KIV/ZOS 2003/2004
Přednáška 9

Relokace a ochrana

Při multiprogramování nastávají dva problémy:

1) programy poběží na různých adresách v paměti - relokace
2) paměť programů musí být chráněna před zasahováním jiných programů - ochrana.

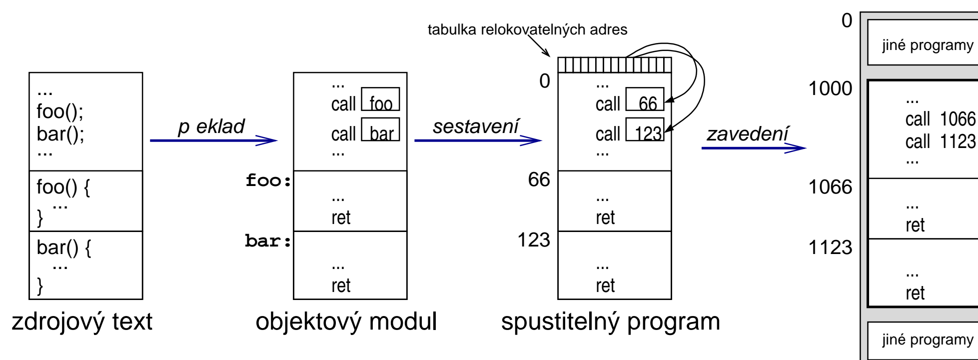
Nyní si uvedeme dva jednoduché způsoby relokace a související mechanismy ochrany.

Relokace při zavedení do paměti

* překlad a sestavení programu
  - většina aplikací je napsaných ve vysokoúrovňovém jazyce
  - SW systémy většího rozsahu jsou rozděleny do modulů, které musejí být přeloženy a sestaveny do spustitelného programu
  - výsledkem překladu jsou tzv. objektové moduly; příkazy ve zdrojovém textu jsou přeloženy do strojových instrukcí, ale zůstávají symbolické reference (odkazy) na adresy proměnných, procedur, fcí apod.
  - výsledný spustitelný program vznikne pomocí sestavení (linking) modulů a knihoven
  - při sestavení se řeší hlavně externí reference:
    . všechna místa výskytu referencí se sdruží do seznamu
    . ve chvíli, kdy se adresa stane známou, vloží se na všechna místa, která ji používají
  - tj. symbolické odkazy se převedou na číselné hodnoty - adresy
  - výsledek - spustitelný program

* komplikace při více programech v paměti
  - představme si, že první instrukcí programu je volání podprogramu na adrese 66 (call 66)
  - pokud bude program umístěn v oblasti začínající od adresy 1000, musí ve skutečnosti provést call 1066 (podobně se posunou všechny ostatní odkazy)

* jedno možné řešení je modifikovat instrukce programu při zavedení do paměti
  - linker musí do spustitelného programu přidat seznam nebo bitmapu označující všechna místa v kódu programu obsahující adresu
  - při zavádění programu do paměti se ke každé adrese přičte adresa začátku oblasti



Relokace při zavedení programu do paměti se nazývá {statická relokace}.

Tímto způsobem pracoval např. OS/MFT fy IBM.

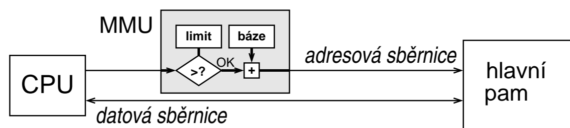
Poznámka pro zajímavost (ochrana paměti při statické relokaci)

* výše uvedené schéma neřeší problém ochrany paměti, tj. proces by mohl
  zasahovat do paměti jiných procesů
* např. IBM 360 řešilo ochranu takto:
 - paměť rozdělena do bloků velikosti 2 KB
 - s každým blokem je hardwarově sdružen 4 bitový kód ochrany
 - PSW procesoru obsahuje 4 bitový klíč
 - při pokusu o přístup k paměti jejíž kód ochrany se liší od klíče v
   PSW nastane výjimka
 - kód ochrany a klíč může měnit pouze OS (privilegované instrukce)
 - výsledek - uživatelské procesy nemohou zasahovat do paměti jiných
   procesů nebo do OS

[]

Mechanismus báze a limitu
....

