskládat, aby vyhovovali sekvenčnímu čtení), FreeBSD

Adresáře

- organizace adresářů $\Longrightarrow$ vliv na výkon
- různé struktury
  - spojový seznam - jednoduchá iplementace, větší složitost //lineární časová složitost
  - hash tabulka - komplikace s implementací
  - varianty B-stromů - časté u moderních FS //umožňují velké adresáře a vyhledávat s logaritmickou časovou složitostí
- umístění informací o souborech
  - součást adresáře
  - součást souboru (UNIX) $\Longrightarrow$ problém: listování adresáře, možnost mít soubor ve více nebo žádném adresáři

7.5. Cache a selhání systému

- kvůli rychlejšímu přístupu nejsou často data zapisována přímo na disk $\Longrightarrow$ nejdříve do cache
- při výpadku (pád systému, výpadek napájení) nemusí být data ve write-back cache zapsána $\Longrightarrow$ poškození FS
- potřeba opravit FS (`fsck, chkdisk`) $\Longrightarrow$ časově náročné
- případné narušení systému
  - jeden uživatel zapíše data na disk a smaže je
  - druhý uživatel vytvoří velký soubor a po zapsání medadat (nezapsání dat) vyvolá výpadek
  - po restartu čte data prvního uživatele

Řešení

- synchronní zápis $\Longrightarrow$ zpomalení, konzistence nemusí být zaručena
- soft update - uspořádání zápisů podle určitých pravidel (BSD) //složité
- žurnálování

7.6. Žurnálování

- data se zapisují v transakcích (přsun FS z jednoho konzistentního stavu do druhého)
- nejdříve se transakce zapíše do žurnálu (logu)
- po zapsání do žurnálu je záznam označen speciální značkou a data se můžou zapsat na disk
- po zapsání na disk je zápis z žurnálu odstraněn
- při připojení FS se kontroluje stavu žurnálu
  - zápis záznamu do žurnálu nebyl dokončen (transakce se neprovede)
  - případně, transakce se provede podle informací ze žurnálu
- často se žurnálují jen metada //v případu výpadku můžeme ztratit data, ale není narušena bezpečnost a není potřeba obnovovat strukturu FS
- žurnál je cyklický; při zaplnění se zapíší/uvolní ty na začátku
- pro transakce je potřeba atomických zápisů na disk
- cache & buffery komplikují implementaci atomických zápisů na disk

7.7. FAT

- souborový systém pro MS-DOS (přežil se až do Windows ME)
- jednoduchý design
- soubory se jmény ve tvaru 8+3, nepodporuje oprávnění
- nemá metody proti poškození dat
- disk rozdělený na bloky (clustery)
- soubory popsány pomocí File Allocation Table (FAT) - spojový seznam
- disk (oddíl) rozdělen na 4 části:
  - bootsector (rezervovaná oblast) + informace o svazku
  - 2 × FAT
  - kořenový adresář
  - data
- adresáře jako soubory, kořenový svazek je vytvořen hned na začátku
- původní FAT nepodporoval adresáře

7.8. FAT: varianty

- FAT12, 16, 32: podle velikosti clusteru (max. kapacity - 32 MB, 2 GB, 8 TB)
- další omezení na velikosti souboru

Virtual FAT

- s Windows 95
- podpora dlouhých jmen (LFN)
- až 256 znaků
- soubor má dvě jména - dlouhé a ve tvaru 8+3
- dlouhá jména uložena jako další záznamy v adresáři //další záznamy mají nesmyslná metadata a obsahují uložený celý název

exFAT

- určen pro flash paměti
- podpora větších disků (512 TB, 64 ZB)
- podpora v novějších Windows (původně Windows CE 6)
- zatížen patenty //kvůli tomu se prakticky nerozšířil mimo Windows zařízení

