KIV/ZOS 2003/2004
Přednáška 10

Algoritmus Not-Recently-Used (NRU, NUR)
.....

* OS se snaží zjistit, které stránky se používají a nepoužívané vyhazovat
* systémy s VM poskytují HW podporu - stavové bity Referenced (R) a Dirty
  (zde M jako Modified) v tabulce stránek
* bity nastavované HW podle způsobu přístupu ke stránce
 - bit R - nastaven na 1 při čtení nebo zápisu do stránky
 - bit M - nastaven na 1 při zápisu do stránky; označuje, že
           se stránka má při vyhození zapsat na disk
 - po nastavení bitu zůstane na 1 dokud ho SW nenastaví zpět na 0

* algoritmus NRU:
 - na začátku mají všechny stránky nastaveny R=0, M=0
 - bit R je OS nastavován periodicky na 0 (např. při přerušení
   časovače) - tím se rozliší, které stránky byly referencovány
   v poslední době
 - OS rozlišuje 4 kategorie stránek:

   třída 0: R=0, M=0
   třída 1: R=0, M=1 ;; vznikne z třídy 3 po tiku, který nastaví R=0
   třída 2: R=1, M=0
   třída 3: R=1, M=1

 - algoritmus NRU vyhodí stránku z nejnižší neprázdné třídy, výběr mezi
   stránkami ve stejné třídě je náhodný

* algoritmus předpokládá, že je lepší vyhodit modifikovanou stránku
  která nebyla použita 1 tik než nemodifikovanou stránku, která se
  právě používá

* výhody algoritmu NRU:
 - jednoduchost, srozumitelnost
 - efektivně implementovatelný

* nevýhody:
 - výkonnost (jsou i lepší algoritmy)

Pokud by HW neměl bity R a M, můžeme je simulovat následujícím způsobem:

* při startu procesu se všechny jeho stránky označí jako nepřítomné
  v paměti
* při odkazu na stránku nastane výpadek stránky - OS interně nastaví
  R=1 a nastaví mapování v režimu READ ONLY
* při pokusu o zápis do stránky nastane výjimka - OS výjimku zachytí, nastaví
  M=1 a změní režim přístupu do stránky na READ/WRITE.

Algoritmy ”Second Chance” a ”Clock”
.....

* algoritmy ”Second Chance” a ”Clock” vycházejí z algoritmu FIFO

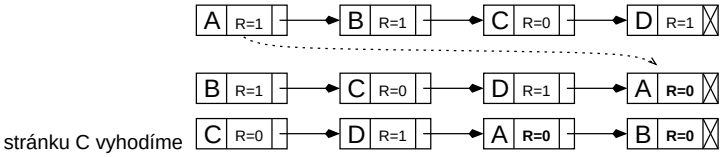
Obchod: V algoritmu FIFO jsme vyhazovali zboží, které bylo zavedeno před
nejdelší dobou (bez ohledu na to, jestli ho někdo chce nebo ne). V algoritmu
”Second Chance” začneme evidovat, jestli zboží někdo v poslední době koupil
(pokud ano, prohlásíme ho za čerstvě zavedené zboží).

* jak modifikovat FIFO, abychom zabránili vyhození často používané stránky?
* algoritmus (Second Chance - vyhledání stránky pro vyhození):
 - podívat se na bit R nejstarší stránky
 - pokud R=0, stránka je nejstarší a zároveň nepoužívaná => vyhodíme
 - pokud R=1, nastavíme R na 0 a přesuneme na konec seznamu stránek
   (jako by byla nově zavedena)

Příklad:

Stránky uchováváme v seznamu uspořádaném podle času příchodu. V paměti budeme mít např. stránky A, B, C, D (viz obrázek); algoritmus ”Second Chance” bude probíhat v následujících krocích:

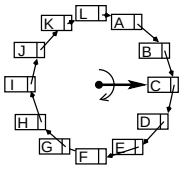
1. krok: Nejstarší je A; má R=1 => nastavíme R na 0 a přesuneme na konec seznamu;
2. krok: Druhá nejstarší je B; má také R=1 => nastavíme R na 0 a opět přesuneme na konec seznamu;
3. krok: Další nejstarší je C, R=0 => vyhodíme jí.



[]

* algoritmus ”Second Chance” vyhledává nejstarší stránku, která nebyla referencována ”v poslední době”
* pokud byly všechny referencovány, degeneruje na čisté FIFO:
 - postupně všem stránkám nastavíme bit R na 0 a přesuneme je na konec seznamu
 - dostaneme se opět na stránku A, tentokrát má R=0 => vyhodíme jí
 => algoritmus končí nejvýše po $počet_rámců + 1$ krocích

* algoritmus ”Clock” = optimalizace datových struktur algoritmu Second Chance:
 - stránky udržovány v kruhovém seznamu
 - ukazatel na nejstarší stránku (”ručička hodin”)

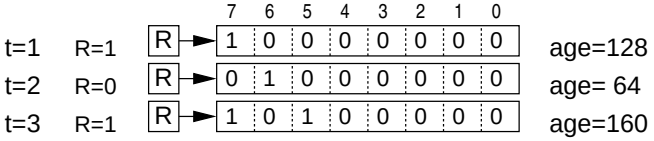


* výpadek stránky -> vyhledáváme stránku k vyhození
 - stránka kam ukazuje ”ručička”:
  . má-li R=0, stránku vyhodíme a ručičku posuneme o 1 pozici
  . má-li R=1, nastavíme R na 0, ručičku posuneme o 1 pozici; opakujeme dokud nenalezneme stránku s R=0
* od algoritmu Second Chance se liší pouze implementací
* varianty algoritmu Clock používají např. systémy BSD UNIX

Softwarová aproximace LRU
....