* mezi CPU a paměť vložíme tzv. jednotku správy paměti (MMU)
* tu vybavíme dvěma registry - nazveme báze a limit
* do báze zavedeme počáteční adresu oblasti
* do registru limit zavedeme velikost oblasti



* fce MMU - dostává adresy od CPU, převádí na adresy do fyzické paměti
 - nejprve zkontroluje, zda adresa není větší než limit
 - je => výjimka
 - není => k adrese přičte bázi

* např. pokud báze=1000 a limit=60
 - adresa 55 -> ok, výsledek=1055
 - adresa 66 -> není ok, výjimka

* tomuto (a podobným) mechanismům se říká {dynamická relokace},
  protože se provádí dynamicky za běhu
* nastavení báze a limitu může měnit pouze OS (privilegované instrukce)

Např. 8086 používá slabší variantu tohoto schématu, bázové registry (=
segmentové registry DS, SS, CS, ES), nemá limit.

Správa paměti s odkládáním celých procesů
===

* pro dávkové systémy můžeme považovat dosud uvedené mechanismy přidělování
  paměti za přiměřené (jednoduchost, efektivita)

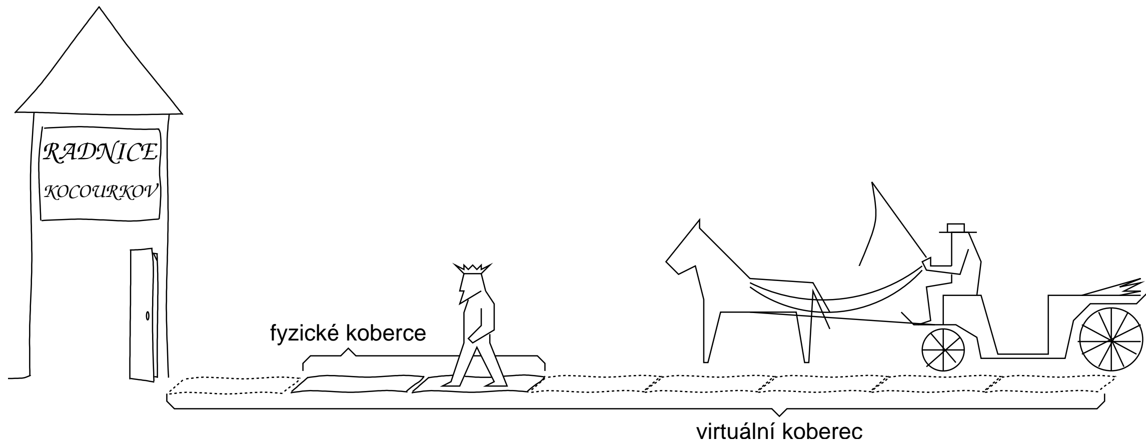
* systémy se sdílením času - odlišná situace: uživatelé mají obvykle
  spuštěno více procesů, než kolik se jich vejde současně do paměti
* používají se dvě strategie:
 - jednodušší - mechanismus odkládání celých procesů, angl. swapping
  . procesy se nevejdou do paměti => některý "nadbytečný" proces se odloží na
    disk
  . v paměti je místo => některý odložený proces se zavede z disku do paměti
  . mechanismus odkládání celých procesů používal například původní
    systém UNIX Version 7
 - složitější mechanismus - virtuální paměť - procesy nemusejí být zavedeny v
   paměti celé
* nejprve popíšeme mechanismus odkládání celých procesů

