B4B35OSY: Operační systémy

Lekce 7. Alokace paměti

Petr Štěpán
stepan@fel.cvut.cz

26. října, 2022

Outline

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

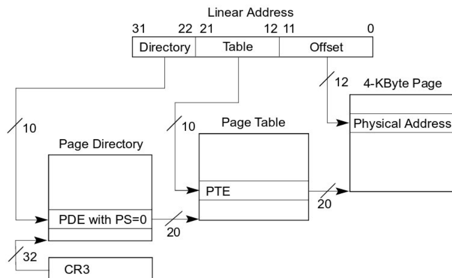
4 Systém NOVA

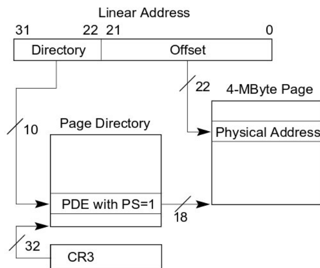
5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Dvouúrovňové stránkování 32-bitů

- 32-bitový procesor se stránkou o velikosti 4 KiB – 12 bitů posunutí (offset)
- 10 bitů index v tabulce tabulek (page directory – PT_0)
- 10 bitů index v tabulce stránek (page table – PT_1)
  - při nastaveném bitu PS – velikost stránky 4 MiB, nepoužije se tabulka PT_0





Více úrovňové stránkování (32bitů) – bitová aritmetika

Tabulka tabulek - vrchních 10 bitů adresy

```
pdir[virt >> 22]
```

Test přítomnosti tabulky stránek v paměti

```
if ((pdir[virt >> 22] & PRESENT) == 0)
// tabulka stranek neni v pameti
```

Pozice tabulky stránek v paměti

```
ptab = pdir[virt >> 22] & ~PAGE_MASK;
```

Tabulka stránek - prostředních 10 bitů adresy

```
ptab[(virt >> PAGE_BITS) & 0x3ff]
```

Kvíz

Více-úrovňové stránkování:

A - **zrychluje** nalezení odpovídající paměti, ale **zvyšuje** velikost paměti pro uložní tabulky.

B - **zrcyhluje** nalezení odpovídající paměti a **zmenšuje** velikost paměti pro uložní tabulky.

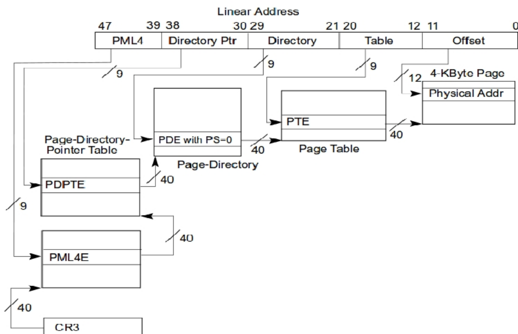
C - **zpomaluje** nalezení odpovídající paměti, ale **zmenšuje** velikost paměti pro uložní tabulky.

D - **zpomaluje** nalezení odpovídající paměti i **zvyšuje** velikost paměti pro uložní tabulky.

Stránkování IA-32e

$4\times$ pomalejší pokud není nalezen překlad v TLB.

- Lineární adresa 48 bitů – Virtuální prostor o velikosti 256 TiB
- Fyzická adresa 52 bitů – což je 4 PiB RAM
- Varianty s 4KiB, 2MiB nebo 1GiB stránkami
- Posunutí 12 bitů, 21 bitů, nebo 30 bitů
- 9 bitů indexy do tabulek tabulek/stránek



Kvíz – ukazatele

Pokud v programu je ukaztel `int *p = &a;` Pak ukazatel `p` obsahuje:

A - virtuální adresu.

B - fyzickou adresu v RAM.

C - ukazatel do stránkovací tabulky.

D - záleží na typu překladače.

Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Opakování virtualizace – stránkování

- Tabulka stránek pro každou stránku obsahuje
  - Příznak, zda je ukazatel na rámec platný
  - Ukazatel na rámec, kde jsou data, nebo tabulka nižší úrovně
  - Další příznaky ohledně práv a přístupu k datům v této stránce
- Každý proces má pro sebe celý virtuální prostor (až na část věnovanou OS)
  - Každý proces má stránkovací tabulku (alspoň tu na nejvyšší úrovni).
  - Proces využívá jen malou část virtuálního prostoru.
  - Nepoužitý virtuální prostor nezabírá RAM, prostě ho nelze využít k ukládání a čtení hodnot.
    - V 64-bitovém systému většina stránkovacích tabulek neexistuje, protože jimi mapovaný virtuální prostor není použit.
    - Stránkovací tabulky jsou uloženy v RAM, takže jejich neexistencí se šetří další RAM
- Jeden rámec (část RAM) může být ve více virtuálních prostorech
  - Sdílení kódu programů a dynamických knihoven

Virtualizace paměti

- Sdílený kód
  - Jediná read-only kopie kódu ve FAP sdílená více procesy
    - více virtuálních instancí editoru, shellů, jen jednou ve FAP.
- Privátní data
  - Každý proces si udržuje svoji vlastní virtuální kopii kódu a svoje reálná data
  - Stránky s privátním kódem a daty mohou být kdekoliv v LAP
- Sdílená data
  - Potřebná pro implementaci meziprocesní komunikace (mmap)

Odkládání na disk

- Úsek FAP přidělený procesu je vyměňován mezi vnitřní a vnější (sekundární) pamětí oběma směry
  - Uložení stránek na disk, načtení z disku
  - Swap out, swap in (roll out, roll in)
  - Trvání výměn je z podstatné části tvořena dobou přenosu mezi pamětí a diskem je úměrná objemu vyměňované paměti
- Při nenalezení stránky ve fyzické paměti, nebo při porušení práv při přístupu do stránky nastane přerušení – chyba stránky (page fault)
- POZOR – některé části systému nelze odložit na disk
  - obsluha přerušení, která spravuje výpadek stránky
  - data potřebná k obsluze tohoto přerušení

Principy stránkování

Kdy stránku zavádět do FAP? (Fetch policy)

- Stránkování při spuštění
  - Program je celý vložen do paměti při spuštění
  - velmi nákladné a zbytečné, předem nejsou známy nároky na paměť, dříve se nevyužívalo, dnes je využívána
- Stránkování či segmentace na žádost (Demand Paging/Segmentation)
  - Tzv. „líná metoda“, nedělá nic dopředu
  - Řeší problémy s dynamickou alokací proměnných
- Předstránkování (Prepaging)
  - Nahrává stránku, která bude pravděpodobně brzy použita
- Čištění (Pre-cleaning)
  - změněné rámce jsou ukládány na disk v době, kdy systém není vytížen
- Kopírovat při zápisu (copy-on-write)
  - Při tvorbě nového procesu není nutné kopírovat žádné stránky, ani kódové ani datové. U datových stránek se zruší povolení pro zápis.
  - Při modifikaci datové stránky nastane chyba, která vytvoří kopii stránky a umožní modifikace