* algoritmus LRU vždy vyhazuje nejdéle nepoužitou stránku

* algoritmus Aging:
 - každá položka v tabulce stránek má pole ”stáří” age, N bitů (např. N=8)
 - na počátku age=0
 - při každém přerušení časovače pro každou stránku:
  . posun pole ”stáří” o 1 bit vpravo
  . zleva se přidá hodnota bitu R
  . nastavení R na 0



* to odpovídá následujícímu kódu (v Turbo Pascalu):

```
    age := age shr 1;           { posun o 1 bit vpravo }
    age := age or (R shl N-1);  { zleva se přidá hodnota bitu R }
    R := 0;                     { nastavení R na 0 }
```

* při výpadku stránky se vyhodí stránka, jejíž pole age je má nejnižší hodnotu

Dva rozdíly od LRU:

* několik stránek může mít stejnou hodnotu pole age a nevíme která stránka byla odkazovaná dříve (u LRU to víme vždy)
 - rozlišení je ”hrubé” (= po ticích časovače)
* pole age se může snížit na 0 - nevíme, zda stránka byla naposledy odkazovaná před 9 nebo před 1000 tiky časovače
 - uchovává pouze omezenou historii
 - v praxi není problém: pokud je tik časovače po 20 ms a N=8, nebyla odkazována 160 ms => nejspíš není tak důležitá, můžeme jí vyhodit
* pokud se musíme rozhodovat mezi dvěma stránkami se stejnou hodnotou age, vybíráme náhodně

Shrnutí algoritmů pro nahrazování stránek
.....

* optimální algoritmus (MIN čili OPT)
 - není implementovatelný, ale je užitečný pro srovnání
* FIFO
 - vyhazuje nejstarší stránku
 - jednoduchý, ale je chopen vyhodit důležité stránky a trpí Beladyho anomálií
* LRU (Least Recently Used)
 - výborný
 - implementace vyžaduje speciální HW, proto prakticky používán zřídka
* NRU (Not Recently Used)
 - rozděluje stránky do 4 kategorií podle bitů R a M
 - efektivita nic moc, přesto občas používán
* ”Second chance” a ”Clock”
 - vycházejí z FIFO, před vyhozením zkontrolují zda se stránka používala
 - mnohem lepší než FIFO
 - používané algoritmy (např. některé varianty UNIXu)
* Aging
 - dobře aproximuje LRU => efektivní
 - často prakticky používaný algoritmus

Ostatní problémy stránkované virtuální paměti

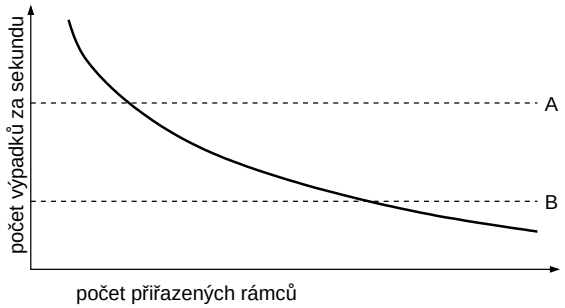
Alokace fyzických rámců
.....

* 2 základní metody - globální a lokální alokace:
 - globální alokace - pro vyhození se uvažují všechny rámce
 - lokální alokace - pro vyhození se uvažují pouze rámce alokované procesem (tj. obsahující stránky procesu, jehož výpadek stránky se obsluhuje)
  . počet stránek alokovaných pro proces se nemění
  . program se vzhledem ke stránkování chová přibližně stejně při každém běhu
 - u globální alokace vybírá ze všech rámců
  . lepší průchodnost systému - proto globální alokace častější
  . na běh procesu má vliv chování ostatních procesů

* při lokální alokaci - kolik rámců dát kterému procesu?
 - nejjednodušší - všem procesům dáme stejně, ale potřeby procesů mohou být různé
 - proporcionální - dáme každému proporcionální díl podle velikosti procesu

- nejlepší metoda - podle frekvence výpadků stránek (Page Fault Frequency, PFF)

Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců:



PFF se snažíme udržet v rozumných mezích:

* pokud je PFF větší než A, přidáme procesu rámce
* pokud je PFF menší než B, proces má asi příliš paměti, rámce mu mohou být odebrány

Zahlcení
.....

* proces pro svůj rozumný běh potřebuje pracovní množinu stránek
* pokud se pracovní množiny stránek aktivních procesů nevejdou do paměti, nastane tzv. zahlcení (angl. trashing)

* jak to vypadá:
 - v procesu nastane výpadek stránky
 - paměť je plná (není volný rámec) => je třeba některou stránku vyhodit
 - stránka pro vyhození bude ale brzy zapotřebí, takže se bude muset vyhodit jiná používaná stránka
 - z uživatelského hlediska se to projeví tak, že systém pracuje intenzivně s diskem a běh procesů řádově zpomalí (stráví víc času stránkováním než během)

* řešení - při zahlcení snížit úroveň multiprogramování (zahlcení lze detekovat pomocí PFF)

Zhodnocení mechanismu virtuální paměti
.....