7.9. UFS: Unix File System

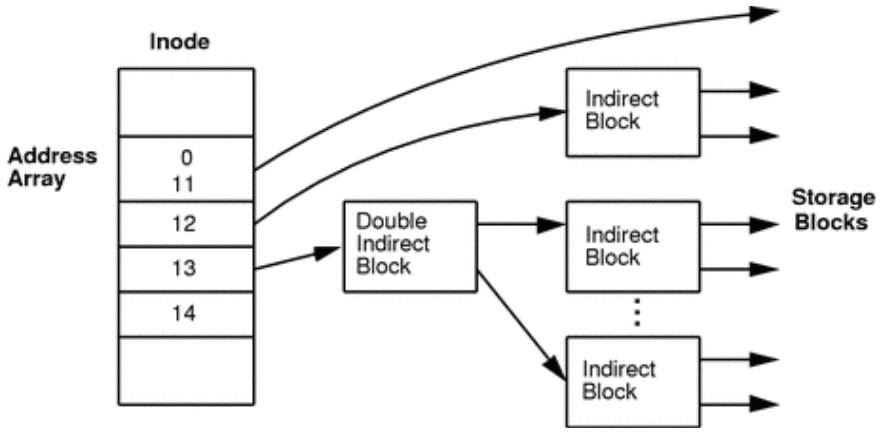
- v různých variantách přítomných v unixových OS - BSD, Solaris, System V, Linux (ext[2,3,4])
- disk se skládá:
  - bootblock - místo pro zavaděč OS
  - superblock - informace o souborovém systému
  - místo pro inody
  - místo pro data
- obrázek je v druhem PC - DODAT !!!

Inoda

- struktura popisující soubor
- informace o souboru
  - typ souboru, vlastníka (UID, GID), oprávnění (rwx)

  - časy (vytvoření, přístup)
  - počet ukazatelů, počet otevřených popisovačů
- informace o uložení dat
  - patnáct ukazatelů na bloky na disku
  - bloky 0-11 ukazují na bloky 1. úrovně
  - blok 12 - nepřímý blok 1. úrovně //1. úroveň - obsahuje odkazy na bloky dat
  - blok 13 - nepřímý blok 2. úrovně //obsahuje odkaz na odkazy bloku, které jsou uloženy v části data a obsahují odkazy na bloky s daty
  - blok 14 - nepřímý blok 3. úrovně //obsahuje odkazy na odkazy odkazu bloku, které jsou uloženy v data
- obrázek
- struktura inody umožňuje mít řídké soubory
- adresář je soubor obsahující sekvenci dvojic (jméno souboru, číslo inody)
- k evidenci volného místa a inod se používají bitmapy
- svazek může být rozdělený na několik tzv. skupin - každá mající vlastní inody, bitmapy, atd. + kopii superblocku $\Longrightarrow$ sloučení souvisejících dat $\Longrightarrow$ eliminace přesunů hlavičky
- velikost bloku $\Longrightarrow$ rychlejší přístup k větším souborům vs. nevyužité místo
- možnost rozdělit blok na několik fragmentů
- konkrétní detaily se mohou lišit
- např.: FreeBSD přidává možnost dělat snapshoty

7.10. Inode



7.11. FS v Linuxu

- Linux nemá jeden hlavní FS
- nejčastěji se používá: Ext2/3/4
- název souboru může mít až 256 znaků (s vyjimkou znaků \a a \0)
- vychází z UFS

- ext2: maximální velikost souboru 16 GB - 2 TB, disku: 2 TB - 16 TB
- ext3: přidává žurnál (3. úrovně - journal, ordered, unordered), binárně kompatibilní s ext2
- ext4: přidává vylepšení:
  - maximální velikost souboru 16 GB - 2 TB, disku 1 EB
  - podporu extentů (místo mapování bloků je možné alokovat blok až do velikosti 128 MB)
  - optimalizace alokací
  - lepší práce s časovačem
- další FS: ReiserFS, BtrFS, JSF, XFS ...

7.12. NTFS

- hlavní souborový systém Windows NT
- kořeny v OS/2 a jeho HPFS (vyvíjen od roku 1993)
- velikost clusteru podle velikosti svazku (512 B - 4 KB) ⇐ max. velikost disku 256 TB, max. velikost souboru 16 TB
- oproti FAT (souborovému systému W9x) ochrana před poškozením + práva
- žurnálování a transakce
- podpora více streamů v jednom souboru
- dlouhé názvy (255 znaků) + unicode
- podpora standartu POSIX; hardlinky, symlinky
- komprese a řídké soubory

Adresáře

- opět technicky soubory
- jména v B+ stromech
- částe metadat je součástí adresáře, část metadat je součástí souborů