* problém - kolik paměti má být procesu alokováno?
* pokud je proces vytvořen s pevnou velikostí, je alokace jednoduchá -
  alokuje se přesně kolik je zapotřebí
* ve většině současných programovacích jazyků mohou data procesu růst
 - pro proces je alokováno o něco více paměti než bezprostředně potřebuje

```
 - při požadavku na paměť může růst
 - pokud proces potřebuje překročit alokovaný prostor, můžeme:
  . přesunout proces do dostatečně velké "díry"
  . překážející proces odložit a tím vytvořit prostor pro růst procesu
  . odložit žadatele o paměť, až bude prostor tak ho znovu zavést a alokovat
    mu prostor pro růst
  . proces zrušit (to se nám nelíbí, ale nic jiného nám nezbyde např.
    pokud proces nemůže růst v paměti a odkládací oblast na disku je
    plná)

* pokud procesy mají dva rostoucí segmenty (nejčastěji data a zásobník),
  nabízí se možnost, aby rostly proti sobě
 - při překročení velikosti může být proces opět přesunut, odložen nebo zabit.


Alokace odkládací oblasti
.........................

* buď alokace na celou dobu běhu programu, při skončení dealokována
* nebo alokace při každém odložení

* používají se stejné algoritmy jako pro přidělení paměti, velikost
  oblasti na disku je ovšem zaokrouhlena nahoru na násobek alokační
  jednotky disku


Virtuální paměť
===============

* už před dlouhou dobou problém, že programy jsou větší než dostupná fyzická
  paměť
 - 2 způsoby řešení:
  1. mechanismus překrývání (overlays)
  2. virtuální paměť.

1. překrývání (overlays):
-------------------------

* program rozdělen na části, například moduly
* nejjednodušší varianta: při startu spuštěna část 0, při svém
  skončení zavede část 1, ...
* problém - časté zavádění některých modulů, proto některé systémy
  složitější:
 - více překryvných modulů + data v paměti současně
 - moduly zaváděny podle potřeby, mechanismus odkládání (podobně jako
   u odkládání procesů)

* mechanismus zavádění zařizoval OS, ale rozdělení programů i dat na
  části musel navrhnout programátor
 - vhodné rozdělení do modulů ovlivňuje výkonnost, značná komplikace
 - pro každou úlohu nutné nové rozdělení

=> snaha "hodit to na počítač" (aby to zařídil)

* mechanismus který se stal známý pod názvem virtuální paměť
  (Fortheringham 1961)

2. virtuální paměť
------------------

* potřebujeme počítač s velmi rozsáhlým, prakticky téměř neomezeným
  adresovým prostorem
* kdybychom celý prostor realizovali skutečnou pamětí - neúměrně drahé
* chceme, aby ve skutečné paměti byla realizována pouze část adresového
  prostoru (zbytek může odložen na disku)
* kterou část mít v paměti? - vždy tu, kterou právě potřebujeme
```

Poznámka:

Jako mnoho jiných technických objevů má i tato myšlenka své předchůdce v historii. Podobný nápad ušetřil konšelům města Kocourkova spoustu peněz za koberce, když se chystali uvítat ve městě krále. Neznámý génius přišel tehdy na to, že k (virtuálnímu) pokrytí celé cesty od městské brány až k radnici stačí všeho všudy dva (fyzické) koberce. Když král jeden z nich přejde, musí ho dva sluhové (sloužící coby operační systém) přenést dopředu dříve, než král dojde na konec druhého.



[]

* tj. jinými slovy - budeme chtít, aby fyzická paměť sloužila coby cache virtuálního adresního prostoru procesů

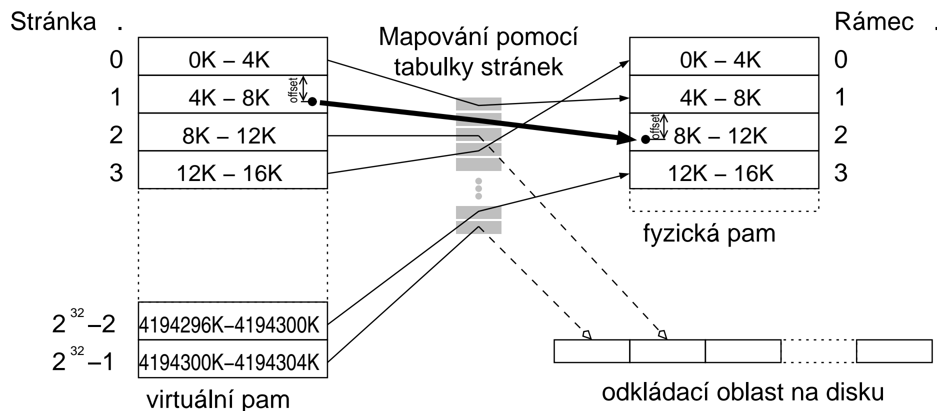
* procesor používá tzv. virtuální adresy (adresy ve virtuálním paměťovém prostoru)
* pokud je požadovaná část virtuálního paměťového prostoru ve fyzické paměti, MMU převede virtuální adresu na fyzickou adresu => přístup k paměti
* pokud není ve fyzické paměti - OS jí musí přečíst z disku
* čtení z disku je I/O => CPU může být dána jinému procesu

* většina systémů virtuální paměti používá techniku nazývanou stránkování (paging)

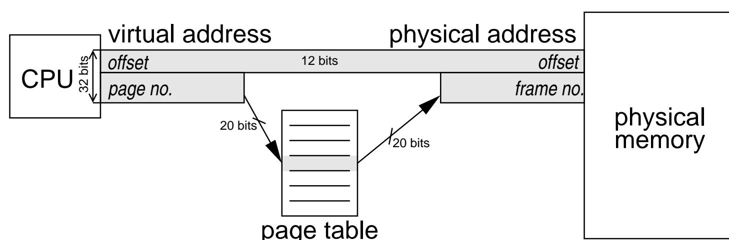
Mechanismus stránkování
....