Virtuální paměť má podstatné výhody oproti předchozím mechanismům:

* rozsah virtuální paměti (např. 2 GB pro proces - NT nebo Linux na i386)
 - adresový prostor úlohy není omezen velikostí fyzické paměti
 - multiprogramování (= počet procesů) není zásadně omezeno rozsahem fyzické paměti
* efektivnější využití fyzické paměti
 - není vnější fragmentace paměti
 - nepoužívané části adresového prostoru úlohy nemusejí být ve fyzické paměti

Nevýhody:

* režie při převodu virtuálních adres na fyzické adresy
* režie procesoru (údržba tabulek stránek a rámců, výběr stránek pro vyhození, plánování I/O)
* režie I/O při čtení/zápisu stránky
* paměťový prostor pro tabulky stránek
* vnitřní fragmentace

Segmentace

* dosud diskutovaná virtuální paměť byla jednorozměrná:
  od adresy 0 do nějaké maximální virtuální adresy
* pro mnoho programů by bylo výhodnější mít víc samostatných virtuálních
  adresových prostorů
* příklad - mám několik tabulek a chci, aby jejich velikost mohla růst

=> paměť nejlépe mnoho nezávislých adresových prostorů = segmenty

* segment = logické seskupení informací
* každý segment lineární posloupnost adres, začínající od adresy 0
* programátor o segmentech ví, používá explicitně (adresuje konkrétní
  segment)
* příklad - překladač Pascalu může používat samostatné segmenty pro:
  - kód přeloženého programu
  - globální proměnné
  - hromadu
  - zásobník návratových adres
  - je možné i jemnější dělení (segment pro každou proceduru/fci)
* často se používá také pro implementaci:
  - přístupu k souborům (1 soubor = 1 segment)
    . není třeba open, read...
  - sdílených knihoven:
    . dnešní programy využívají rozsáhlé knihovny - knihovnu potřebuje prakticky každý
      program
    . myšlenka vložit knihovnu do segmentu a sdílet mezi více programy

* každý segment je logická entita - má smysl, aby měl samostatnou ochranu

Čistá segmentace

* každý odkaz do paměti se skládá z dvojice: (selektor, offset)
 - selektor: číslo segmentu, určuje segment
 - offset: relativní adresa v rámci segmentu
* technické prostředky musí přemapovat dvojici (selektor, offset) na fyzickou
  (= lineární) adresu
* k tomu slouží tabulka segmentů, každá položka tabulky obsahuje:
  - počáteční adresu segmentu (bázi)
  - rozsah segmentu (limit)
  - příznaky ochrany segmentu (nejčastěji čtení, zápis, provádění = rwx)
* postup při převodu na fyzickou adresu:
  - PCB obsahuje odkaz na tabulku segmentů procesu
  - odkaz do paměti má tvar (selektor, offset)
    např. v důsledku instrukce LD R, selektor:offset
  - selektor = index do tabulky segmentů
  - zkontroluje se zda je offset<limit; ne => chyba porušení ochrany paměti
  - zkontroluje se, zda dovolen způsob použití; ne => chyba porušení
    ochrany paměti
  - adresa = báze + offset
* často možnost sdílet segment mezi více procesy

* mnoho věcí podobných jako přidělování paměti po sekcích, ale rozdíl:
  - po sekcích - pro procesy
  - segmenty - pro části procesu
* stejné problémy jako přidělování paměti po sekcích: externí
  fragmentace paměti, mohou zůstávat malé díry (tj. dále již prakticky
  nepoužitelné)

Segmentace na žádost

* segment může být zavedený v paměti nebo odložený na disku
* pokus o adresování segmentu, který není v paměti způsobí výpadek segmentu
* OS zavede segment do paměti
* není-li místo, je některý jiný segment odložen na disk

* HW podpora - v tabulce segmentů bity:

  - bit ”segment je zaveden v paměti” (Present/absent)
  - bit Referenced

* používal např. systém OS/2 pro i80286 - pro výběr segmentu k odložení
  algoritmus Second Chance

Segmentace se stránkováním
.....

* pokud velké segmenty, je nepraktické je udržovat v hlavní paměti celé
  => myšlenka stránkování segmentů, v paměti pouze potřebné stránky
 - implementace - např. každý segment bude mít vlastní tabulku stránek

Příklad pro zajímavost: procesor Intel Pentium

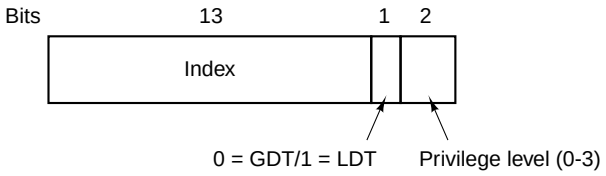
Poznámka: Obrázky v této podkapitole byly převzaty z knihy (A. S. Tanenbaum:
Modern Operating Systems, 2nd ed., Prentice Hall 2001).

* CPU má možnost jak segmentace, tak stránkování, tak segmentace se
  stránkováním

* základem dvě tabulky - LDT (Local Descriptor Table) a GDT (Global
  Descriptor Table)
 - každý proces má vlastní LDT
 - GDT je jedna, sdílená všemi procesy
* LDT popisuje segmenty lokální pro program (kód, data, zásobník, ...)
* GDT popisuje systémové segmenty, včetně OS

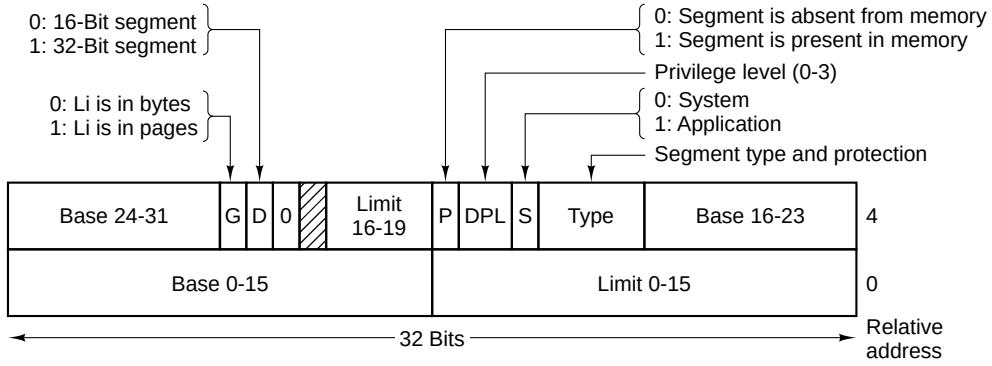
* Pentium má 6 segmentových registrů
 - nejdůležitější:
  . CS - kódový segment (Code Segment)
  . DS - datový segment (Data Segment)
  . SS - zásobníkový segment (Stack Segment)