Líná metoda – Demand paging

- Při startu procesu zavede OS do FAP pouze tu část programu (LAP) kam se předává řízení – vstupní bod programu
  - Pak dochází k dynamickému zavádění částí LAP do FAP po stránkách „na žádost” tj. až když je jejich obsah skutečně referencován
- Pro překlad LA $\rightarrow$ FA se využívá Tabulka stránek (PT)
  - Sada stránek procesu, které jsou ve FAP – rezidentní množina (resident set)
  - Odkaz mimo rezidentní množinu způsobuje přerušení výpadkem stránky (page fault) a tím vznikne „žádost“
    - Proces, jemuž chybí stránka, označí OS jako pozastavený
    - OS spustí I/O operace k zavedení chybějící stránky do FAP (možná bude muset napřed uvolnit některý rámec, viz politika nahrazování dále)
    - Během I/O přenosu běží jiné procesy; po zavedení stránky do paměti se aktualizuje tabulka stránek, „náš“ proces je označen jako připravený a počká si na CPU, aby mohl pokračovat
- Výhoda: Málo I/O operací, minimum fyzické paměti
- Nevýhoda: Na počátku běhu procesu se tak tvoří série výpadků stránek a proces se „pomalu rozbíhá“

Princip lokality

- Odkazy na instrukce programu a data často tvoří “shluky”
- Vzniká časová lokalita a prostorová lokalita
  - Provádění programu je s výjimkou skoků a volání podprogramů sekvenční
  - Programy mají tendenci zůstávat po jistou dobu v rámci nejvýše několika procedur
  - Většina iterativních výpočtů představuje malý počet často opakovaných instrukcí,
  - Často zpracovávanou strukturou je pole dat nebo posloupnost záznamů, které se nacházejí v „sousedních“ paměťových lokacích
- Lze pouze dělat odhady o částech programu/dat, která budou potřebná v nejbližší budoucnosti

Heuristiky stránkování

- Předstránkování (Pre-paging)
  - Sousední stránky LAP obvykle sousedí i na sekundární paměti, a tak je jich zavádění poměrně rychlé
    - bez velkých přejezdů diskových hlaviček
  - Platí princip časové lokality – proces bude pravděpodobně brzy odkazovat blízkou stránku v LAP. Zavádí se proto najednou více stránek
  - Výhodné zejména při inicializaci procesu – menší počet výpadků stránek
  - Nevýhoda: Mnohdy se zavádějí i nepotřebné stránky
- Čištění (Pre-cleaning)
  - Pokud má počítač volnou kapacitu na I/O operace, lze spustit proces kopírování změněných stránek na disk
  - Výhoda: uvolnění stránky je rychlé, pouze nahrání nové stránky
  - Nevýhoda: Může se jednat o zbytečnou práci, stránka se ještě může změnit

Copy-on-write

Kopírování až při zápisu

- velmi vhodné při vytvoření procesu – služba fork
- kód je sdílen, ten se nekopíruje ve FAP, pouze se připojí do nového virtuálního prostoru (zkopíruje se pouze odpovídající část stránkovací tabulky)
- data by měla být vlastní, měla by se vytvořit kopie dat ve FAP
  - není nutné to dělat, pokud nikdo (rodič ani potomek) nebude data měnit
  - to se dá pojistit zakázáním zápisu do stránek dat
    - při zápisu do stránky, nastane chyba stránkování (page fault)– OS zjistí, že je potřeba tuto stránku zkopírovat mezi rodičem a potomky (mohlo dojít k více voláním fork)
  - složitost této metody je dána možností vytvoření více potomků se stejnými datovými stránkami, z nichž některé jsou již zkopírovány a některé sdíleny

Stránkování – politika nahrazování

- Co dělat, pokud není volný rámec ve FAP
  - Např. při startu nového procesu
- Politika nahrazování (Replacement Policy)
  - někdy též politika výběru oběti
- Musí se vyhledat vhodná stránka pro náhradu (tzv. oběť)
  - Kterou stránku „obětovat“ a „vyhodit“ z FAP?
  - Kritérium optimality algoritmu: minimalizace počtu (či frekvence) výpadků stránek

Co dělá kdo?

HW – CPU (MMU)

1. MMU automaticky převádí logickou adresu programu na fyzickou adresu podle tabulek stránek
2. když MMU nemůže převést logickou adresu na fyzickou vyvolá výjimku (přerušení)

SW – operační systém

1. při svém zavádění nastaví CPU, aby používalo stránkování (tj. zapne MMU)
   - Kvůli zpětné kompatibilitě podporují moderní procesory více typů stránkování (32-bit, 64-bit, PAE, ...). OS si rozhodne, jaký typ se použije (většinou ten nejnovější, podporovaný jak HW, tak OS)
2. plní obsahy tabulek stránek, aby logické adresy odpovídaly určeným fyzickým rámcům (každý proces má vlastní tabulku stránek)
3. řeší výjimky (výpadky stránek) – přístup k virtuálním adresám, které:
   - nejsou mapovány do fyzické paměti (buď stránka na HDD (swap), nebo typicky chyby v programu, např. dereference NULL ukazatele)
   - jsou mapovány s jinými příznaky (např. zápis do read-only stránky $\Rightarrow$ buďto chyba nebo aktivace copy-on-write)

Stránkování – výběr oběti

- Určení oběti:
  - Politika nahrazování říká, jak řešit problémy typu:
    - Kolik rámců procesu přidělit?
    - Kde hledat oběti?
    - Jen mezi stránkami procesu, kterému stránka vypadla nebo lze vybrat oběť i mezi stránkami patřícími ostatním procesům?
- Některé stránky nelze obětovat
  - Některé stránky jsou dočasně „zamčené“, tj. neodložitelné
  - typicky V/V vyrovnávací paměti, řídicí struktury OS, ...
- Je-li to třeba, musí se rámec zapsat na disk („swap out“)
  - Nutné to je, pokud byla stránka od svého předchozího „swap in“ modifikována. K tomu účelu je v PT příznak dirty (modified) bit, který je automaticky (hardwarově) nastavován při zápisu do stránky (rámce).

