

B4B350SY: Operační systémy

Lekce 7. Alokace paměti

Petr Štěpán

stepan@fel.cvut.cz



14. prosince, 2017

Outline

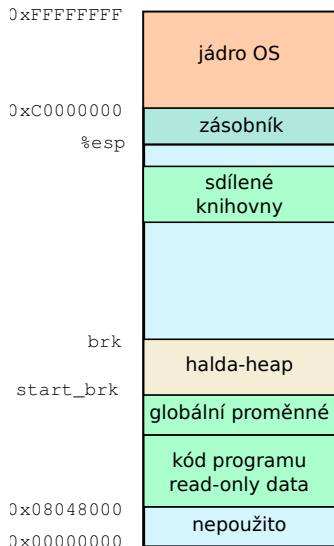
- 1 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 Alokace fyzické paměti

Obsah

- 1 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 Alokace fyzické paměti

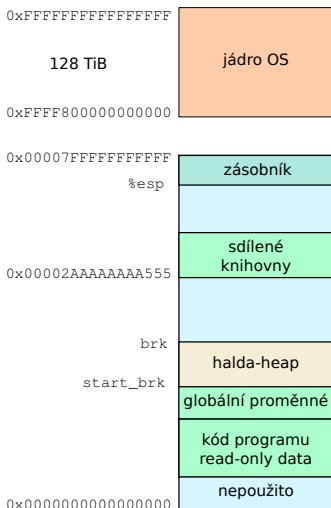
Rozdělení paměti 32-bitový systém Linux

- 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 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 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

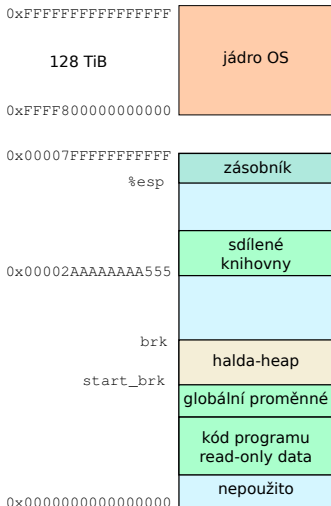
- **Statická velikost, statická alokace**
 - globální proměnné (i static proměnné jednotlivých modulů)
 - linker přiřadí místo pro globální proměnné ve virtuální paměti
 - zavaděč (OS) přiřadí konkrétní místo ve fyzické paměti
- **Statická velikost, dynamická alokace**
 - Lokální proměnné na zásobníku (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 množina 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)

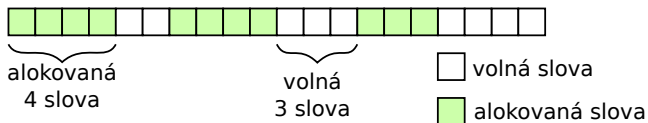
■ Vedlejší cíle

- 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



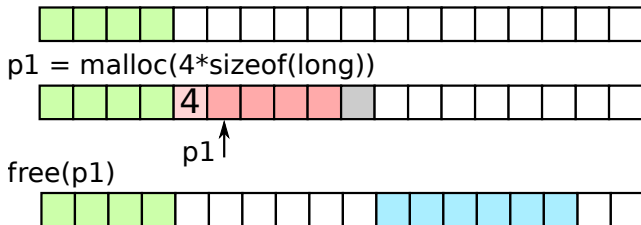
Problémy alokace

- 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

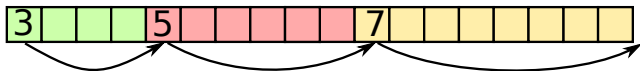
- Informaci o velikosti bloku lze uchovat před začátkem alokovaného bloku
- Číslo před začátkem udává velikost bloku a je označováno jako hlavička
- Informace vyžaduje dodatečné místo při alokaci



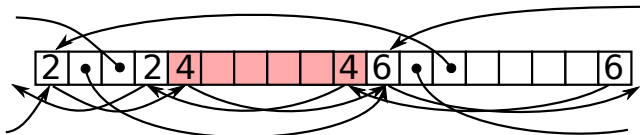
Informace o volných blocích

4 základní metody udržování informace o alokovaných blocích a volném místě

- 1 implicitní seznam s použitím délky



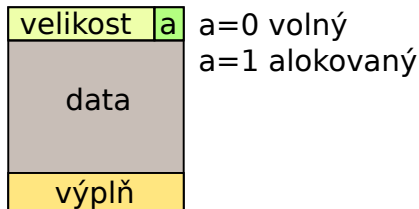
- 2 explicitní seznam volných bloků



- 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.