* před přístupem k segmentu se do některého segmentového registru zavede
  selektor segmentu
 - selektor je 16 bitový a má tvar:
  . 13 bitů - index do tabulky GDT nebo LDT
  . 1 bit: 0 = GDT, 1 = LDT
  . 2 bity: úroveň privilegovanosti (0 - 3)



* 13 bitů pro index => LDT i GDT mohou mít max. 2^13 = 8192 položek
* selektor 0 je ”zakázaný” (jeho použití způsobí výjimku) - používá
  se pro indikaci případu, že segment není dostupný

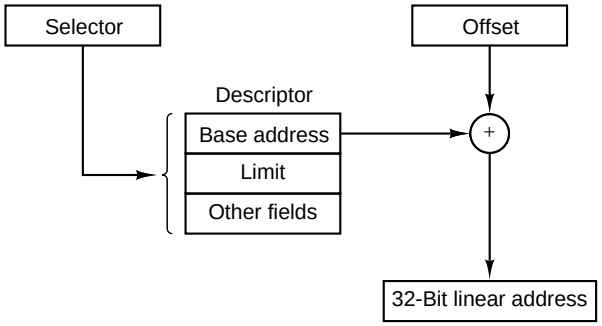
* ve chvíli zavedení selektoru do segmentového registru CPU také zavede
  odpovídající popisovač z LDT nebo GDT do vnitřních registrů CPU
 - podle bitu 2 selektoru se vybere LDT nebo GDT
 - popisovač segmentu je na adresa (selektor and 0fff8h)+(zač. tabulky LDT/GDT)
* popisovač segmentu má délku 64 bitů:
 - 32 bitů báze
 - 20 bitů limit (je buď v bytech (tj. do 1 MB) nebo ve 4K stránkách (do 2^32))
 - příznaky (typ a ochrana segmentu, segment přítomen v paměti...)



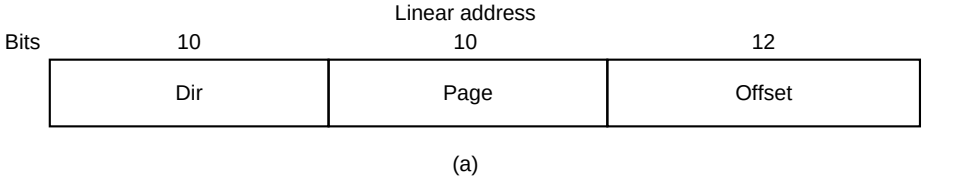
* tj. proces může používat 16K nezávislých segmentů, každý může mít velikost až 2^32 bytů

Konverze na fyzickou adresu:

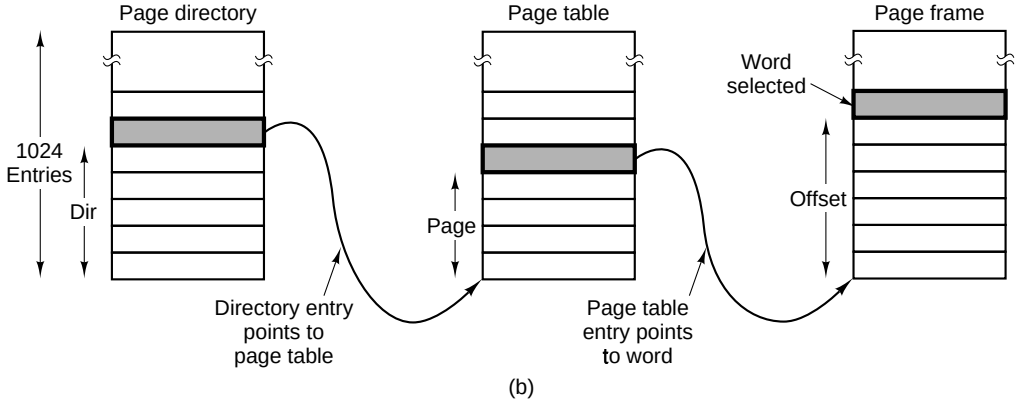
* proces adresuje paměť pomocí některého segmentového registru
* CPU použije odpovídající popisovač segmentu v interních registrech
* pokud segment neexistuje nebo je ”absent”, nastane výjimka
* je-li offset>limit, nastane výjimka
* 32 bitová lineární adresa = báze + offset
* není-li zapnuto stránkování (bitem v globálním řídícím registru), je lineární adresa = fyzická adresa



* je-li zapnuto stránkování, lineární adresa je chápána jako virtuální adresa, která je mapována na fyzickou adresu pomocí tabulek stránek
* pro zmenšení velikosti tabulky stránek je použito dvouúrovňové mapování:



(a)



(b)