7.12.1. Struktura disku

- na začátku disku: boot sector
- 12 % MFT (Maste File Table), 88 % data souborů
- MFT je soubor popisující všechny soubory na FS (MFT je taky soubor)
- MFT se skládá ze záznamů o velikosti 1 KB
- každý soubor je popsán tímto záznamem
- 32 prvních souborů má speciální určení ($MFT, $MFTMirr, $LogFile, $Volume, $Bitmap, $Boot, $BadClus,...)
- informace o souborech včetně jména, časů, atd. uloženy jako záznam v MFT jako dvojice atribut-hodnota

- tělo souboru je taky atribut ⇐ uniformní přístup, možnost uložit malé soubory přímo do MFT
- alternativní proudy ⇐ opět atributy
- v případě potřeby může jeden soubor zabrat více záznamů v MFT
- případně lze použít místo mimo MFT (rezidentní a nerezidentní atributy)

obrázek

- data v souboru jsou popsána pomocí (atributu) tabulky mapující VCN (virtual cluster number) na LCN (logical cluster number)
- VCN - číslo clusteru v souboru (indexováno od nuly)
- LCN - číslo clusteru ve svazku
- každý záznam v tabulce je ve tvaru: VCN, LCN, počet clusterů, např.: //nejsou za sebou ⇐ vzniknou nam ridke soubory, 33 min

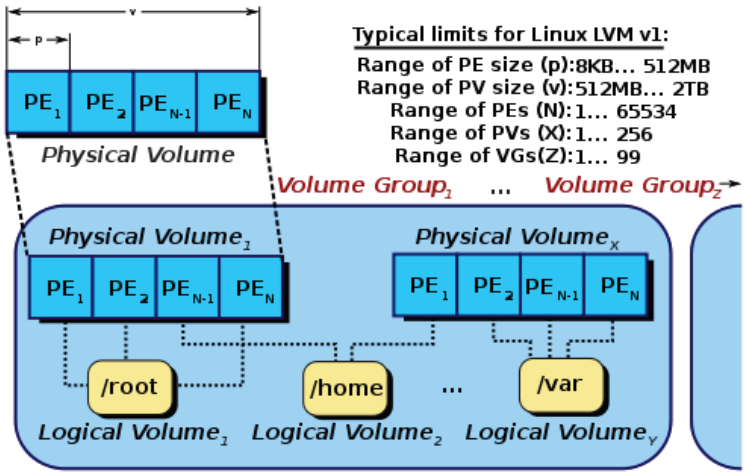
VCN	LCN	počet
0	42	4
4	123	8
32	456	15

Komprese

- řídké soubory
- možnost transparentně komprimovat obsah (vždy po 16 clusterech) ⇐ bloky dat zarovnány na 16 clusterů, pokud zabírá míň místa je komprimován
- čtení i zápis provádí (de)kompresi (LZ77) ⇐ dopad na výkon

7.13. LVM: Logical Volume Management

- problém: svazky mají pevnou velikost (rozdělení disku je pevně dané)
- řešení: logical volume management - vrstva mezi blokovými zařízením a FS
- fyzické disky (PV: physical volumes) rozdělen na rozsahy (PE: physical extends)
- jednotlivé PE poskytnuty do společné Volume Group
- odtud jsou pak přidělovány jednotlivým logickým svazkům ⇐ možnost dynamicky měnit velikost svazku ⇐ nutná podpora FS
- možnost emulovat RAID
- možnost vložit vrstvu, která se bude starat o snapshoty/klony (CoW)
- možnost transparentně provádět kódování
- ve Windows implementace podobná: Logical Disk Manager & Volume Snapshot Service (umožňující SW RAID), spolupráce s FS
- někdy dodáván jako software třetích stran