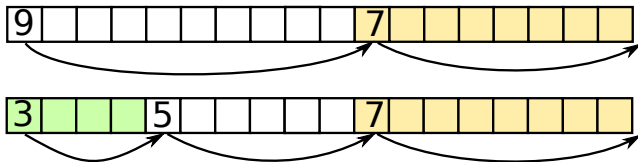


Alokace paměti

- 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

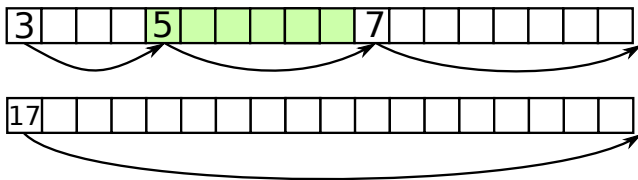
Alokace paměti

- 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

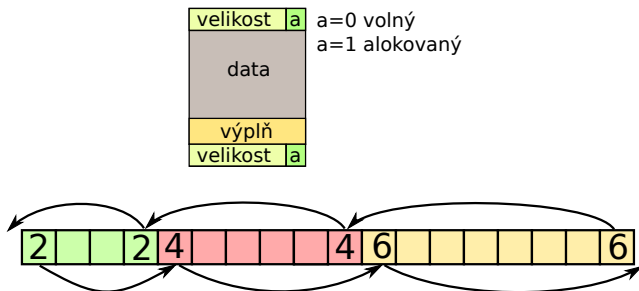
- uvolnění paměti je přepsání bitu "a" na 0 a spojení s předchozím a nebo následujícím volným blokem
- jak najít předcházející blok?
 - buď musíme projít celou haldu a spojovat dopředu
 - nebo musíme něco přidat do naší struktury bloku



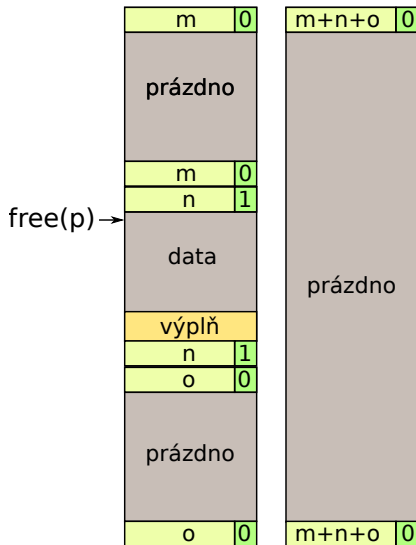
Obousměrný implicitní seznam

Co přidat

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

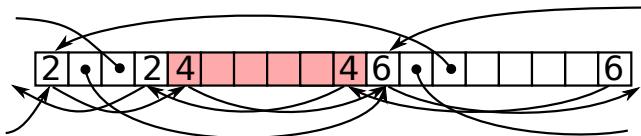


Uvolnění paměti



Explicitní seznam volných bloků

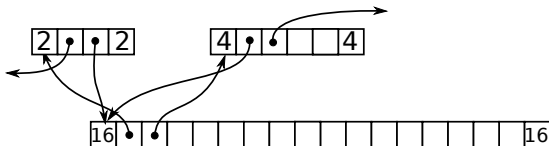
- princip využít volné místo v bloku pro uložení přímých ukazatelů na další a předchozí volný blok
- potřebujeme ponechat velikost bloku (na začátku i na konci), protože je nutná pro spojování uvolněných bloků



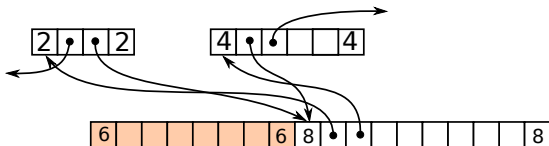
Alokace s ukazateli

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

Před:



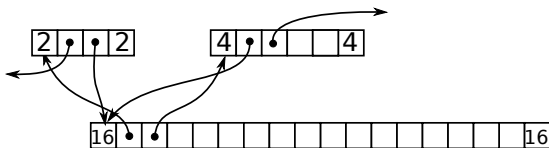
Po:



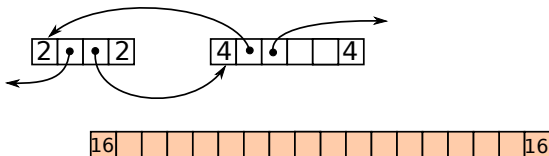
Alokace s ukazateli

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

Před:



Po:



Uvolnění bloku

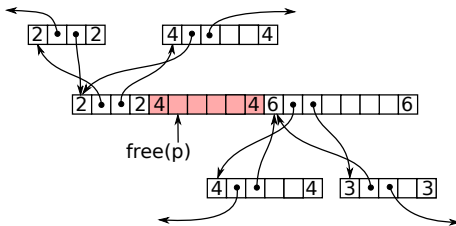
kam zařadit volný blok

- 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

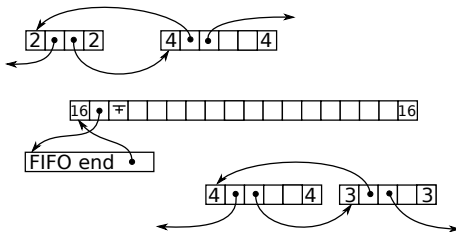
Uvolnění bloku FIFO

- spojování bloků, nutné přepojit ukazatele původních bloků

Před:



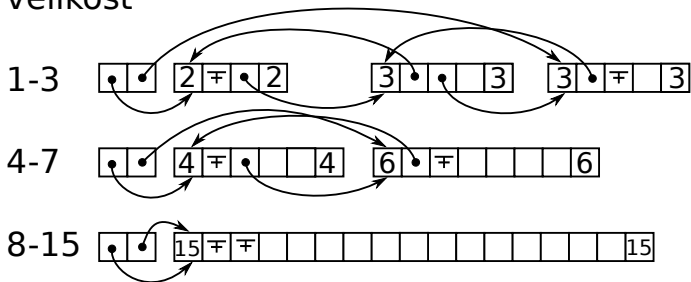
Po:



Oddělené seznamy

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

velikost



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 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 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 periférií 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 perifériemi, hlavně DMA přenosy HDD – RAM (starší periférie 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 stránkových tabulek) se musí měnit podle toho, do jaké paměti je potřeba přistupovat. To hodně zpomaluje běh systému.

Nano úvod do C++

```
class A {
public:
    enum B {Ex, Ey};
    int Ni;
    static int Si;
    A(int z): Ni(z) {}
    int f(int);
    static int Sf(int);
};
```

Znak :: je použit pro definici a pro odkazy na statické prvky třídy A

```
void A::f(int x) {
    return -1*x;}
void A::Sf(int x) {
    return -2*x;}
// globalni definice promenne
int A::Si;
```

Znak :: je použit i při použití vnitřních struktur např. enum B

```
int m = A::Sf(A::Ex);
int n = A::Si;
```

Znak . je použit i pro přístup k prvkům instance třídy:

```
A a(10);
int m = a.f(A::Ey);
int n = a.Ni;
```

kalloc.h a malé použití C++

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;
}
```

Třída Kalloc - použití

```
void *stack = Kalloc::allocator.alloc_page(1,
    Kalloc::FILL_0);
if (!Ptab::insert_mapping (0x1000,
    Kalloc::virt2phys (stack),
    Ptab::PRESENT | Ptab::RW | Ptab::USER)) {
    panic ("Not enough memory\n");
}
```

Přechod virtuální a fyzická adresa

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

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

- 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

- seznam řádu **k** znamená, že udržuje volné bloky o velikosti $2^k \times PAGE_SIZE$
- pokud není volný rámeček 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

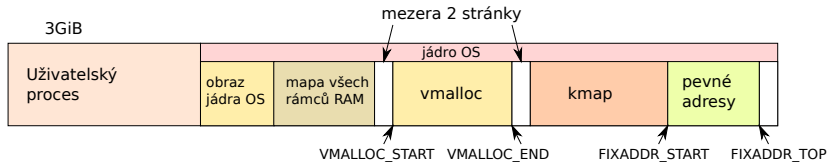
```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    struct page **pprev_hash;
    struct buffer_head * buffers;
#ifdef CONFIG_HIGHMEM || 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 rámeček 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 spojitě 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 HIGHMEM
 - **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