- každý program má adresář stránek (jeho začátek je udán registrem),

```
   obsahuje 1024 32 bitových položek = ukazatelů na tabulky stránek
 - každá položka adresáře ukazuje na tabulku stránek, která má také 1024 32
   bitových položek; položka tabulky stránek má 20 bitů číslo rámce, bity R a
   M (pod jinými názvy), bity ochrany a další
 - lineární adresa rozdělena do tří polí:
   . 10 bitů Dir - index do adresáře stránek (zde získáme adresu tabulky
     stránek)
   . 10 bitů Page - index do tabulky stránek (získáme fyzickou adresu rámce)
   . 12 bitů Offset - offset do 4K stránky

* závěr - implementace CPU Intel Pentium umožňuje:
 - čistou segmentaci
 - čisté stránkování (base=0, limit = maximum)
  . používají všechny současné OS pro Pentium
 - stránkované segmenty.

Implementační problém - každý převod lineární adresy na fyzickou by vyžadoval
několik přístupů do paměti. Proto malá asociativní paměť pro mapování
posledních použitých kombinací Dir - Page (nazývána TLB).

Shrnutí metod správy paměti:
............................

* systémy bez odkládání a stránkování (+ základní mechanismy)
 - jednoprogramové - celá paměť patří programu
 - multiprogramování s pevným přidělením - oblasti pevné velikosti,
   program dostane přidělenu oblast
 - multiprogramování s proměnnou velikostí oblasti - každý program
   dostane přidělenu paměť podle jeho potřeby
  . informaci můžeme udržovat pomocí bitmap nebo seznamů
  . alokovat paměť můžeme metodami first fit, next fit, best fit
  . pokud známe další charakteristiky systému, můžeme si vymyslet jiný
    algoritmus (quick fit apod.)
  . ve speciálních případech se používá buddy system (potřebujeme-li
    rychlou alokaci a dealokaci, ale nevadí nám neefektivita využití paměti)

* systémy s odkládáním (swapping) - pokud potřebujeme paměť, odkládáme
  celé procesy na disk
 - paměť i odkládací prostor se přiděluje stejným způsobem jako při
   multiprogramování s proměnnou velikostí oblasti

* systémy s překrýváním (overlays) - program rozdělen na moduly
  (rozděluje programátor), které se zavádějí podle potřeby (zařizuje
  OS)

* systémy s virtuální pamětí - program může být zaveden jen částečně,
  načítání potřebných částí zařizuje OS
 - stránkování na žádost
  . adresní prostor úlohy rozdělíme do stránek
  . paměť rozdělíme do rámců
  . zavedeme mapování mezi stránkami a rámci pomocí tabulky stránek
  . pokud je odkaz na stránku která není v paměti, nastane ”výpadek stránky”
    (page fault) a OS jí zavede a nastaví mapování
  . pokud je paměť plná, odložíme některou stránku na disk (LRU, NRU, FIFO,
    Clock...)
 - segmentace na žádost
 - segmentace se stránkováním

Vstupy a výstupy na nízké úrovni
################################

Principy vstupně/výstupního hardware
====================================

Vývoj rozhraní mezi CPU a zařízeními
------------------------------------

S vývojem postupně narůstala složitost a ”inteligence” jednotlivých komponent
- postupně více a více úloh je prováděno bez přímé účasti CPU:
```

1. CPU přímo řídí periferní zařízení
  - CPU přímo vydává elektrické impulsy potřebné pro zařízení
  - CPU dekóduje přímo signály poskytované zařízením
  - nejjednodušší HW, nejméně efektivní z hlediska využití CPU
  - dnes pouze v jednoduchých mikroprocesorem řízených zařízeních, např. dálkové ovládání televize

2. Mezi CPU a periferii přidán řadič (device controller, adapter)
  - řadič převádí příkazy vydávané CPU na elektrické impulzy pro zařízení
  - poskytuje procesoru informaci o stavu zařízení
  - komunikace mezi CPU a řadičem pomocí registrů řadiče na známých I/O adresách
  - řadič obsahuje také HW buffer alespoň pro jeden záznam (blok, znak, řádku)
  - mnoho řadičů je schopno obsluhovat více stejných (podobných) jednotek
  - rozhraní mezi řadičem a zařízením může být standardizováno (např. SCSI, IDE)
  - např. operace zápisu by se prováděla v těchto krocích:
   . CPU zapíše data do bufferu, informuje řadič o požadované operaci
   . po dokončení výstupu zařízení nastaví příznak, který může CPU otestovat
   . přenos OK => CPU může vložit další data...
  - CPU musí dělat všechno = tzv. programované I/O
  - podstatná část času CPU strávena aktivním čekáním na dokončení I/O operace

3. Řadič umí vyvolat přerušení
  - CPU zadá I/O operaci stejně jako výše, ale nemusí průběžně testovat příznak dokončení
  - při dokončení I/O vyvolá řadič přerušení, CPU začne provádět instrukce na předdefinovaném místě = obslužná procedura přerušení; určí co dál
  - postačuje pro pomalá zařízení, např. sériové I/O

4. Řadič může přistupovat k paměti pomocí DMA
  - DMA přenosy mezi pamětí a buffery
  - CPU vysílá příkazy, při přerušení analyzuje status zařízení apod.
  - vhodné pro rychlá zařízení, např. disky, síťové rozhraní apod.

5. I/O modul rozšířen - umí interpretovat speciální I/O programy
  - název I/O procesor, interpretuje programy v hlavní paměti
  - CPU spouští I/O procesor, I/O procesor provádí své instrukce samostatně

6. I/O modul provádí programy, má vlastní paměť - je de facto počítačem
  - zejména pro složité a časově náročné I/O operace (grafika, zvuk, šifrování atd.)

V PC převážně (3) a (4), ale specializované karty i (6).