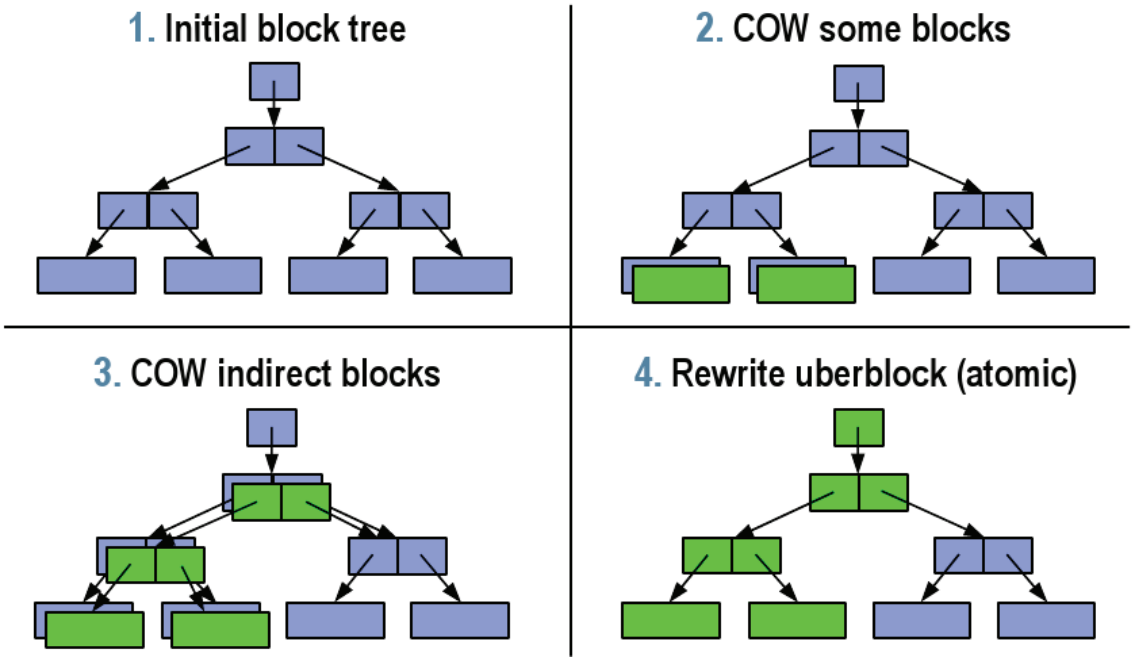


7.14. ZFS

- moderní souborový systém (r. 2005), SUn (Oracle)
- podpora (open)Solaris, FreeBSD, NetBSD, Mac OS X, Linux (licenční problémy)
- kombinuje prvky LVM, RAID
- interně 128 bitová adresace (max. kapacita 256 ZB, ostatní limity kolem 16 EB)
- disky jsou spojeny do *poolu*, FS dělá automatický stripping ⇐ rozprostře se přes všechny disky
- bloky dat různých velikostí
- little- a big-endian (podle aktuání situace)
- ditto blocks(zdvojené zápisy) //kvůli bezpečnosti
- deduplikace //pokud jsou stejné bloky dat ve dvou souborech, použije se pouze jeden
- podpora komprese

7.15. Konzistence

- RAID-Z: podobný RAID-5, ale má různě velké bloky (odpovídají logickým blokům) ⇐ např.: 3 bloky dat + 1 paritní, atd.
- u dat jsou evidovány kontrolní součty ⇐ ochrana proti tichému poškození (chyba HW i SW)
- konzistence založena na metodě Copy-on-Write
- používaná data nikdy nejsou přepsána ⇐ nejdřív jsou zapsána data a pak jsou (automaticky) změněna metadata
- ⇐ výdhoné slučovat operace do transakcí
- ⇐ FS je vždy v konzistentním stavu
- ⇐ infrastruktura pro vytváření snapshotů/klonů souborového systému



1. alokujeme bloky pro data
2. pro kazda data vytvorime novy blok
3. vytvorime metadata, ktere predstavuji bloky dat
4. po zapsani provedeme prohozeni těch částí, které jsou měněné

7.16. ISO-9660

- souborový systém pro CD-ROM, podpora všech OS
- zápis jen jednou, sekvenční čtení ⇐ není potřeba dělat kompromisy
- logický sektor 2048 B (může být i větší)
- na disku může být víc logických svazků, svazek může být na více discích
- na začátku 16 rezervovaných bloků + 1 blok (Primary volume Descriptor) ⇐ informace o disku, odkaz na kořenový adresář
- adresář popsán pomocí záznamů proměnlivé délky
  - textová data v ASCII
  - binární 2× (little- i big-endian)
- možnosti formátu určeny úrovněmi a rozšířeními
- **Level 1** - soubory 8.3, všechny soubory spojité, 8 úrovní adresářů
- **Level 2** - jména až 31 znaků
- **Level 3** - nespojité soubory (jednotlivé souvislé bloky se mohou opakovat a prokládat)

7.17. Rozšíření

Rock Ridge

- kompatibilita s unixy
- přidává dlouhá jména
- neomezené zanoření adresářů
- unixová oprávnění
- podpora symbolických odkazů, možnost mít na disku soubory zařízení