Algoritmus FIFO

Hledáme algoritmus, který je rychlý a vede na nejmenší počet výpadků stránek

- Obětí je vždy nejstarší stránka
- FIFO – jednoduché, rychlé, ale neefektivní
- Nevýhoda – i staré stránky se používají často

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	5	5	5	5	4	4	4
2		2	2	2	2	2	2	1	1	1	1	5	5
3			3	3	3	3	3	3	2	2	2	2	2
4				4	4	4	4	4	4	3	3	3	3

Celkem 10 výpadků

Optimální algoritmus

- Oběť – stránka, ke které bude přistupováno (čtení či zápis) ze všech nejpozději
  - tj. po nejdelší dobu se s ní nebude pracovat
- Budoucnost však v reálném případě neznáme
  - lze jen přibližně predikovat
- Lze užít jen jako porovnávací standard pro ostatní algoritmy
  - Zpětně při analýze jiných algoritmů „známe budoucnost“

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	2	2	2	2	2	2	2	2	4	4	4
3			3	3	3	3	3	3	3	3	3	3	3
4				4	4	4	5	5	5	5	5	5	5

Celkem 6 výpadků

Algoritmus LRU

- Predikce založená na historii
  - Předpoklad: Stránka, ke které nebylo dlouho přistupováno, nebude potřeba ani v blízké budoucnosti
- Oběť – stránka, ke které nejdéle nikdo nepřistoupil
  - LRU se považuje za nejlepší aproximaci optimálního algoritmu
  - bez věštecké křišťálové koule lze těžko udělat něco lepšího

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	1	1	1	1	1	5	5
2		2	2	2	2	2	2	2	2	2	2	2	2
3			3	3	3	3	5	5	5	5	4	4	4
4				4	4	4	4	4	4	3	3	3	3

Celkem 8 výpadků

LRU – implementace

- Řízení časovými značkami
  - Ke každé stránce (rámci) je hardwarově připojen jeden registr, do nějž se při přístupu do stránky hardwarově okopírují systémové hodiny (time stamp)
  - Při hledání oběti se použije stránka s nejstarším časovým údajem
  - Přesné, ale náročné jak hardwarově tak i softwarově
    - prohledávání časovacích registrů
    - každá instrukce musí modifikovat časovou značku 1–2 stránek
- Zásobníková implementace
  - Řešení obousměrně vázaným zásobníkem čísel referencovaných stránek
  - Při použití přesune číslo stránky na vrchol zásobníku
  - Při určování oběti se nemusí nic prohledávat, oběť je na dně zásobníku
  - Problém:
    - Přesun na vrchol zásobníku je velmi náročný, hardwarově složitý a nepružný; softwarové řešení nepřichází v úvahu kvůli rychlosti
    - Nutno dělat při každém přístupu do paměti!

Aproximace LRU

- Příznak přístupu (Access bit, reference bit) – a-bit
  - Spojen s každou stránkou, po „swap-in“ = 0, při přístupu ke stránce hardwarově nastavován na 1
- Algoritmus druhá šance
  - Používá a-bit, FIFO seznam zavedených stránek – tzv. mechanismus hodinové ručičky
    - Každé použití stránky nastaví a-bit
    - Každé ukázání hodinové ručičky způsobí vynulování a-bitu (stránka dostane druhou šanci)
    - Obětí se stane stránka, na niž ukáže hodinová ručička a a-bit je nulový
  - Akce ručičky závisí na hodnotě a-bitu:
    - a=0: vezmi tuto stránku jako oběť
    - a=1: vynuluj a-bit, ponechej stránku v paměti a posuň ručičku o pozici dále
  - Jednoduché jako FIFO, při výběru oběti se vynechává stránka aspoň jednou referencovaná od posledního výpadku
  - Numerické simulace – dobrá aproximaci čistého LRU

Algoritmus druhé šance

- Ukázka příkladu s algoritmem druhé šance

číslo rámce	1	2	3	4	1	2	5
1	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$5_{a=1}$
2		$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=0}$
3			$3_{a=1}$	$3_{a=1}$	$3_{a=1}$	$3_{a=1}$	$3_{a=0}$
4				$4_{a=1}$	$4_{a=1}$	$4_{a=1}$	$4_{a=0}$

2	1	5	4	5	3
$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=0}$
$2_{a=1}$	$2_{a=0}$	$2_{a=0}$	$2_{a=0}$	$2_{a=0}$	$3_{a=1}$
$3_{a=0}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$
$4_{a=0}$	$4_{a=0}$	$4_{a=0}$	$4_{a=1}$	$4_{a=1}$	$4_{a=0}$

Celkem 7 výpadků

Modifikovaná druhá šance

- Algoritmus označovaný též NRU (not recently used)
  - Vedle a-bitu se používá i bit modifikace obsahu stránky (dirty bit, d-bit)
    - nastavován hardwarem při zápisu do stránky
  - Hodinová ručička maže a-bity
    - proto je možná i stránka s nastaveným d-bitem a nulovým a-bitem

d	a	Význam
0	0	stránka se vůbec nepoužila
0	1	ze stránky se pouze četlo
1	0	stránka má modifikovaný obsah, ale dlouho se k ní nepřistupovalo
1	1	strákna má modifikovaný obsah a byla i nedávno použita

- Pořadí výběru (da): 00, 01, 10, 11
- Využití d-bitu šetří nutnost zápisu modifikované stránky na disk před odstraněním z paměti

Přidělování rámců procesům

- Pevné přidělování
  - Procesu je přidělen pevný počet rámců
    - buď zcela fixně, nebo úměrně velikosti jeho LAP
  - Podhodnocení potřebného počtu rámců ⇒ velká frekvence výpadků
  - Nadhodnocení ⇒ snížení maximálního počtu spuštěných procesů
- Prioritní přidělování
  - Procesy s vyšší prioritou dostanou větší počet rámců, aby běžely „rychleji“
  - Dojde-li k výpadku, je přidělen rámec patřící procesu s nižší prioritou
- Proměnný počet rámců přidělovaných globálně (tj. z rámců dosud patřících libovolnému procesu)
  - Snadná a klasická implementace, užíváno mnoha OS (UNIXy)
  - Nebezpečí „výprasku“ (thrashing)
    - mnoho procesů s malým počtem přidělených rámců ⇒ mnoho výpadků
- Proměnný počet rámců přidělovaných lokálně (tj. z rámců patřících procesu, který způsobil výpadek)
  - Metoda tzv. pracovní množiny (working sets)

Thrashing

- Jestliže proces nemá v paměti dost stránek, dochází k výpadkům stránek velmi často
  - nízké využití CPU
  - OS „má dojem“, že může zvýšit počet běžících vláken/procesů, aby se CPU víc využilo, protože se stále se čeká na dokončení I/O operací
    - odkládání a zavádění stránek
  - Tak se dostávají do systému další procesy a situace se zhoršuje
- Thrashing – ”Výprask”– počítač nedělá nic jiného než výměny stránek

Pracovní množiny

Model pracovní množiny procesu P_i (working set) WS_i

- Množina stránek, kterou proces referencoval při posledních n přístupech do paměti ($n \approx 10.000$ – tzv. okno pracovní množiny)
- Pracovní množina je aproximace prostorové lokality procesu. Jak ji ale určovat?
  - Při každém přerušení od časovače lze např. sledovat a-bity stránek procesu, nulovat je a pamatovat si jejich předchozí hodnoty. Jestliže a-bit bude nastaven, byla stránka od posledního hodinového „tiku“ referencována a patří do WS_i
  - Časově náročné, může interferovat s algoritmem volby oběti stránky, avšak účelné a často používané
  - Pokud suma všech WS_i (počítaná přes všechny procesy) převýší kapacitu dostupné fyzické paměti, vzniká „výprask” (thrashing)
    - Snadná ochrana před „výpraskem” – např. jeden proces se pozastaví

