

## Virtuálna pamäť

- proces si myslí, že má väčšiu pamäť (alebo inak adresovanú) ako je skutočná pamäť v počítači
- vznik v dobe, keď počítače mali malú pamäť
- **overlay** – systém prekrytia (napr. v Turbo Pascale)
  - prechodca virtuálnej pamäti
  - pomocou neho bolo možné napísať program, ktorý sa nezmestí do pamäte. Program sa rozdelil na „unity“ a definovali sa, ktoré zdieľajú pamäť, ktoré nie, ... (čiže čo s čím musí byť naraz v pamäti a čo nie)
  - ale to rozdelenie ostalo na programátorovi
- **stránkovanie** – jeden z druhov virtualizácie pamäte
- virtuálna pamäť sa dá implementovať len v prípade, že prepočet adres robí už procesor (napr. MMU – memory management unit) – CPU posiela MMU virtuálne adresy a ten ich prepočíta na fyzické adresy

### Stránkovanie

- myšlienka: nenechávajme na programátorovi starosti o veľkosť pamäte
- proces dostane adresný priestor  $0..2^n$  – ten rozdelíme na jednotne veľké kusy (**stránky**).  
Typická stránka má 4096 bytov (Intel)

Potom adresy (nech majú 20 bitov):

**0000010**010011000110

**boldom** – číslo stránky

*italicom* – offset (posunutie) vrámci stránky

=> je jednoduché zistiť, v ktorej stránke sa adresa nachádza

- rovnako podelíme aj fyzický priestor na **rámce (frames alebo page frames)** s rovnakou veľkosťou ako stránky

Operačný systém teraz musí riešiť 2 úlohy:

1. pripraviť prepočtovú (stránkovú) tabuľku pre MMU (ktorá stránka je v ktorom rámci)
2. v prípade, že už nemam voľné ďalšie page framy – treba vybrať nejakú stránku a zapísať ju na disk a na jej miesto dať tú, ktorú je zrovna treba

- ak nastane situácia, že adresovaná stránka nie je v pamäti tak -> MMU vygeneruje prerušenie -> spustí sa nejaký podprogram OS, ktorý túto stránku prinesie do pamäti a upraví prepočtovú tabuľku -> zopakuje sa inštrukcia, ktorá volala tú stránku

Stránková tabuľka môže byť uložená:

- **v procesore**
  - problém – lebo by sme v procesoroch potrebovali pomerne veľa pamäte (veľkosť)
- **v pamäti**
  - a budem mať nejaký pointer, ktorý bude ukazovať na začiatok tabuľky
  - pri adresovaní pamäti budem musieť liezť do pamäte 2 krát (rýchlosť)
- v princípe – stránková tabuľka musí byť tak veľká, ako virtuálny adresný priestor každého procesu

- riešením týchto problémov je rozdelenie tabuľky na viac tabuliek.

Adresu rozdelím na 3 časti (10-10-12 bitov):

**010101010101010101010101010101**

a adresujem 2 úrovňovo: prvými 10 bitmi zaadresujem do 1. tabuľky - tam nájdem adresu druhej tabuľky... do nej zaadresujem druhými 10 bitmi - mám frame, posledných 12 bitov - pamäťové miesto vo frame  
=> toto rieši veľkosť, ale je to pomalé

- typicky jeden riadok stránkovej tabuľky obsahuje:
  - page frame number
  - present/absent: 1 - stránka je v pamäti, 0 - nie je v pamäti
  - referenced (bit R): 1 - ak tento riadok bol použitý pri prepočítavaní adresy
  - modified: 1 - ak dáta v stránke boli zmenené odkedy som ju dal do pamäti (aby som vedel či ju musím zapísať na disk keď sa jej budem chcieť zbaviť)
  - caching disabled: 1 - pre túto stránku sa nepoužíva cache (čiže píše sa priamo do pamäte)
  - protection (viac bitov): čo sa môže a čo nemôže robiť s tou stránkou (napr. sa tým dá zakázať prepisovať kód programov)
- reálne sa v procesoroch používa ešte **TLB - Translation Lookaside buffers**
  - stránkové tabuľky mám v pamäti
  - TLB mám priamo v procesore a obsahuje niektoré (posledne používané) riadky zo stránkovej tabuľky
  - veľkosť TLB je obvykle 8..64 riadkov
  - jeden riadok obsahuje:
    - valid - či ten riadok vôbec niečo hovorí
    - virtual page - číslo stránky
    - page frame - číslo framu
    - bity modified a protection
  - riešenie aj rýchlosti - programy obvykle obsahujú cykly a vrámci cyklov sa adresuje len malá časť adresy - preto obvykle počas cyklu sa stačí pozeráť do TLB a nemusím ísť do pamäti
  - hľadanie podľa čísla stránky je v TLB kurvarýchle - je to reálne používané riešenie (napr. v Inteli)
  - ak MMU chce prepočítať adresu -> pozrie sa do TLB či tam nie je záznam o hľadanej stránke -> ak je takýto záznam, priamo sa adresa prepočíta; ak treba nastaví sa bit modified -> ak nie, tak sa ide pozrieť do stránkových tabuliek v pamäti - a info o stránke sa skopíruje do TLB
  - pri rozhodovaní, ktoré stránky udržiavať v TLB opäť nastupujú rôzne algoritmy buď na úrovni procesora (Intel) alebo na úrovni OS.
- pri všetkej snahe - pri veľabitových procesoroch sú stránkové tabuľky stále veľmi veľké. Riešenie: **invertované tabuľky**. Pri napr. 64bitových procesoroch nastáva situácia, že fyzická pamäť je oveľa oveľa menšia ako teoreticky adresovateľný priestor.
- **Hashovacie tabuľky**
  - dostanem napr. 52 bitovú adresu -> hashom z nej vypočítam 16 bitový index -> zaadresujem do tabuľky -> ak pre rovnaký index mám viac framov pozriem do spájaného zoznamu (virtual page/page frame)