Joliet

- kompatibilita s Windows
- přidává dlouhá jména + podporu Unicode
- neomezené zanoření adresářů

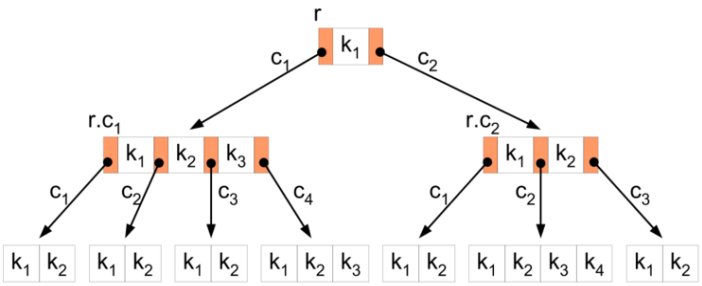
7.18. UDF: Universal Disk Format

- náhrada za ISO-9660
- používám převážně pro DVD a Blue-ray disky
- dlouhé názvy souborů, kapacita až 1 EB
- různé varianty formátu
  - **Plain build** - základní formát (data lze přepisovat, pokud médium podporuje, přepis konkrétních sektorů - DVD-RAM, DVD+RW)
  - **Virtual Allocation Table** - inkrementální zápisy (CD-R)
  - **Spare build** - pokud to médium podporuje, lze data přepisovat, zahrnuta obrana proti opotřebení sektorů

8. Bezpečnost

8.1. B-stromy

- vyvážené stromy
- všechny listy ve stejné hloubce
- každý uzel (mimo kořenového) obsahuje nejméně t-1 klíčů, tj. má t potomků, v neprázdném stromě kořen obsahuje alespoň jeden klíč
- každý uzel má nanejvýše 2t-1 klíčů $\Leftarrow$ 2t potomků $\Leftarrow$ plný uzel
- **Věta:** pokud je počet klíčů $n \geq 1$, pak pro B-strom stupně $t \geq 2$ a výšky h platí: $h \leq log_t n + 1/2$



- ne všechna data v paměti (vs. běžné binární stromy)
- rozdílné přístupové dobry primární paměti a sekundární (desítky až stovky nanosekund vs. milisekundy-vystavení hlavičky)
- preferované sekvenční čtení $\Longrightarrow$ načtení celých stránek
- zobecnění 2,3-stromů, 2,3,4-stromů
- rozlišujeme složitost operací (porovnání, atd.) a I/O operací (zápisy, čtení)
- složitost vyhledávání, vložení: $O(th) = O(tlog_t n)$
- počet přístupů na disk: $O(h) = O(log_t n)$

8.2. Souborové systémy využívající B-stromy

XFS

- navržen SGI pro operační systém IRIX (k dispozici v Linuxech)
- spoléhá na B+ stromy (nutné zaplnění ze 2/3)
- rozdělení disku na agregační jednotky
- evidence volného místa v B-stromech (dva stromy $\Rightarrow$ vyhledávání podle pozice, velikosti)
- uložení souborů $\Rightarrow$ extenty jako v případě NTFS (uloženo v inodách)
- u větších souborů použití B-stormů $\Rightarrow$ zřetězení listů
- malé adresáře v inodách; větší $\Rightarrow$ B-stromy

JFS

- navržen IBM pro AIX
- koncepce žurnálování blízká databázovým systémům
- v některých ohledech podobný přístup jako u XFS

obrázek

8.3. BtrFS

- všechna data uložena v B-stromech
- varianta podporující CoW (řeší integritu) //copy-on-write
- jedna implementace (jednoduchá implementace, kontrolní součty, atd.)
- všechny klíče ve tvaru:

```
struct btrfs_disk_key {
    __le64 objectid;
    u8 type;
    __le64 offset;
}
```

- slučování souvisejících dat vedle sebe
- malé soubory ve stromu
- velké soubory (vlastní extenty), popsané v klíči (využití offsetu)
- automatická defragmentace
- několik speciálních stromů (volné místo)

obrázek btrfs

8.4. Bezpečnostní problémy

Bezpečnostní požadavky

- diskrétnost (confidentiality) - informace o systému jsou k dispozici jenom těm, co k tomu mají nárok
- neporušenost (integrity) - části systému můžou měnit jenom ti, co k tomu mají práva
- dostupnost (availability)
- autentičnost (authenticity) - systém je schopen ověřit identitu uživatele