Četnost výpadků stránek

- Linux nepočítá pracovní množiny, ale četnost výpadků stránek
- Pro každý proces se udržuje statistika, kolik výpadků stránek nastalo v čase
- Procesy s vyšší četností výpadků dostanou více reálné paměti
- Procesy v nižší četností mohou mít méně reálné paměti
- Thrashing nastane, pokud četnost výpadků všech procesů bude růst

Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

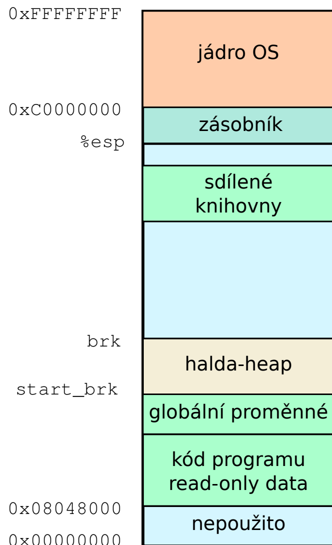
4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

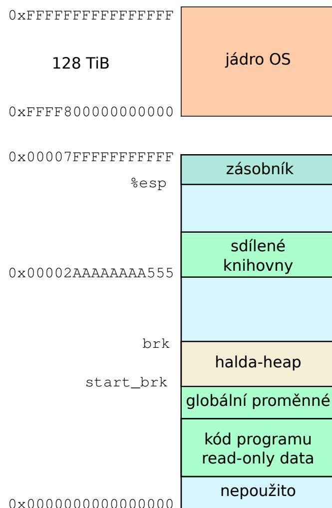
Rozdělení paměti 32-bitový systém Linux

- Virtuální paměťový prostor procesu je rozdělen na:
  - systémovou část – dostupnou pro proces systémovými voláními (1GiB pro OS, Windows má dokonce 2GiB)
  - uživatelskou část – prostor pro program, zásobník (pro všechna vlákna) a jeho data
    - část program, statická data
    - část halda – heap, dynamická data až do 3GiB
    - část mma – mapovaná paměť, dynamické knihovny
    - část zásobník, limit 8MiB
- paměťová mapa procesu je dostupná v /proc/__pid__/maps



Rozdělení paměti 64-bitový systém

- Virtuální paměťový prostor procesu je rozdělen na:
  - systémovou část – dostupnou pro proces systémovými (128 TiB – virtuálně je místa dostatek)
  - uživatelskou část – prostor pro program, zásobník (pro všechna vlákna) a jeho data
    - část program, statická data
    - část halda – heap, dynamická data až do 42TiB
    - část mma – mapovaná paměť
    - část zásobník, limit 8MiB



Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Kvíz – stránkování

`(pdir[virt>>22] & present) == 1` pokud:

A - je odpovídající stránka v paměti RAM.

B - není odpovídající stránka v paměti RAM.

C - je odpovídající tabulka stránek v paměti RAM.

D - je odpovídající tabulka tabulek (page directory) v paměti RAM.

Kvíz – stránkování

```
#define PAGE_BITS   12
#define PAGE_SIZE   (1 << PAGE_BITS)
#define PAGE_MASK   (PAGE_SIZE - 1)

ptab[(virt>>12) & 0x3ff] & ~ PAGE_MASK
```

je:

- A - ukazatel do RAM na virtuální adresu virt.
- B - ukazatel do RAM na začátek rámce, kde jsou data z adresy virt.
- C - ukazatel na tabulku stránek do RAM, kde je uloženo číslo rámce.
- D - číslo rámce.

Start prvního procesu

Detekce ELF souboru

- Prohlédněte si `kern/src/ec.cc`

```
Eh *eh = static_cast<Eh *>
   (Ptab::remap(mod.mod_start));
if (eh->ei_magic != 0x464c457f
  || eh->ei_class != 1
  || eh->ei_data != 1
  || eh->type != 2
  || eh->machine != 3)
   panic ("No ELF");
```

`mod.mod_start` ukazuje do RAM, kam zavaděč (bootloader) nahrál připravený uživatelský program

`virt = Ptab::remap(phys)` nastaví stránkovací tabulku tak, aby ukazatel phys byl namapován na virtuální adresu virt, aby OS mohlo číst data z konkrétní fyzické adresy

`eh` ukazuje na třídu Eh - Elf header viz `kern/include/elf.h`

kontrola magického čísla ELF a spustitelnosti (typ cpu, OS, ...)

Start prvního procesu

```
mword virt=align_dn(ph->v_addr, PAGE_SIZE);
mword phys=align_dn(ph->f_offs
  + mod.mod_start, PAGE_SIZE);
mword fsize=align_up(ph->v_addr%PAGE_SIZE
  + ph->f_size, PAGE_SIZE);
mword msize=align_up(ph->v_addr%PAGE_SIZE
  + ph->m_size, PAGE_SIZE);

while (msize > 0) {
  if (fsize == 0) {
    void *page=Kalloc::allocator.alloc_page
                         (1, Kalloc::FILL_0);
    if (!page)
      panic ("Not enough memory\n");
    phys = Kalloc::virt2phys(page);
  }
  if (!Ptab::insert_mapping(virt, phys, attr))
    panic ("Not enough memory\n");
  phys += PAGE_SIZE; virt += PAGE_SIZE;
  msize -= PAGE_SIZE;
  fsize -= (fsize ? PAGE_SIZE : 0);
}
```

Mapamování dat a programu do virtuálního prostoru

- Prohlédněte si `kern/src/ec.cc`

Namapovat všechny části (program headers) do virtuálního prostoru procesu.
V případě, že programová část není v RAM, tzn. f_size==0, pak je potřeba alokovat rámec funkcí `alloc_page`.

- `ph->v_addr` virtuální adresa, kam má být část umístěna
- `ph->f_offs` posunutí (offset) části od začátku souboru
- `ph->f_size` velikost programové části v souboru, může být nula
- `ph->m_size` velikost programové části v paměti

Start prvního procesu

Vytvoření zásobníku a spuštění procesu

- Prohlédněte si `kern/src/ec.cc`

```
void *stack = Kalloc::allocator
  .alloc_page (1, Kalloc::FILL_0);
if (!stack)
  panic ("Not enough memory\n");
if (!Ptab::insert_mapping (
      0xbfffff000, Kalloc::virt2phys (stack),
      Ptab::PRESENT | Ptab::RW | Ptab::USER))
  panic ("Not enough memory\n");

current->regs.ecx = 0xc0000000;
current->regs.edx = eh->entry;

ret_user_sysexit();
```

Nejdříve si připravíme jeden rámec RAM pro zásobník – stack. Tento rámec připojíme do virtuálního prostoru procesu na adresu `0xbfff000`. Nasimuluje, jako kdybychom se měli vrátit z volání sysenter a pak zavolat sysexit:

- `ecx` - ukazuje na konec zásobníku (adresa `0xc0000000` se nevyužije, první push použije adresy `0xbffffffc`-`0xbfffffff`)
- `edx` - vstupní virtuální adresa pro spuštění programu, uvedena v hlavičce ELF souboru

Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Alokace paměti

- Paměť není neomezená
  - musí být alokována a spravována
  - mnoho aplikací velmi intenzivně potřebuje paměť
    - prohledávání stavového prostoru, chemická analýza složitých molekul, mapování v robotice
- Chyby při alokaci a správě jsou závažné a špatně detekovatelné
  - pokud je to možné, využívejte nástroje pro kontrolu alokace paměti – valgrind, -fsanitize=address
  - Přístup do paměti není vždy stejně rychlý
    - při programování je nutné brát ohled na využití cache, která je daleko rychlejší, než hlavní paměť počítače
    - úprava programu na lepší využití cache může významně zrychlit výpočet programu

Alokace paměti

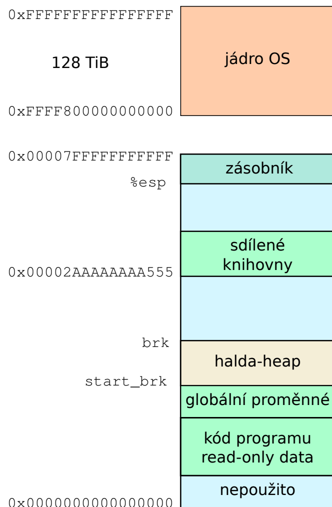
- Statická velikost, statická alokace
  - globální proměnné (i static proměnné jednotlivých modulů)
  - linker přiřadí místo pro globální proměnné ve virtuální paměti
  - zavaděč (OS) přiřadí konkrétní místo ve fyzické paměti
- Statická velikost, dynamická alokace
  - Lokální proměnné na zásobníku (stack)
  - překladač generuje kód tak, že k datům na zásobníku se přistupuje relativně vzhledem k jeho vrcholu (či tzv. frame pointeru)
- Dynamická velikost, dynamická alokace
  - kontroluje program
  - alokace na heapu – jak?

Dynamická alokace

- Explicitní a implicitní alokace
  - Explicitní – program alokuje a uvolňuje paměť pro dynamické proměnné
    - např. funkce malloc a free v jazyce C, new/delete v C++
  - Implicitní – program alokuje paměť pro nové proměnné, ale již je neuvolňuje
    - např. garbage collection v Javě nebo Pythonu
- Alokace v obou případech uvažuje paměť jako množinu bloků

Proces v paměti

- brk – omezení haldy, nejvyšší možná využitelná paměť (kromě zásobníku a sdílených knihoven)
- `int brk(void *addr)` a `void *sbrk(intptr_t increment)` posouvají využitelnou paměť pro haldu
- zvětšení `brk` je spojené s alokací fyzické paměti a namapováním virtuální paměti na alokovanou fyzickou



Balíček malloc

```
#include <stdlib.h>
```

- `void *malloc(size_t size)`
  - Při úspěšné alokaci:
    - vrací ukazatel na blok paměti o velikosti alespoň `size` bajtů, (typicky velikost zarovnaná na 8-bajtové bloky)
    - při `size == 0`, vrací `NULL` a nic nealokuje
    - Při nedostatku paměti: vrací `NULL` (0) a nastaví errno
- `void *calloc(size_t nmemb, size_t size)`
  - Varianta malloc která navíc inicializuje paměť na 0
- `void free(void *ptr)`
  - Vrací blok `ptr` do ”bazénu”(pool) volných bloků paměti
  - `ptr` muselo být alokované původně funkcí `malloc`
- `void *realloc(void *ptr, size_t size)`
  - Změní velikost bloku `ptr` a vrátí ukazatel na nový blok s novou velikostí
  - Obsah nového bloku se nezmění až do minima z hodnot nové a staré velikosti
  - Pokud nejde blok zvětšit, alokuje se nový blok a původní data se do něj zkopírují
  - Opět – `ptr` muselo být alokované původně funkcí `(m/c)alloc`, nebo `realloc`

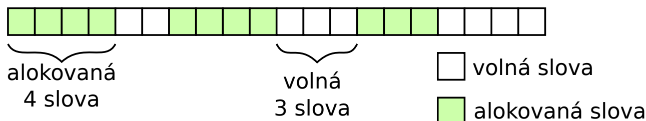
Cíle alokace

- Hlavní cíle
  - co nejrychlejší provedení funkcí malloc a free, měla by být rychlejší než lineárně k počtu alokovaných bloků
  - minimalizovat fragmentaci paměti, co nejlepší využití paměti souvisí s minimální fragmentací (vnitřní i vnější)
- Vedlejší cíle
  - Prostorová lokalita:
    - bloky alokované v podobném čase by měly být blízko u sebe
    - bloky podobné velikosti by měly být blízko u sebe
  - Implementace by měla být robustní:
    - operace free by měla proběhnout pouze na správně alokovaném objektu
    - alokace by měla umožnit kontrolovat, zda se jedná o odkaz na alokované místo

Předpoklady

Konvence pro další část přednášky

- paměť je adresována po slovech (slovo v 32-bitových systémech je 4bajtové, v 64-bitových systémech je 8bajtové)
- čtverečky na obrázcích znamenají slovo
- každé slovo může obsahovat buď celé číslo, nebo ukazatel



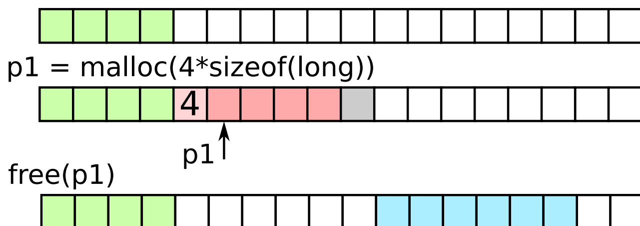
Problémy alokace

- Jak zjistit kolik paměti uvolnit při volání funkce free?
- Jak udržovat informaci o volných blocích v paměti?
- Co udělat s volným místem, pokud alokujeme paměť v díře, která je větší než požadované množství?
- Jak vybrat místo pro alokaci, když jich vyhovuje víc?
- Jak vrátit alokovaný blok paměti po uvolnění do volných bloků?