## Algoritmy výmeny stránok

- algoritmy, ktoré povedia, že táto stránka už nemusí byť v pamäti
- najlepší možný algoritmus: vyberie stránku, ktorú bude treba adresovať neskôr ako všetky ostatné (príp. vôbec)... jediná škoda je, že sa to nedá implementovať
- **Not Recently Used Page Replacement Algorithm**
  - OS v pravidelných intervaloch vynuluje R bit, ak sa stránka použije, bit R sa nastaví na 0. Čiže ak stránka má  $R = 1$  tak to znamená, že v poslednom čase sa použila
  - Stránky sa na základe R a M bitu rozdelia na:
    - $R=0, M=0$  – najlepší kandidát na výmenu (netreba nič ukladať na disk)
    - $R=0, M=1$  – trochu horsí kandidát na výmenu (stránku ešte treba uložiť)
    - $R=1, M=0$
    - $R=1, M=1$
- **FIFO**
  - musím si pamätať, kedy bola ktorá stránka donesená do pamäti
  - ako prvá pôjde preč stránka, ktorá je v pamäti najdlhšie
- **Second chance**
  - ako FIFO... ak najstaršia stránka má  $R=1$  (sa použila) tak jej dám „druhú šancu“ - hodím ju na koniec zoznamu a nastavím jej  $R=0$
  - vyhodím najstaršiu stránku ak  $R=0$
  - reálna implementácia: **clock algorithm** (cyklický zoznam s jedným smerníkom na „koniec“ kruhu) – netreba nič presúvať v dátovej štruktúre
- **Least recently used (najdávnejšie použité)**
  - rozdiel oproti FIFO (nie kedy prišla do pamäti ale kedy sa naposledy použila)
  - recently used na začiatok, least at rear
  - neefektívne: pri každom adresovaní treba upravovať zoznam stránok
  - alternatíva: použiť pre stránku počítadlo a incrementovať ho pri adresovaní. Raz za čas vynulovať počítadlá.
- **LRU**
  - mám polia of boolean rozmerov  $n \times n$ ; n je počet stránok;
  - na začiatku sú všade nuly
  - ak adresujem i-tu stránku, tak sa vyjedničkuje i-ty riadok a vynuluje i-ty stĺpec => ak potom riadky prečítam v dvojkovej sústave, tak najmenšie číslo je pri najmenej používanej stránke
  - pekne navrhnuté, ale nerealizovateľné (treba veľa veľa veeeeľa pamäti pre tieto polia)
- **NFU – not frequently used**
  - pred každým nulovaním R bitu pripočítam hodnotu R bitu do nejakého počítadla => stránka s najvyšším počítadlom je najviac používaná => problém: zvýhodňuje stránky čo sú dlho v pamäti
  - modifikácia - **aging**: pred každým nulovaním R bitu shiftnem počítadlo vpravo a na začiatok dám hodnotu R bitu. Keď to prečítam ako číslo, tak nedávne použitie má väčšiu váhu ako dávnejšie použitie. Je to reálne implementovateľné priblíženie sa LRU

### - Working Set

- v čase  $t$  je pracovná množina stránok tá, ktorá obsahuje stránky adresované  $k$  tickov dozadu
- empiricky sa zistilo že pracovná množina sa mení pomaly
- myšlienka: pamätajme si pracovnú množinu - obeťou bude tá stránka, ktorá nie je v pracovnej množine = ktorá posledných  $k$  tickov nebola použitá
- implementácia: o stránke si budem pamätať čas posledného použitia a  $R$  bit. Beží nám nejaký virtuálny čas. Pred každým nulovaním  $R$  bitu, ak  $R=1$  tak stránke nastavím posledný čas := virtuálny čas. Keď hľadám obeť, znova ak  $R=1$  tak stránke nastavím posledný čas. Z tých čo majú  $R=0$ , ak ich vek je väčší ako  $x$  tak ho rovno vyhodím, ak jeho vek je menší ako  $x$  tak vyhodím stránku, ktorá má najvyšší vek.
- vylepšenie: **WSClock** – cyklicky
- reálne sa používa Clock (z tých jednoduchších), alebo Aging a WSClock (z tých zložitejších)