Typy útoků

- pasivní (pouze čtení) × aktivní //získání hesel
- cílený útok × viry

Důvody

- „běžné sdílení“ (netechničtí uživatelé, insideři)
- špionáž
- snaha získat peníze nebo alespoň výpočetní výkon
- pro zábavu

8.5. Problematika bezpečnosti a její (celo)společenský dopad

- hackers //má zájem zkoumat, jak věci fungují a použít je nebežným způsobem × crackers //pouze se chce se dostat do systému
- white hat //naboura se do systému za účelem vylepšení sytému × black hat //snaha získání něčeho ve svůj prospěch
- známost exploitu ⇒ fatální důsledky //začnou ho využívat i méně zkušení lidé
- veřejná známost exploitu ⇒ ještě horší důsledky
- ⇒ bezpečnostní problémy vyžadují specifický přístup
  - hlašení chyb diskrétním způsobem
  - všechno má své meze

8.6. Typy útoků

- Trojský kůň (problém i unixu, nechráněný adresář \tmp a $PATH) //povoleni spouštění souborů z aktuální složky a nejen z `/usr/bin`
- login spoofing
- logic bomb (dead man's switch) //programátor nebo inseder nastavi podmínku, že pokud něco, tak se stane něco špatného se systémem
- backdoors //přístup např.: programátor si nechá neoficiální přístup
- buffer overflow //vychází z toho, že systém nekontroluje vstupy
- viry, rootkity
  - navázání na systémové volání
  - metody skrývání
- denail of servise (DoS)
- selhání lidského faktoru (Kevin Mitnick) //vydávání se za někoho jiného a přesvědčení člověka aby vydal nějaké informace

8.7. DoS (Denail of Service)

- zahlcení služby, buď:
  - vysokou spotřebou systémových prostředků (procesorový čas, paměť, přenosové pásmo) `while (1) fork();` //jeden proces se stále dokola dělí na další procesy
  - změna konfigurace (např.: směrovací tabulky, propagace špatné konfigurace, Supronet 2009)
  - narušení komunikace zabraňující efektivní komunikaci ostatních (Slowloris aka outloň váhavý) //držíme otevřené spojení a jednou za několik sekund pošleme požadavek - server neuzavře spojení kvůli timeout
  - varianta DDos (botnety)
  - neúmyslný DoS útok (Slashdot effect)

8.8. Backdoor

- mechanizmus na obejití běžného přihlašovacího postupu

```
while(true) {
    login = getLogin();
    passw = getPassword();
    if (isValid(login, passw)) break;
}
```

s backdoor:

```
while(true) {
    login = getLogin();
    passw = getPassword();
    if ((isValid(login, passw)) || (login.equals("ja"))) break;
}
```

- bez znalosti kódu špatně odhalitelné
- ⇒ Interbase (u: 'politicaly' p: 'correct') 1994-2001! //v DB systemu
- kód na slajdech //pokud má options nastavené WCOLNE a WALL tak se současnému procesu přiřadí ID uživatele 0
- Linux Kernel `sys_wait4`

8.9. Buffer overflow

//vždy testovat, zdali data nejsou větší než buffer

```
void foo(char * str) {
    char buf[1024];
    strcpy(buf, str);
}
```

- jak vypadá zásobník?
- přepis návratové adresy (+ NOP sled)
- problém s \0

```
B8      0100000000  MOV EAX, 1

33C0                XOR EAX, EAX
40                  INC EAX
```

8.10. Format string attack

//vypisovat řetězec od uživatele přes %s a ne přímo

```
int main(int argc, char ** argv) {
    printf(argv[1]);
}
```

- `./foo "%x %x %x"` ⇒ `čteme obsah zásobníku`
- jak zapisovat?

- `%n`
  - do adresy argumentu zapíše počet již vygenerovaných znaků
  - printf("aaa %i – %n", i, &j)
- možnost přepisovat data
- `snprintf(char *str, size_t size, const char *format, ...);`
- máme buffer, máme možnost využít buffer k ukládání adres pro %n

8.11. Directory traversal attack

```
Hlavicka
<?
    $f = fopen("texty/{$_REQUEST["file"]}, "r");
    while (!feof($f))
        echo fgets($f);
    fclose($f);
?>
Paticka
```

- index.php?file=text1.htm $\Longrightarrow$ funguje dobře
- index.php?file=../../../../etc/passwd
- index.php?file=../db-connect.php