Velikost alokované paměti

Jak zjistit jak velká paměť se má uvolnit

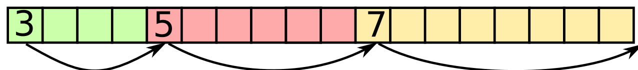
- Informaci o velikosti bloku lze uchovat před začátkem alokovaného bloku
- Číslo před začátkem udává velikost bloku a je označováno jako hlavička
- Informace vyžaduje dodatečné místo při alokaci



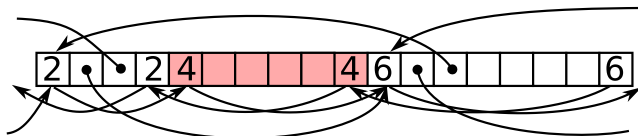
Informace o volných blocích

4 základní metody udržování informace o alokovaných blocích a volném místě

1. implicitní seznam s použitím délky



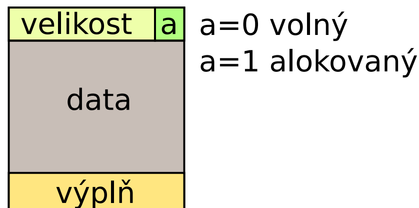
2. explicitní seznam volných bloků



3. rozdílné seznamy volných bloků podle velikosti
4. uspořádaný seznam bloků podle velikosti

Implicitní seznam

- potřebujeme identifikovat, zda je blok volný, nebo použitý
- rezervujeme extra bit
- bit použijeme ve stejném slově jako velikost bloku, protože velikost bloku v bajtech je vždy násobek slova (4 nebo 8 bajtů), můžeme použít nejnižší bit délky.

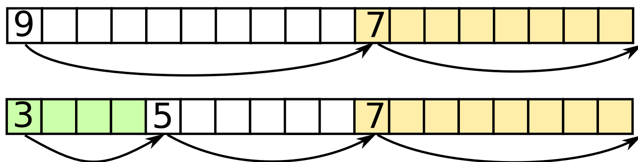


Alokace paměti

- alokace znamená přepsání bitu “a” na 1, nebo rozdělení velkého bloku na dva
- který blok vzít pro alokaci:
  - první blok (first fit) – vezmi první blok od začátku haldy, který má alespoň požadovanou velikost
    - může být náročné, prohledávání celé haldy
    - nezohledňuje velikost, mohou vznikat mini-díry
  - další blok (next fit) – stejné jako první blok, ale pamatuje si kde skončil minule a začne na tom místě
    - rychlejší algoritmus, opět nezohledňuje velikosti
  - nejlepší blok (best fit) – najdi ze všech bloků ten, který má nejbližší velikost (nejlépe přesně požadovanou)
    - nejnáročnější, vždy prohledá celou haldu, vznikají mini-díry
  - nejhorší blok (worst fit) – najděte blok, který je největší
    - stačí si pamatovat, kde je největší blok, po rozdělení je nutné hledat největší blok

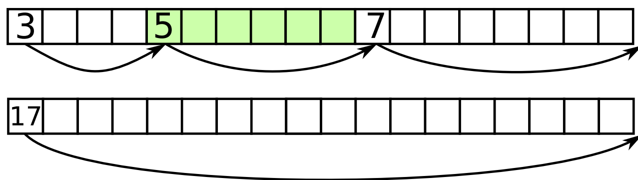
Alokace paměti

- blok volné paměti se při alokaci:
  - rozdělí volný blok na dva bloky, jeden alokovaný, druhý volný
  - celý volný blok se použije jako alokovaný, pokud by zbytek byl moc malý (nemohl by uchovat informaci o volném bloku)



Uvolnění paměti

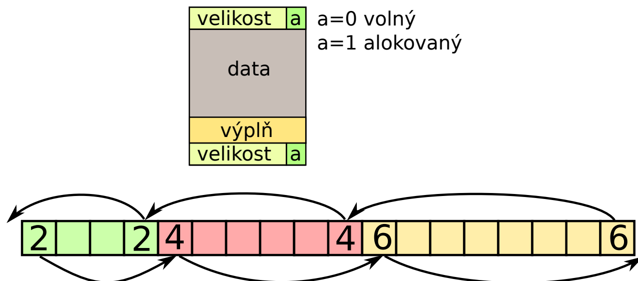
- uvolnění paměti je přepsání bitu “a” na 0 a spojení s předchozím a nebo následujícím volným blokem
- jak najít předcházející blok?
  - buď musíme projít celou haldu a spojovat dopředu
  - nebo musíme něco přidat do naší struktury bloku



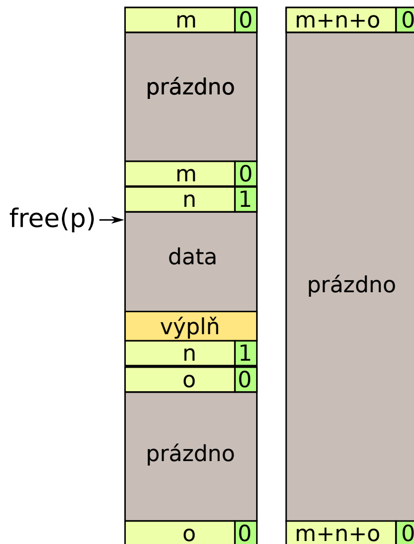
Obousměrný implicitní seznam

Co přidat

- také na konec bloku přidáme jeho velikost [Knuth1973]
- spojení prázdných bloků v konstantním čase

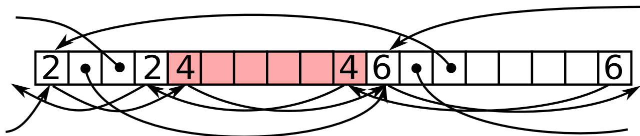


Uvolnění paměti



Explicitní seznam volných bloků

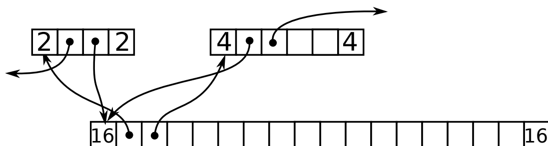
- princip využít volné místo v bloku pro uložení přímých ukazatelů na další a předchozí volný blok
- potřebujeme ponechat velikost bloku (na začátku i na konci), protože je nutná pro spojování uvolněných bloků



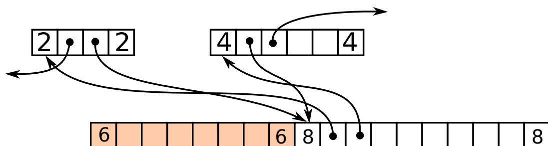
Alokace s ukazateli

- pokud je využita jen část bloku, pak se jen posunou ukazatele

Před:



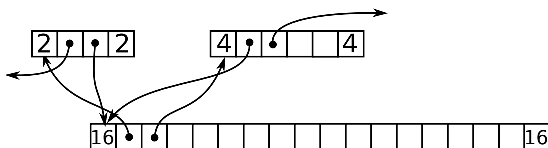
Po:



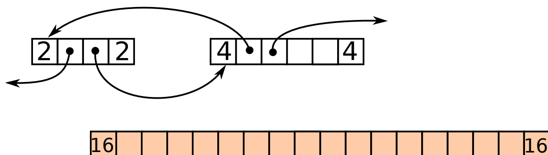
Alokace s ukazateli

- pokud je využit celý blok, pak je třeba přepojit ukazatele předchozího a následujícího volného bloku