* program používá (CPU generuje) virtuální adresy
* potřebujeme rychle zjistit, zda je požadovaná adresa v paměti
* pokud je, musíme převést virtuální adresu na fyzickou
* obojí musí být co nejrychlejší (provádí se při každém přístupu k paměti)

* jak převést virtuální adresu na fyzickou?
  - virtuální adresový prostor rozdělí na stránky (pages) pevné délky
  - délka stránky vždy mocnina 2
  - nejčastěji 4KB, běžně se používají od 512 B do 8 KB
  - fyzická paměť rozdělena na části stejné délky - rámce (page frames, česky také stránkové rámy, regály - terminologie je i v angličtině nejednotná)
  - rámec může obsahovat právě jednu stránku
  - na známém místě v paměti je uložena mapa - tabulka stránek
  - tabulka stránek poskytuje mapování virtuálních stránek na rámce




```
* jak vypadá mapovací funkce, která mapuje virtuání adresy na fyzické adresy:

      virtuální adresa .. VA
      fyzická adresa .... FA
      číslo stránky ..... str
      offset ............ offset
      číslo rámce ....... ramec

   Dále předpokládáme, že velikost stránky je např. 4096 bytů.

1. virtuální adresu rozdělíme na číslo stránky a offset:

      str = VA div 4096
   offset = VA mod 4096

2. číslo stránky převedeme na číslo rámce pomocí tabulky stránek:

   Například tabulka stránek může vypadat takto:

   tab_str[0] = 1
   tab_str[1] = 2
   tab_str[2] = -- (stránka není mapována)
   tab_str[3] = 0

   Je-li VA=100, pak:
      str    = 0
      offset = 100
      ramec  = tab_str[str] = tab_str[0] = 1

3. z čísla rámce a offsetu sestavíme fyzickou adresu:

   FA = ramec*4096 + offset = 1*4096 + 100 = 4196

* při realizaci mapovací funkce v MMU nechceme opravdu dělit (dělení je
  pomalé), proto velikost stránky mocnina dvou - např. 2^12 = 4096 (4 KB)
 - nejnižších 12 bitů virtuální adresy = relativní adresa od počátku stránky
   (offset)
 - svrchní bity virtuální adresy (např. horních 20 bitů pro 32 bitovou
   adresu) = pořadové číslo stránky ve virtuálním adresovém prostoru => index
   do tabulky stránek
 - offset nebudeme transformovat, je i relativní adresou uvnitř rámce
```




```
- převod: podle čísla stránky MMU najde číslo rámce, to vyšle na
```

sběrnici
- fyzická paměť neví nic o rámcích, převod realizuje MMU

Ve výše uvedeném příkladu virtuální adresa 8192
=> str=2, offset=0 nelze! - není mapování.

* stránka není mapována - odkaz na ní způsobí událost výpadek stránky
  - výpadek stránky způsobí výjimku, tu zachytí OS (používá se mechanismus přerušení)
  - OS iniciuje zavádění stránky a přepne na jiný proces
  - po zavedení stránky OS vytvoří pro mapování
  - proces může pokračovat

  => dva problémy: kam stránku zavést a odkud?

Poznámka:

Dosavadní výklad byl poněkud zjednodušený:

1. tabulky stránek mohou být velkého rozsahu; např. adresa 32 bitů -> 1 mil. stránek, ne všechny obsazeny - různé optimalizace uložení
2. přístup musí být rychlý - přístup k paměti úzké místo; není možné pokaždé přistupovat k tabulce stránek v hlavní paměti - různá HW řešení, například kopie části tabulky v MMU

[]

Poznámka (pojem vnější a vnitřní fragmentace)
....

* vnější neboli externí fragmentace: zůstávají nepřidělené (nepřidělitelné) úseky paměti
  - např. dynamické přidělování paměti - díry
  - při stránkování vnější fragmentace nenastává, protože všechny stránky jsou přidělitelné
* vnitřní neboli interní fragmentace:
  - část přidělené oblasti je nevyužita
  - např. stránkování: v průměru polovina poslední stránky procesu je prázdná

(Pojem "vnitřní" a "vnější" fragmentace ještě použijeme při popisu souborových systémů.)