8.12. SQL injection attack

```
<?
    $login = $_REQUEST["login"];
    $passw = sha1($_REQUEST["passwd"]);
    $q = "select * from users where (user = '$login' and pasw='$pasw')";
    $loggedIn = pg_num_rows(pg_query($q));
?>
```

- pokud je login foo a heslo bar $\Longrightarrow$ funguje dobře
- pokud je login foo' or true) – a heslo bar $\Longrightarrow$ máme problém
- $\Longrightarrow$ `select * from users where (user = 'foo' or true) -- pass = '...')`
- přes funkce DB se lze dostat k dalším citlivým informacím
- řešení $\Longrightarrow$ pokud to API umožňuje parametrizované dotazy (JDBC, ADO) nebo důsledná kontrola vstupů

8.13. Cross-site scripting(XSS)

- předáme druhému uživateli námi vytvořený JS kód tak, aby to nepoznal
- např.: na fórum pošleme zprávu obsahující `<script>...</script>` lze použít i atributy `onclick=`, atd.
- daný kód se provede, když uživatel stránku načte $\Longrightarrow$ přístup k citlivým informacím
- máme k dispozici celý DOM a XMLHttpRequest!

- vyžaduje určitou dávku soc. inženýrství
- důsledná kontrola všech vstupů na HTML tagy (nebo konverze <a >na entity)

Cross-site request forgery (CSRF)

```
<img src="http://bank.example.com/withdraw?account=bob&amount=1000000&for=Mal">
```

9. Viry

- různé kategorie
  - boot viry //utok založený na tom, že virus se zapise na boot misto a prvni se spusti virus a pote az OS
  - makro viry
  - companion viry //dva spustitelné programy s ruznými příponami (např.: iexplorer.exe a iexplorer.com a při zadání iexplorer se sputí první iexplorer.com
  - parasitistické viry
- různé účely
- poškození spustitelného programu //image11_2
  - zápis dodatečného kódu (začátek/konec)
  - přemostění jeho spuštění (do přidané části)
  - spuštění viru
  - spuštění původního kódu
- snaha maskovat viry (šiféry)
- snaha měnit vir, snaha zabránit analýze
- komprese dat

9.1. Albánský virus

Hi,
This is an Albanian virus. As you know we are not so technical advanced as in the West. We therefore ask you to delete all your file on your harddiks manually and send this email to all your friedns.
Thanks for helping us, The Albanian Hackers

9.2. Změny kódu

- jmp-makra //ma za cíl zmást toho, kdo bude hledat nějaký vir

```
01:  push ebp              1.:  push 05
02:  mov eax, 123          2.:  ret
                      ⇒    6.:  jmp 07
                           3.:  push ebp
                           4.:  jmp 04
```

- polymorfní viry ⇒ změna kódu programu //přidají např. do asemblerovského kódu operace NOP

9.3. Debugger

- přerušení (jednobytová operace INT3) - EXCEPTION_BREAKPOINT
- debugger přebírá obsluhu vyjímek
- v místě breakpointu je přepsána instrukce na INT3
- vyvolá se vyjímka ⇒ předání řízení debuggeru (pozastavení programu)

- instrukce je vrácena zpět a je provedena znovu

Odhalení debuggeru

- hledání instru INT3 v kódu
- záměrné vyvolání vyjímky a sledování,jesli byla obsloužena programem
- sledování prodle mezi operacemi

9.4. Matice přístupů & TCB

- oddělení politiky od mechanismů
- reference monitor - kontrola přístupu
- doména ochrany (protection domain): množina dvojic, objekt - oprávnění (⇒ matice)
- každý proces běží v nějaké doméně
- přechod mezi doménomi lze specifikovat pomocí domény ochrany
- „schopnosti“ (capabilities) ⇒ řádky

Trusted Computing Base

- TCB
  - část systému, která splňuje bezpečnostní kritéria
  - stará se o dodržování a vynucování práv
  - mj. správa procesů, paměti, I/O

Object manager

- soustřeďuje práci s objekty do jednoho místa
- jednotný způsob práce s objekty
- možnost řídit přítstup k objektům
- kontrola spotřebovaných zdrojů
- globální pojmenování objektů
- typy objektů
  - jaderný (kernel) obejkt (základní struktury)
  - executive objekt (objekt v executive části jádra, může obsahovat jaderné objekty, RuSo)
  - GDI/User objekt (objekty graf. modulu, odlišný přístup
- object manager spravuje executive objekty
- k jednotlivým objektům se z uživatelské prostředí přistupuje přes handly
- jádro může přes ukazatele
- uvolnění objektu na základě počtu referencí
- executive objetky mají sdílenou hlavičku
  - jméno objektu