Před:



Po:



Uvolnění bloku

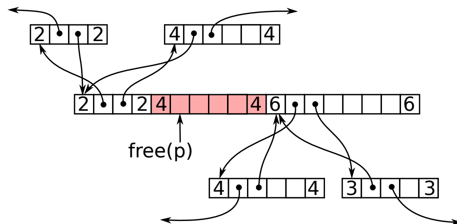
kam zařadit volný blok

- FIFO – zařadit blok na konec fronty (bere se i zprostřed, podle strategie výběru)
  - pro: rychlé, konstantní čase
  - proti: vyšší fragmentace
- podle adresy – udržovat setříděný seznam podle adresy
  - pro: studie ukazují, že je menší fragmentace
  - proti: zatřídění uvolňovaného bloku trvá déle

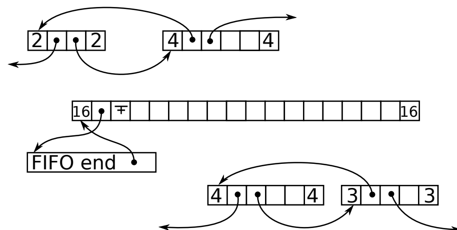
Uvolnění bloku FIFO

- spojování bloků, nutné přepojit ukazatele původních bloků

Před:

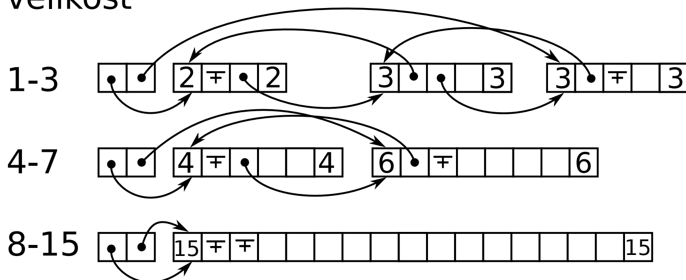


Po:



Oddělené seznamy

- seznamy volných bloků podle jejich velikostí
- jeden seznam obsahuje bloky o velikosti v zadaném rozmezí
  - rozmezí většinou limity podle mocnin 2



Oddělené seznamy

- alokace bloku o velikosti **m**:
  - najdu první volný blok o velikosti **n**, že $m < n$
  - pokud je blok výrazně větší ($n - m$ umožňuje začlenit blok do oddělených seznamů) pak vytvořím prázdný blok a vložím ho do seznamu volných bloků rozměru $n - m$
  - pokud blok není výrazně větší, pak použiji na alokaci celý nalezený blok
- uvolnění bloku:
  - je nutné zkontrolovat sousední bloky a pokud byly volné, vytvořit nový blok o větší velikosti a umístit ho do seznamu správné velikosti

Vyhledávací struktura

- všechny volné bloky jsou setříděny ve struktuře, která umožní vyhledat nejvhodnější velikost (best-fit) v čase – $O(log(n))$
- vyhledání prvního většího volného bloku je stejně náročné – $O(log(n))$
- nejčastěji se používají červeno-černé stromy (red-black tree), které jsou relativně jednoduché na implementaci a přitom efektivní

Obsah

1 Opakování

2 Virtualizace paměti

3 Rozdělení paměti

4 Systém NOVA

5 Uživatelská alokace paměti

6 Alokace fyzické paměti

Zóny paměti x86

- NUMA (non-uniform memory access) – u víceprocesorových systému trvá přístup do jednoho místa v paměti jinak dlouho, podle toho, z jakého procesoru přistupuji – souvisí s fyzickým umístěním paměťových čipů na základní desce
- UMA (uniform memory access) – jediná paměť se stejným přístupem
- Přestože PC jsou UMA i tak má paměť různé zóny, vzhledem k omezení přístupu periferií do fyzické paměti:

jméno	rozsah
ZONE_DMA	0–16 MiB of memory
ZONE_NORMAL	16–896 MiB
ZONE_HIGHMEM	896 MiB – End

- DMA – paměť vhodná pro použití komunikace s periferiemi, hlavně DMA přenosy HDD – RAM (starší periferie neuměly adresovat víc než 16 MiB paměti)
- NORMAL – paměť celá mapovaná do oblasti jádra OS
- HIGHMEM – veškerý zbytek paměti, který se nevejde do NORMAL.

Pozor – 32bitový systém má pro jádro vyhrazen 1 GiB (někdy 2 GiB) adresového prostoru. Pokud je fyzické paměti víc než 1 (2) GiB, nemůže být všechna fyzická paměť namapována do adresního prostoru jádra současně a mapování (obsah stránkovacích tabulek) se musí měnit podle toho, do jaké paměti je potřeba přistupovat. To hodně zpomaluje běh systému.

Alokace v jádře

Třída Kalloc - zkráceno

```
class Kalloc {
 private:
   const mword begin;
   mword end;
 public:
   enum Fill {
    NOFILL = 0, FILL_0,
    FILL_1
   };

   static Kalloc allocator;
   Kalloc (mword virt_begin,
    mword virt_end) :
    begin (virt_begin), end (virt_end) {}

   void * alloc_page (unsigned size,
    Fill fill = NOFILL);
   void free_page (void *);

   static void * phys2virt (mword);
   static mword virt2phys (void *);
};
```

Třída Kalloc - definice

```
void * Kalloc::phys2virt (mword phys) {
  mword virt = phys +
    reinterpret_cast<mword>(&OFFSET);
  return reinterpret_cast<void*>(virt);
}

mword Kalloc::virt2phys (void * virt) {
    mword phys = reinterpret_cast<mword>(virt) -
      reinterpret_cast<mword>(&OFFSET);
    return phys;
}
```

Přechod virtuální a fyzická adresa

Jak je možné, že přechod mezi virtuální a fyzickou adresou je tak jednoduchý?

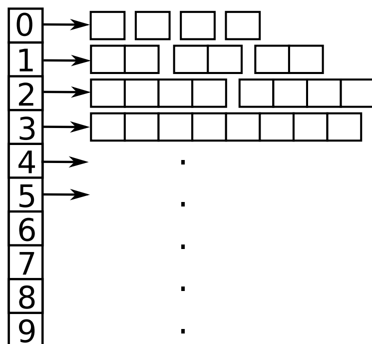
```
mword Kalloc::virt2phys (void * virt) {
  mword phys = reinterpret_cast<mword>(virt) -
    reinterpret_cast<mword>(&OFFSET);
  return phys;
}
```

Platí to vždy?