[]

Pro zjednodušení dalšího výkladu proberu nejprve mechanismus čistého stránkování (tj. bez odkládání a zavádění stránek z odkládací oblasti), a poté stránkování na žádost (s odkládáním).

Mechanismus čistého stránkování

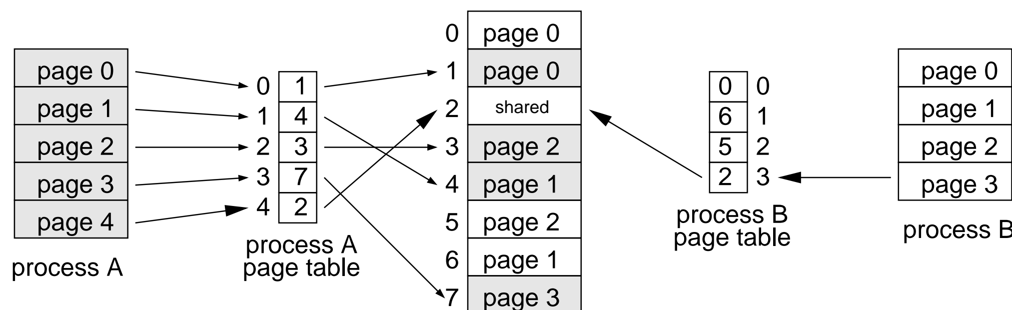
Poznámka:

Někdy se "čisté stránkování" nebo jen "stránkování" uvádí jako samostatný mechanismus (tj. bez souvislosti s virtuální pamětí) - pouze jako řešení problému přidělování paměti. Místo o virtuálním adresním prostoru zde mluvíme o logickém adresním prostoru.

[]

* čisté stránkování
  - stránkování dovoluje, aby (souvislý) logický adresní prostor procesu byl mapován do (nesouvislých) částí paměti (potřebuje-li proces paměť, může být alokován kterýkoli volný rámec)
  - OS udržuje jednu tabulku rámců a pro každý proces tabulku stránek

* tabulka rámců
  - pro správu fyzické paměti
  - informace, které rámce jsou volné a které obsazené




```
* tabulka stránek procesu
 - mapuje číslo stránky na číslo fyzického rámce
 - obsahuje další potřebné informace (příznaky ochrany apod.)
 - řeší problém relokace i ochrany
  . relokace: mapování logických/virtuálních adres na fyzické adresy
  . ochrana: v tabulce stránek procesu jsou pouze ty stránky, ke kterým
    má proces přístup
 - při přepnutí procesu přepnutí MMU na jinou tabulku stránek

Poznámka (rámce je možné sdílet mezi procesy)

  V tabulce stránek dvou nebo více procesů může být odkaz na stejný rámec.
  Tím dosáhneme sdílení části adresního prostoru mezi procesy. Například ve
  výše uvedeném příkladu je rámec č. 2 sdílen mezi procesy A a B.

[]

* dva důležité praktické problémy:
 1) tabulka stránek může být velmi rozsáhlá
 2) převod virtuálních adres na fyzické adresy musí být velmi rychlý

1. problém: jednoduchá tabulka stránek by byla příliš dlouhá
............................................................

Příklad:

 Pokud je virtuální adresa 32 bitů a délka stránky 4 KB (= 2^12 bytů), bude
 celý virtuální adresový prostor 2^20 stránek. Pokud jsou položky tabulky
 stránek dlouhé 32 bitů (4 byty), je pro tabulku stránek každého procesu
 zapotřebí max. (2^20)*4 = 4 MB.

* proces ale obvykle nepoužívá celý virtuální adresní prostor (např. 2^32
  bytů), ale jenom některé jeho části
* příklad uspořádání virtuálního adresového prostoru procesu (proces v OS
  Linux):
 - na začátku virtuálního adresního prostoru je kód (=instrukce) programu
 - pak následují data (oblast "data" pro inicializovaná data a "bss" pro
   neinicializovaná data)
  . oblast "bss" může růst, tj. za ní bude nealokovaný prostor, do kterého je
    v případě potřeby možné zařazovat další stránky
 - pak následuje prostor alokovaný pro sdílené knihovny a jejich data
 - na nejvyšší virtuální adrese je dno zásobníku, zásobník může růst
   směrem dolů a v případě potřeby mu OS může přidělovat další stránky
```

zásobník	**rw-** , 20 KB
	2 145 738 KB neobsazeno
	rw- , 25 KB
/lib/libc-2.2.5.so	**r-x** , 1 126 KB
	rw- , 4 KB
/lib/ld-2.2.5.so	**r-x** , 78 KB
	938 786 KB neobsazeno
bss	**rw-** , 266 KB
data	**rw-** , 12 KB
kód programu	**r-x** , 163 KB

zavad + sdílené knihovny

* myšlenka - v tabulce stránek by mohly být jenom ty stránky, které
  představují existující paměť