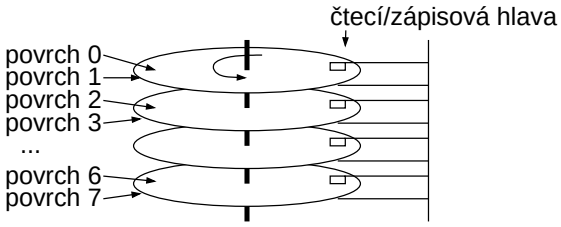
Jak komunikuje CPU s řadičem? Tři možnosti:

1) adresní prostor pro I/O je odlišný od adresního prostoru pro paměť
  - CPU zapisuje do registrů řadiče pomocí speciálních I/O instrukcí
  - vstup: IN R, port
  - výstup: OUT port, R
2) je pouze 1 adresní prostor, pro komunikaci s řadičem vyhrazené adresy (nazývá se ”paměťově mapované I/O”)
  - HW musí umět pro příslušné adresy nebo rámce vypnout cachování
  - pokud chceme zpřístupnit I/O zařízení některému procesu, můžeme ho mapovat do jeho virtuálního adresního prostoru
3) hybridní schéma
  - přístup k řídícím registrům pomocí I/O instrukcí
  - HW buffer je mapován do paměti
  - používá se na PC (buffery mapovány do oblasti 640K až 1 MB)

Magnetické disky
----

* disk, např. pevný disk - jedna nebo více ploten (platter) na společné ose
 - v prehistorických časech počítačů disk pouze jednu plotnu
 - tehdy zvyšování kapacity zvětšováním průměru plotny
 - nyní disk několik ploten, používají se obě strany plotny

- plotny rotují konstantní rychlostí
- data čtou a zapisují hlavičky disku



* geometrie disku:
 - HD několik ploten (běžně průměr od 1.8 do 5.25 in)
 - každá plotna má dvě strany - povrchy (surface) pokryté magnetickým materiálem (podobným jako magnetofonová kazeta nebo disketa)
 - nad každým povrchem ”plave” čtecí/zápisová hlavička (disk head)
 - hlavičky sice mohou být nezávislé, ale z praktických důvodů jsou většinou na společném raménku
 - každá pozice, kterou může zaujímat hlavička = stopa (track)
 - stopa rozdělena do sektorů (sectors) = nejmenší velikost dat, které je možné z disku číst nebo na něj zapisovat
 - množina stop na jedné pozici diskového raménka (tj. všechny stopy ”pod sebou”) tvoří cylindr; data ze stejného cylindru je možné číst/zapisovat bez přesunu hlav

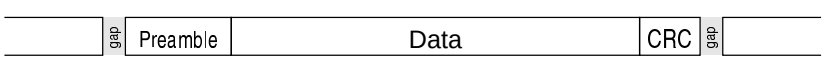
* pro přenos je třeba disková adresa = povrch, stopa, sektor
* moderní disky - aby nebyla menší hustota záznamu informací na vnějších stopách:
 - disk rozdělen do zón, vnější zóny více sektorů než vnitřní
 - navenek disk prezentuje ”virtuální geometrii”, vnitřně přemapovává na skutečnou
 - druhá (lepší) možnost: logické adresování bloků - sektory jsou číslovány od 0 postupně bez ohledu na geometrii disku, disková adresa = číslo sektoru

* pokud požadavek čtení nebo zápisu:
 - čekání, až bude volné zařízení a kanál pro komunikaci s ním
 - vyslán požadavek přesunout hlavičku na požadovanou stopu (seek)
 - žádost o čtení sektoru - až bude záznam pod hlavičkou, přenos dat do bufferu řadiče
 - případný přenos dat pomocí DMA na určené místo hlavní paměti
 - řadič vyvolá přerušení - oznamuje dokončení I/O operace

* doba přístupu k datům - můžeme popsat pomocí:
  . doba vyhledání (seek time) - přesun hlaviček na stopu obsahující záznam (např. 5 ms)
  . latence - doba od přijetí příkazu zařízením do začátku čtení nebo zápisu (rozběh, rotační zpoždění disku - např. také 5ms)
  . rychlost přenosu - v bitech nebo bytech za sekundu (např. 160 MB/s)
  . velikost záznamu = nejmenší adresovatelná jednotka (např. 512 bytů)

Formátování disků
....

* na začátku na disku není žádná informace
* před použitím musí být disk nízkoúrovňově naformátován - každý sektor tvar:



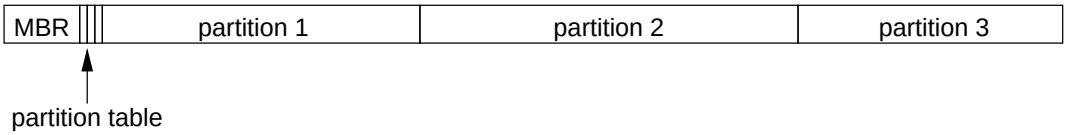
* preambule:
 - začíná bitovým vzorkem, který dovolí HW rozpoznat začátek sektoru
 - obsahuje číslo cylindru a číslo sektoru + další informace
* data (často 512 bytů)
* CRC (často 16 bitů)

* mezi sektory malá mezera

* naformátovaný disk má asi o 20% nižší kapacitu než je tzv.
  neformátovaná kapacita disku

* v některých OS disk rozdělen na několik nezávislých oblastí
  (partitions), každá jako samostatné zařízení (DOS - C:, D:, atd.)
* v některých OS i možnost logického sdružování disků do jednoho
  logického zařízení => angl. název volume, česky nejčastěji svazek
  (Win NT, Linux 2.4)
* oblast nebo svazek možno považovat za ”virtuální disk”

* rozdělení na oblasti po výše popsaném nízkoúrovňovém formátování
 - oblast (partition) = množina po sobě následujících sektorů na disku
 - rozdělení je popsáno v tabulce oblastí (partition table) nebo jinde
* např. na PC:
 - sektor 0 obsahuje MBR (Master Boot Record)
 - tabulka oblastí je uvedena na konci MBR



* po rozdělení na oblasti může být provedeno tzv. vysokoúrovňové
  formátování - do oblasti se zapíše prázdný souborový systém

Obsluha chyb
.....