## Segmentovanie

- menej používaný prístup ako stránkovanie (ale starší – pojem segment už v 8086)
- myšlienka: pamäťový priestor programu je nejak logicky rozdelený – mám v ňom viac dátových štruktúr, ktoré rastú – čiže si pamäť rozdelím systémom: tu bude toto pole, tu bude toto, atď... ale problém nastane, ak chcem zväčšiť niečo do miest, kde už je niečo iné.
- svet by bol oveľa krajší, ak by bol adresný priestor dvojrozmerný – mal by som napr. 5 segmentov a tie by mohli nezávisle rásť a zmenšovať sa. Potom by sa adresovalo systémom: segment 3, adresa 2156. => túto virtualizáciu by zabezpečoval OS
- o každom segmente si musíme pamätať počiatočnú adresu a veľkosť segmentu. Zároveň by to zozložilo správu pamäti (diery). Pri adresovaní segmentu 2, miesto 1520 sa sčíta počiatočná adresa segmentu + 1520 = fyzická adresa.
- narozdiel od stránok programátor segmenty vidí a sám ich ovláda (zväč mi segment, vytvor segment, zruš segment...)
- použité napr. vo Windows 3.x (napr. pascal sa prekladal: globálne premenné 1 segment, zásobník 1 segment, každý unit 1 segment)
- o segmente sa dá povedať či obsahuje kód alebo dáta (to sa v stránkach nedalo)

### Reálny režim (8086):

dátový, kódový a zásobníkový režim

registre DS, CS a SS (pointre na začiatok dátového, kódového a zásobníkového režimu)

register IP (instruction pointer) neukazoval na najbližší príkaz. Najbližší príkaz bol na adrese CS + IP.

Adresy sa sčítavali systémom (boli 20- bitové):

SEGMENT	0000	
+	OFFSET	
		FYZICKÁ ADRESA

### Chránený režim:

nie adresy na začiatky segmentov, ale čísla segmentov a podľa tabuľky sa im priradia adresy

## Mix stránkovania a segmentovania

### a.) OS Multics

- navrchu boli segmenty a každý segment bol stránkovaný

- adresa sa delila na 3 časti: segment, stránka, offset

## b.) Pentium

### **Ako to funguje v procesoroch Pentium**

#### 86 – reálny režim

#### **Chránený režim - zavedený 286ke**

- zaviedol segmentovanie
- zachoval dvojrozmernosť adries
- adresa sa delí na dve časti:
  - 16 bitový selector (index do tabuľky segmentov – odtiaľ sa zistí adresa segmentu)
  - 32 bitový offset (od 386 - v 286 16-bitový)
- reálna adresa sa potom spočíta ako adresa segmentu + offset \*
- selector sa ďalej delí na:
  - 13 bitový index
  - bit TI
  - 2 bity RPL (bity ochrany)
- TI
  - môže byť 0=GDT, 1=LDT
  - v procesore existujú 2 tabuľky segmentov – lokálna (LDT) a globálna (GDT). Existuje totiž jedna tabuľka spoločná pre všetky procesy a jedna lokálna pre daný proces
- čiže máme  $2^{14}$  segmentov ( $2^{13}$  lokálnych a  $2^{13}$  globálnych)
- každý segment môže byť teoreticky veľký až  $2^{32} = 4\text{GB} \Rightarrow 4\text{GB} * 2^{14} = \text{viac}$ , ako je vôbec možné fyzicky mať v 32-bitovom počítači
- u Billa si povedali, že **Windowsy** nebudú segmentovanie na úrovni OS používať = **flat memory model**
- preto každý proces vlastne dostane 1 segment (=virtuálna pamäť 4GB) ... ale v rámci toho segmentu je to klasická lineárna pamäť. A príkazy ako PridajMiSegment, Uvoľňujem segment nemajú tým pádom význam -> čiže proces sa hraje len v rámci svojho jedného segmentu
- jedna položka v LDT/GDT (deskriptor – popisuje 1 segment) vyzerá takto (má 64 bitov):
  - 32 bitová báza – lineárna adresa do pamäti, kde daný segment začína
  - 20 bitový limit – veľkosť segmentu + 1 bit G, ktorý hovorí – ak G=0 tak limit je v bytoch, ak G=1 tak limit je v 4KB stránkach
  - 1 bytové prístupové práva (rozdelené na 1 bit S (system/application segment), 2 bitový DPL (privilege level), 1 bit P (present – je v pamäti, nie je v pamäti), 4 bitový typ segmentu (typy môžu byť – dátový, kódový, systémový – využíva sa to napr. tak, že do kódového segmentu sa nedá písať, atď...));
- \* adresa ešte nemusí byť fyzická – je to len adresa, ktorá sa dvojúrovňovo prepočíta (stránkovanie) na fyzickú adresu