* řešení: rozdělit tabulku do menších částí
 - např. 32 bitovou adresu rozdělíme do 3 polí
  . offset - 12 bitů, význam jako obvykle (tj. stránky jsou 4 KB)
  . PT2 - 10 bitů
  . PT1 - 10 bitů
 - při převodu MMU:
  . nejprve použije PT1 jako index do tabulky stránek první úrovně
    (obsahem položky v PT1 je adresa tabulky stránek druhé úrovně)
  . pak použije PT2 jako index do do tabulky stránek druhé úrovně
    (obsahem je číslo rámce)
 - proces může používat např. PT1=0 (4 MB kód+data) a PT1=1 (4 MB sdílené
   knihovny a jejich data), pak až PT1=1023 (4 MB zásobník)
 - tj. tabulka stránek bude mít jen 4 K položek
* PT1 a PT2 můžou být zvoleny jinak velké, adresa se může rozdělit na
  více částí atd. - zde byl uveden pouze princip

2. problém: rychlost převodu virtuálních adres na fyzické adresy
....

* při naivní realizaci by při každém přístupu do stránky nastal přístup do
  tabulky stránek, tj. bylo by 2x více přístupů k paměti
* řešení - HW obsahuje rychlou HW cache pro poslední používané položky
* tato cache často nazývána TLB (Translation Look-aside Buffer)
* je-li položka v TLB, trvá přístup jen asi o 5-10% déle než kdybychom
  přistupovali k paměti přímo (tj. bez mechanismu stránkování)
* při každém přepnutí kontextu musí být TLB vymazána (t.j. než se
  TLB zaplní položkami, bude přístup k paměti pomalý)

* co obsahuje položka v tabulce stránek:
 - číslo rámce
 - příznak platnosti (Valid/invalid) - zda je položka platná, tj. zda
   jí lze použít pro převod čísla stránky na číslo rámce
 - příznaky ochrany (stránka je READ/WRITE nebo READ ONLY) - pokus o
   zápis do READ ONLY stránky způsobí výjimku; některé HW systémy mají další
   bity ochrany
 - příznakové bity "Modified" (někdy nazýván "Dirty") a "Referenced"
  . bit Modified - při zápisu do stránky se automaticky nastaví na 1
  . bit Referenced - nastaven na 1, kdykoli je stránka použita pro
    čtení nebo pro zápis
 - může obsahovat další příznaky (některé popíšeme později)

Poznámka pro zajímavost (invertovaná tabulka stránek)
....

Výše popsaná tabulka stránek slouží pro převod (virtuálního) čísla stránky na (fyzický) rámec. Převod je jednoduchý, protože číslo stránky je součástí virtuální adresy. Představme si ale případ 64bitových počítačů, kde by při 4 KB stránkách by tabulka stránek měla mít max. 2^52 stránek, což není rozumně řešitelné.

Výše naznačený problém řeší některé architektury pomocí tzv. invertované tabulky stránek, která obsahuje položky pro každý fyzický rámec paměti ve formě:

(identifikátor procesu, číslo stránky)

Invertovaná tabulka má vždy rozumnou velikost, protože je dána velikostí fyzické paměti; např. pro 64 bitové virtuální adresy, 4 KB stránky a 256 MB RAM nám stačí 65536 položek (tj. asi 512 KB)

Převod virtuální adresy je ovšem obtížnější:

* pokud je položka v TLB, převod provede HW
* pokud položka není v TLB, nastane výjimka, musí řešit OS (tj. SW) prohledáním invertované tabulky stránek
* položka je nalezena - dvojice (číslo stránky, číslo rámce) se vloží do TLB
* pro optimalizaci se používá tabulka hashovaná podle virtuální adresy

Tento mechanismus používá např. IBM RS 6000 (RS=RISC System).