* všechny současné disky mají defekty
 - způsobeno tím, že hustota záznamu na hranici možností technologie
 - např. 5000 bitů/mm

* dva způsoby jak zacházet s defektními bloky:
 - řešení v řadiči
 - řešení v OS

* řešení řadičem:
 - na disku jsou sektory určené pro nahrazení vadných sektorů (náhradní
   sektory)
 - disk při výrobě otestován, výrobce na něj zapíše tabulku vadných
   sektorů + které náhradní se mají místo nich použít
 - uvedeno také např. v preambuli nebo ve zvláštní tabulce pro každou
   stopu - při požadavku na čtení/zápis vadného sektoru se použije
   příslušný náhradní sektor

Někdy se chyby objevují za chodu. Pokud nastane chyba při čtení sektoru,
řadič se pokusí sektor přečíst znovu. Pokud řadič zaznamená opakované chyby
na některém sektoru (některé chyby totiž mohou ”mizet”), vybere za něj
náhradní sektor, zapíše mapování do tabulky a problematický sektor do
náhradního zkopíruje.

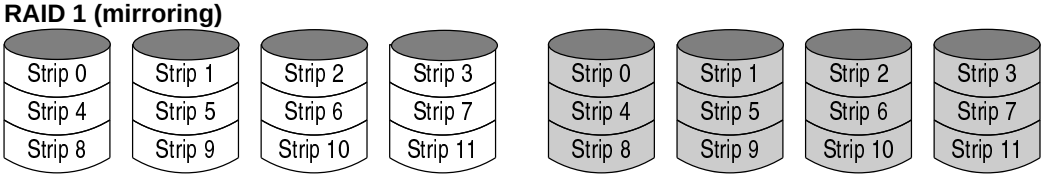
* pokud řadič neumí výše uvedené, musí řešit OS
 - buď má přístup k tabulce vadných sektorů nebo otestuje celý disk
 - musí zajistit, aby se vadný sektor neobjevil v žádném souboru
 - nejjednodušší - při vysokoúrovňovém formátování vytvořit soubor,
   který se bude skládat pouze z vadných sektorů (musí být skrytý, aby
   se ho uživatelé nepokusili číst nebo zrušit)

RAID [v roce 2003 uvedeno pouze pro zajímavost]
....

* jak zvýšit rychlost čtení/zápisu a spolehlivost?
* simulujeme jeden (virtuální) disk pomocí paralelního čtení/zápisu na
  více disků
 - spolehlivost: ukládání redundantních dat
 - výkonnosti: disky vyhledávají a čtou bloky paralelně

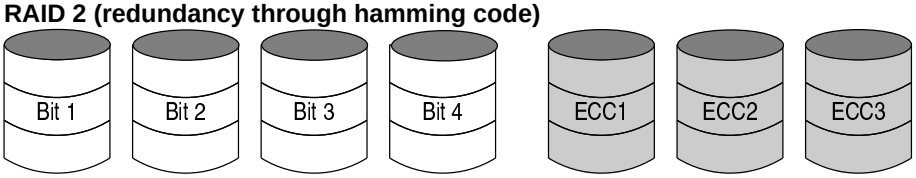
```
* myšlenka se poprvé objevila v článku z r. 1987 od výzkumníků z UCB
  [Patterson, Gibson, and Katz: "A Case for Redundant Arrays of
  Inexpensive Disks (RAID)."]
* odtud pojem RAID (=Redundant Arrays of Inexpensive Disks)
* výrobci HW "přejmenovali" Inexpensive -> Independent

* článek popisuje 5 architektur (úrovní) - RAID 1 až RAID 5

* RAID-1 - zrcadlení disku
 - virtuální disk rozdělen na "strips" velikosti $k$ sektorů (může být
   i $k$=1)
 - při zápisu zapsáno 2x = je vytvořena záloha (na obrázku šedá)
 - čtení rychlejší - požadavky lze rozdělit mezi obě kopie a číst
   paralelně

Poznámka (možné urychlení zápisu)

 Operace zápisu může být urychlena také pokud HW implementace umí
 rozdělit velký požadavek např. na 8 příkazů zápisu a provést tyto
 zápisy paralelně.

[]
```

RAID 1 (mirroring)



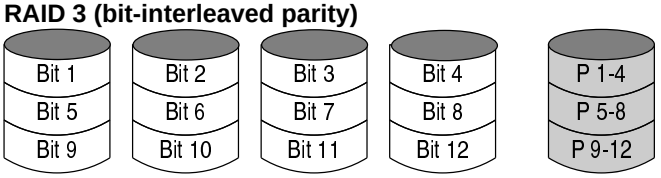

```
* RAID-2 - bitové prokládání, zabezpečení Hammingovým kódem
 - příklad - mějme např. 7 rotačně synchronizovaných disků
   . ke 4 bitům dat přidáme 3 bity Hammingova kódu
   . na každý disk zapíšeme 1 bit
   . seek bude stejně pomalý jako pro 1 disk, ale zápis/čtení 4x rychlejší
 - má smysl pouze pro velký počet disků (např. 32 datových+6 pro zabezpečení)
```

RAID 2 (redundancy through hamming code)



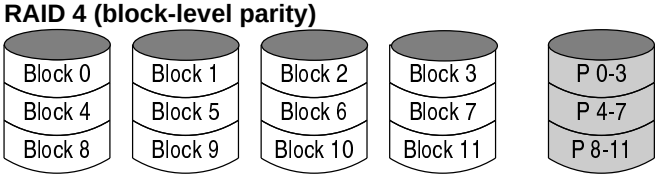

```
* RAID-3 - bitové prokládání, zabezpečení pomocí parity
 - podobně jako RAID-2, ale jediný kontrolní (paritní) disk
 - na paritní disk se zapisuje XOR dat zapisovaných na datové disky
 - havárie jednoho disku => rekonstrukce pomocí XOR všech zbývajících
 - výhoda - ušetříme diskový prostor, čtení/zápis podobně rychlé jako u
   RAID 2
 - nevýhoda - stejně jako pro RAID-2 musejí být disky rotačně
   synchronizované
```