- Ne - platí to jen pro úsek paměti, který je celý namapován do fyzické paměti, jako jeden blok
- DŮLEŽITÉ: Po zapnutí stránkování nemůže ani JOS přistoupit přímo k fyzické paměti
  - Celé jádro OS se pohybuje také ve virtuálním prostoru
- OS si pro sebe zabere zónu DMA a NORMAL a využívá ji přednostně pro alokaci objektů, u kterých potřebuje znát fyzickou adresu
  - Tabulky stránek
    - OS musí vyplnit rámec tabulky stránek do tabulky tabulek
  - Přístup k souborům na disku přes DMA
    - OS musí vyplnit rámec, kam se budou kopírovat data z DMA
    - DMA neví nic o stránkování
    - DMA není procesem, ale HW

Zóny paměti

- každá zóna si udržuje seznam volných a použitých rámců
- pokud počet volných rámců klesne pod stanovenou mez, spouští se swap démon, který začne připravovat odložení stránek na disk
- při dosažení spodní limitní hranice se alokující proces blokuje do ukončení uvolňování stránek
  - některé procesy nelze blokovat, ty mohou provést alokace i pod tento spodní limit
- každá zóna si uržuje seznam volného místa v blocích stránek



Zóny paměti

- seznam řádu **k** znamená, že udržuje volné bloky o velikosti $2^k \times PAGE_SIZE$
- pokud není volný rámec požadovaného řádu, vezme se nejbližší volný blok vyššího řádu, rozdělí se poloviny a přesune do nižšího řádu, případně rekurzivně až do požadovaného řádu
- uvolnění bloku může vést ke spojení se sousední blokem a přechod do vyššího řádu
- k detekci možného spojení slouží bitová mapa pro všechny bloky daného řádu
- každá zóna má seznamy řádu 0 až MAX (většinou 10 – blok 4MiB)
- zóna má funkce alloc_pages, free_pages pro alokaci bloku stránek zadaného řádu a jeho uvolnění

Mapa rámců paměti RAM

- každý rámec fyzické paměti má v paměti alokovanou strukturu, která popisuje jeho využití

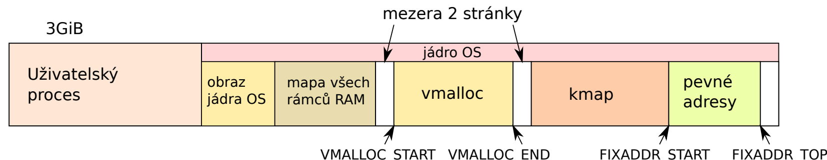
```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    struct page **pprev_hash;
    struct buffer_head * buffers;
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
    void *virtual;
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

Rámce paměti

- uvedená struktura slouží pro uchování všech informací o rámci:
  - list – odkaz na spojový seznam rámců (např. clean, dirty, lock)
  - mapping – odkaz na soubor, který tento rámec mapuje do paměti
  - count – kolik stránek odkazuje na tento rámec (např. sdílené stránky kódu procesů)
  - flags – příznaky stránky (např. active, inactive, unused, dirty, slab, lru)
  - lru – odkaz do spojového seznamu pro algoritmus lru – active/inactive seznam
  - buffer – odkaz hlavičky bufferů, pokud stránka slouží jako disková cache, nebo má svoji kopii ve swap
  - virtual – (volitelně) odkaz na virtuální adresu stránky, pro HIGMEM rámce v jádru (má pouze jednu virtuální adresu)

Rozložení paměti jádra (Linux)

- obraz jádra – rozbalený kód jádra OS
- mapa rámců paměti – struct page pro každý rámec RAM
- oblast vmalloc – oblast pro vmalloc
- oblast kmap – oblast kam jsou dočasně mapovány stránky z HIGHMEM
- oblast fixovaných adres – adresy, které je potřeba znát před spuštěním jádra např. ACPI



Alokace v jádře

- kmalloc – funkce pro alokaci dynamických objektů v prostoru jádra, vytváří fyzicky i virtuálně spojité úseky paměti. Pro alokaci používá algoritmus známý jako SLAB.
- vmalloc – alokace celých stránek (spojitých ve virtuální paměti, ne nutně spojitých ve fyzické paměti)
- GFP (get free page) příznaky – kde, co a jak alokovat
  - GFP_ATOMIC – nelze proces uspat, je nutné dokončit alokaci přímo
  - GFP_NOHIGHIO – alokace v jádře, proces může být uspán, ale požaduje paměť ze zóny NORMAL
  - GFP_KERNEL – obyčejná alokace v jádře, proces může být uspán a vzbuzen po uvolnění paměti, zóna HIGMEM
  - GFP_HIGHUSER – obyčejná alokace normálního uživatele v zóně HIGHMEM, může být uspán

vmalloc

- alokuje v jádře blok paměti, který ve fyzické paměti nemusí být souvislý
- kroky vmalloc:
  - najdi místo ve virtuální paměti pro daný blok
  - alokuj potřebný počet stránek pro daný blok
  - upraví referenční tabulku jádra OS
  - při prvním přístupu procesu k alokovanému prostoru se vyskytne chyba stránky a OS nakopíruje informace o rámci z referenční tabulky stránek OS do tabulky stránek OS.
- velikost bloků paměti alokovaná vmalloc je zaokrouhlena na stránky
- mezi jednotlivé alokované bloky je vložena prázdná stránka, která chrání před přesahy (off-by-one chyby) mezi bloky paměti

kmalloc

- SLAB (SLUB) algoritmus
- SLAB má tři základní cíle:
  - alokovat malé bloky paměti a při tom zabránit fragmentaci paměti
  - udržovat volné nejčastěji používané objekty pro jejich rychlé opětovné využití (cache uvolněných objektů stejné velikosti, např. datagramy)
  - lepší využití procesorové L1 a L2 cache zarovnáním objektů na velikost L1, nebo L2 cache
- informace o využití SLAB cat /proc/slabinfo
- objekty jsou alokované z tzv. cache, tj. struktur, které obsahují alokované stránky pro objekty dané velikosti
  - standardně se vytvoří cache pro základní typy velikostí (obdoba oddělených seznamů podle velikostí)
  - cache vytváří uživatelé, tedy části jádra OS, pokud chtějí zrychlit alokaci mnoha malých kousků paměti
  - kmalloc při požadavku na alokaci paměti najde cache, která nejlépe odpovídá požadované velikosti
  - cache zajistí alokaci zjištěním volného místa v slab – kontejneru na několik objektů zadané velikosti

Cache pro kmalloc

- každá cache má 3 seznamy slab podle obsazenosti:
  - plný – tento slab nelze použít pro alokaci
  - poloprázdný – tento slab je kandidátem pro alokaci nového objektu
  - prázdný – tento slab je kandidátem pro uvolnění
- každá cache se snaží udržovat rozumný počet volného místa pro další alokaci objektů zadané velikosti
- od jádra 2.6 se vytváří také cache per-CPU
  - tyto cache jsou spojeny s procesory a snaží se, aby objekty, které patří jednomu vláknu byly umístěny ve stejné L1, nebo L2 cache, tedy fyzicky blízko
  - tato procesorově orientovaná alokace umožňuje, aby se data při běhu lépe vešla do procesorové cache a tím aby proces běžel rychleji