[]

Stránkování na žádost

* postup při vytvoření procesu
  - vytvoří se prázdná tabulka stránek
  - alokuje se místo na disku pro odkládání stránek
  - v některých systémech se odkládací oblast inicializuje kódem programu a daty ze spustitelného souboru (tj. aby bylo odkud zavádět program, zkopíruje se do odkládací oblasti)

* při běhu procesu
  - na začátku není žádná stránka v paměti
  - při prvním přístupu nastane výpadek stránky (page fault)
  - OS ošetří zavedením stránky do paměti
  - postupně se do paměti dostanou požadované stránky (tzv. pracovní množina stránek)
  - má-li proces svou pracovní množinu stránek v paměti, může dále pracovat bez mnoha výpadků (dokud se pracovní množina stránek nezmění, např. při přechodu do další fáze výpočtu)

[ukázat: jak vypadají odkazy do paměti v čase]

* ošetření výpadku stránky
  - při výpadku stránky vyvolán OS (výjimka, implementováno mechanismem přerušení)
  - OS zjistí z HW registrů, pro kterou virtuální stránku nastal výpadek
  - z toho OS určí umístění stránky na disku (je v nějaké tabulce - často přímo v tabulce stránek)
  - najde rámec, do kterého bude požadovaná stránka zavedena (pokud jsou všechny rámce plné, vyhodí některou stránku z paměti na disk)
  - přečte požadovanou stránku do rámce
  - nastaví příslušnou položku v tabulce stránek
  - vrátí se
  - HW dokončí instrukci, která způsobila výpadek stránky

Algoritmy nahrazování stránek

* pokud všechny rámce obsazené a nastane výpadek stránky, je nutné některou stránku "vyhodit" a rámec uvolnit
* vyhození
  - pokud byla stránka modifikována (Dirty=1), zapíše se na disk
  - pokud oproti kopii na disku nebyla modifikována, bude pouze uvolněna

* otázka - kterou stránku vyhodit?
  - např. vybereme náhodně - kdybychom ale vybrali užívanou stránku, musela by se zase brzy zavést, což stojí čas
  - lepší vybrat takovou, která se dlouho nebude potřebovat
  - existuje mnoho teoretických i experimentálních studií o různých aspektech algoritmů pro nahrazování stránek; my zmíníme jen ty nejzajímavější

Algoritmus First-In, First-Out (FIFO)

Představme si samoobsluhu s plnými regály, která potřebuje zavést nový výrobek (na který byla reklama v televizi). Plné regály => je třeba vybrat výrobek k odstranění. Nejjednodušší algoritmus: vybrat ten, který je nejstarší (předpoklad: je nejstarší, asi ho už nikdo nechce). Analogicky:

* udržujeme seznam všech stránek v pořadí, ve kterém byly zavedeny
* vyhazujeme nejstarší stránku (nejdéle zavedenou = první na seznamu)

To, že bylo zboží zavedeno nejdelší dobu neznamená, že už ho nikdo nechce (např. chleba). Analogicky:

* často používané stránky mohou být v paměti dlouho => je možné, že se vyhodí stránka, která se bude brzy potřebovat

=> FIFO se zřídka používá v čisté podobě

Poznámka (Beladyho anomálie)

* intuitivní předpoklad - čím více bude rámců paměti, tím nastane méně výpadků
* Belady našel protipříklad pro algoritmus FIFO - nazváno "Beladyho anomálie"

* algoritmus FIFO, řetězec odkazů (referencí): 0 1 2 3 0 1 4 0 1 2 3 4

```
3 rámce:   ref.:0 1 2 3 0 1 4 0 1 2 3 4
           ----------------------------
           1| . 0 1 2 3 0 1 4 4 4 2 3 3
           2| . . 0 1 2 3 0 1 1 1 4 2 2
           3| . . . 0 1 2 3 0 0 0 1 4 4
           ----------------------------
                P P P P P P P     P P = 9 výpadků

4 rámce:   ref.:0 1 2 3 0 1 4 0 1 2 3 4
           ----------------------------
           1| . 0 1 2 3 3 3 4 0 1 2 3 4
           2| . . 0 1 2 2 2 3 4 0 1 2 3
           3| . . . 0 1 1 1 2 3 4 0 1 2
           4| . . . . 0 0 0 1 2 3 4 0 1
           ----------------------------
                P P P P     P P P P P P = 10 výpadků
```

* tj. pro 3 rámce nastane 9 výpadků, pro 4 rámce 10 výpadků
* objev pana Beladyho způsobil vývoj teorie stránkovacích algoritmů a jejich vlastností