RAID 3 (bit-interleaved parity)



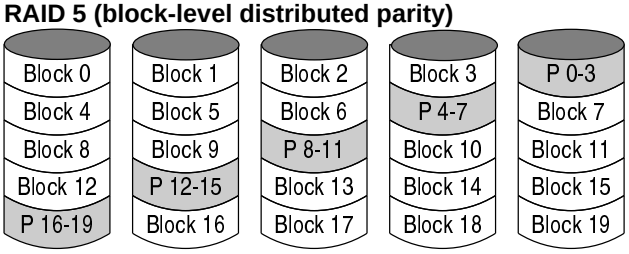

```
* RAID-4 - stripy, jeden paritní disk
 - stejně jako RAID-1 "strips" velikosti $k$ sektorů
 - stejně jako v RAID 3 je jeden disk pole rezervován pro paritu
 - problém malých zápisů - při změně jednoho sektoru by bylo třeba
   přečíst všechny sektory, aby bylo možné změnit paritu
  . malá optimalizace - před zápisem přečíst pouze stará data a starou
    paritu, z toho spočíst novou paritu
  . přesto zápis vyžaduje 2 čtení a 2 zápisy
 - paritní disk se stává úzkým místem
 - výhodné tam, kde je mnohem větší počet čtení než zápisů
```

RAID 4 (block-level parity)



```
* RAID-5 - stripy, parita postupně na všech discích
 - jako RAID-4, ale parita uložena postupně na všech discích
 - umožňuje paralelizaci čtení i zápisu: v jednu chvíli je možný zápis
   na dvě místa ”virtuálního disku”
```

RAID 5 (block-level distributed parity)




```
* nejčastěji implementováno RAID-1 a RAID-5
* mnohé OS poskytují také ”RAID-0”
 - není popsán v původním článku, neobsahuje redundanci dat
 - zvýšení výkonnosti - data rozdělena na jednotlivé disky, čtení po
   dat může probíhat nezávisle
 - méně spolehlivý než pokud jsou data na jednom disku
 - speciální aplikace (zpracování obrazu apod.)
* např. Windows NT implementuje v SW RAID-0, RAID-1 a RAID-5


Plánování pohybu diskového raménka
..................................

* i když se výrobci snaží, aby byl seek (přesun raménka) co
  nejrychlejší, ve srovnání s rychlostí rotace disku je stále pomalý

* jak redukovat průměrný čas čekání?
* naivní řešení: pokud nastane více požadavků na přesun raménka,
  provedeme jako první přesun na nejkratší vzdálenost
* problém vyhladovění: požadavky na blízké cylindry zabrání vykonání
  požadavku na vzdálené cylindry
* v reálných podmínkách by se raménko při větším vytížení pohybovalo
  blízko prostředních cylindrů, na požadavky blízko okrajů by se
  nedostalo

* lepší algoritmus: raménko se pohybuje pouze jedním směrem a obsluhuje
  požadavky na nejkratší přesun tímto směrem
* není-li žádný požadavek v daném směru, směr se obrátí a obsluhují
  se nejbližší požadavky ve druhém směru
* toto schéma se nazývá ”výtahový algoritmus” (elevator algorithm) -
  totéž chování jako výtah (výtahy na ZČU nebo na kolejích se tak
  ovšem nechovají)
```

```
Pro zajímavost uvedu řešení výtahového algoritmu s použitím monitoru
(Hoare 1974):

* předpokládáme, že s disk skládá z N cylindrů, očíslovaných 0 .. N-1
* raménko se může pohybovat ”nahoru” - směrem k vnějšímu cylindru N-1,
  nebo ”dolů” k vnitřnímu cylindru 0

* plánovač disku je součástí OS, má dvě vstupní procedury
 - request(dest) - vyvoláno těsně před vyvoláním přesunu hlavičky na
   cylindr dest
 - release - vyvoláno dokončení všech přenosů na daném cylindru

monitor disk_head_scheduler;

type
    direction = (up, down);

var
    headpos: integer;   { aktuální pozice hlavičky }
    dir:     direction; { směr ve kterém se hlavička pohybuje }
    busy:    boolean;   { true, pokud proces přistupuje k disku }
    upsweep, downsweep: condition;

procedure request(dest)
begin
    if busy then
```

```
        { Když zadám nový požadavek, nastane jedna z následujících situací:
          a) headpos<dest: požadavek můžu vyřídit při cestě ”up”
          b) headpos>dest: požadavek můžu vyřídit při cestě ”down”
          c) headpos=dest: jsem tam, požadavek bych měl vložit do aktuálního
             směru }

        if headpos<dest or (headpos=dest and dir=up)
        then upsweep.wait(dest)      { prioritní čekání, nejmenší číslo }
        else downsweep.wait(N-dest); {                 = nejvyšší priorita}

    busy    := true;
    headpos := dest
end;


procedure release;
begin
    busy := false;
    if dir=up then
        if not empty(upsweep)  { je požadavek v daném směru }
        then upsweep.signal
        else
        begin { není požadavek, změníme směr }
            dir := down;
            downsweep.signal
        end
    else { dir=down, jinak totéž co výše }
        if not empty(downsweep)
        then downsweep.signal
        else
        begin
            dir := up;
            upsweep.signal
        end
end;

begin { inicializace }
    headpos := 0;
    dir     := up;
    busy    := false
end.
```

❇