  - adresář objektu
  - práva přístupu
  - počet otevřených handlů a ukazatelů
  - využití přidělených kvót
  - seznam procesů mající otevřený handle na objekt
  - ukazatel na typ objektu
    * typické oprávnění
    * příznaky pro synchronizaci
    * metody objektu (open, close, delete, ...)
    * a další
- handle je vlastně ukazatel do tabulky nandlů (každý proces má vlastní)
- záznamy v tabulce: pointer na objekt + příznaky (použití, dědičnost) + oprávnění
- trojúrovňová tabulka
- objekty ve vlastním jmeném prostoru

9.5. Kategorie bezpečnosti podle Orange Book

- DoD definuje několik úrovní zabezpečení
- D - žádné požadavky
- C - systém spolupracujících uživatelů
  - chráněný režim; možnost ovládat oprávnění
  - dokumentace; testování
  - úrovně C1, C2
  - v úrovni C2 nestsačí kontrola na úrovni unixových rwx
- B - zprísněná pravidla
  - popsaný bezpečtnostní model (neformálně nebo formálně)
  - požadavky na architekturu (omezení složitosti ⇒ možnsto testovat/ověřit)
  - posílené bezpečnostní prvky (např.: přihlašování)
  - detekce napadení, atd.
  - tři úrovně B1, B2, B3
- A
  - A1 - formání model ochrany, včetně důkazů správnosti (XTS-400)

9.6. Víceúrovňová bezpečnost

Bell-La Padula Model

- proces s oprávněním na úrovni k může číst pouze na své úrovni a nižší
- proces může zapisovat pouze do objektů na své úrovni a vyšší
- ⇒ armáda
- zajištěno utajení, ne integrita dat

Biba model

- proces s oprávněním na úrovni k může číst pouze na své úrovni a vyšší
- procesmmůže zapisovat pouze do objektů na své úrovni a nižší

9.7. Singularity

- OS z Microsoft Research ⇒ nové koncepty
- softwarově izolované procesy (SIP); mohou se skládat z vláken
- izolace na bázi softwaru (ostatní OS využívají HW)
- managed prostředí založené na MSIL, každý SIP má vlastní běhové prostředí, GC
- procesy spolu mohou komunikovat pouze zasíláním zpráv
- ⇒ „kanály“ s jednoznačně definovaným rozhraním ⇒ jazyk Sign#
- jde automaticky ověřit, že komunikace neskončí v nežádoucím stavu
- jde (je nutné) ověřit, že program neovlivňuje okolí (spolupráce překladače)
- mikrojádro převážně implementované v Sign#
- přepnutí kontext/systémového volání rychlé (není potřeba měnit nastavení stránek, mazat TLB)
- základ pro projekt Midori (nástupce Widnwos?)

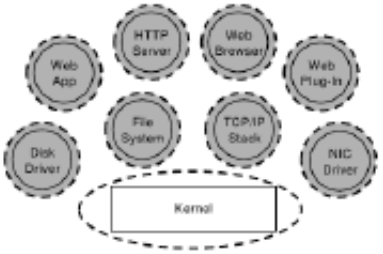


Figure 4a. Micro-kernel configuration (like MINIX 3). Dotted lines mark protection domains; dark domains are user-level, light are kernel-level.

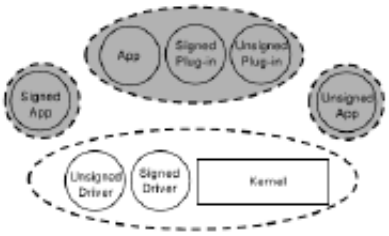


Figure 4b. Monolithic kernel and monolithic application configuration.

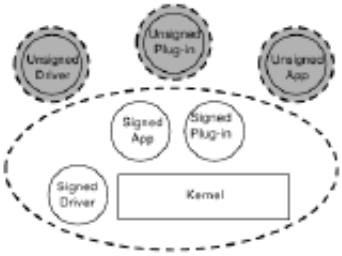


Figure 4c. Configuration with distinct policies for signed and unsigned code.