[]

```
Algoritmus MIN resp. OPT - optimální nahrazování
................................................

* jedním výsledkem studia stránkovacích algoritmů (po objevu Beladyho
  anomálie) byl algoritmus pro optimální nahrazování
* má nejmenší možný počet výpadků stránek

V analogii s obchodem - vyhodíme zboží, které nejdelší dobu nikdo
nebude požadovat.

* algoritmus:
 - v paměti je množina stránek, každá stránka je označena počtem
   instrukcí, po který se k ní nebude přistupovat
   (např. p[0]=10, p[1]=100, p[3]=1000)
 - v okamžiku výpadku stránky se vybere stránka s nejvyšším označením
 - algoritmus je optimální, tj. vybere se stránka která bude zapotřebí
   nejvzdáleněji v budoucnosti

* jediný problém algoritmu je, že není realizovatelný
 - neexistuje způsob, jakým by OS mohl zjistit, která stránka bude
   zapotřebí jako příští
 - optimální algoritmus slouží pouze pro srovnání s realizovatelnými algoritmy
 - program poběží v simulátoru, uchovají se odkazy na stránky a z toho
   se spočte počet výpadků pro MIN/OPT
 - počet výpadků MIN/OPT se srovná s počtem výpadků realizovatelného
   algoritmu (pokud je rozdíl 1%, není možné vylepšení o více než o 1%)

* dále popíšeme realizovatelné algoritmy, nejdřív ten nejlepší co známe

Algoritmus Least Recently Used (LRU, LUR)
.........................................
* česky "nejdéle nepoužitá"

* pozorování (vyplývá z principu lokality):
 - stránky, které se používaly v posledních několika instrukcích se
   budou pravděpodobně používat i v následujících instrukcích
 - stránky, které se dlouho nepoužívaly pravděpodobně nebudou v
   nejbližší době zapotřebí
=> dobré přiblížení se optimálnímu algoritmu: algoritmus LRU - vyhodit
   nejdéle nepoužívanou stránku

Obchod: Vyhazovat zboží, na kterém je v prodejně nejvíce prachu =>
nejdéle nebylo požadováno.

* obtížná implementace
* čistě v software? (není možné)
 - měli bychom seznam stránek v pořadí referencí
 - při výpadku bychom vyhazovali stránku ze začátku seznamu
 - při přístupu ke stránce přesunout stránku na konec seznamu
   zpomalilo by přístup k paměti nejméně 10x
 => algoritmus LRU není SW realizovatelný z výkonnostních důvodů, je
    nutná podpora HW

1. implementace pomocí čítače
 - MMU obsahuje čítač (např. 64 bitů), který je automaticky zvětšen po
   každém přístupu do paměti
 - každá položka v tabulce stránek má pole pro uložení obsahu čítače
 - při každém odkazu do paměti se obsah čítače zapíše do tabulky stránek
   - do položky pro odkazovanou stránku
 - při výpadku stránky se vyhledá a vyhodí stránka, jejíž položka
   obsahuje nejnižší číslo == nejdéle nepoužitá stránka

2. implementace pomocí matice (uvádím pouze pro zajímavost)
 - MMU udržuje matici $n*n$ bitů, kde $n$ je počet rámců
 - na počátku všechny prvky matice nastaveny na 0
 - při odkazu na stránku odpovídající $k$-tému rámci:
   . nejprve nastavit všechny bity $k$-tého řádku matice na 1
   . pak nastavit všechny bity $k$-tého sloupce na 0
 - v každém okamžiku představuje řádek s nejnižší binární hodnotou
```

nejdéle nepoužitou stránku

Příklad:

reference v pořadí: 3 2 1 0

```
     0.1.2.3       0.1.2.3       0.1.2.3       0.1.2.3
  0. 0 0 0 0    0. 0 0 0 0    0. 0 0 0 0    0. 0 1 1 1
  1. 0 0 0 0    1. 0 0 0 0    1. 1 0 1 1    1. 0 0 1 1
  2. 0 0 0 0    2. 1 1 0 1    2. 1 0 0 1    2. 0 0 0 1
  3. 1 1 1 0    3. 1 1 0 0    3. 1 0 0 0    3. 0 0 0 0
```

Výhody algoritmu LRU:

* z časově založených algoritmů nejlepší
* Beladyho anomálie nemůže pro LRU nastat

Nevýhody:

* při každém odkazu na stránku nutnost aktualizace záznamu (položky v tabulace stránek nebo řádek & sloupec v matici) => zpomalení výpočtu (místo jednoho odkazu do paměti se provádějí dva)

=> LRU se pro stránkovanou virtuální paměť prakticky nepoužívá (používá se v jiných případech, např. bloková cache pro soubory)

Proto teď uvedu ještě několik dalších (snáze implementovatelných) algoritmů různé kvality.

❇