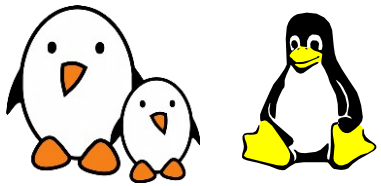
Open Source Programování

http://rtime.felk.cvut.cz/osp/

Pavel Píša
<pisa@fel.cvut.cz>
http://cmp.felk.cvut.cz/~pisa
Michal Sojka
František Vacek
DCE FEL ČVUT



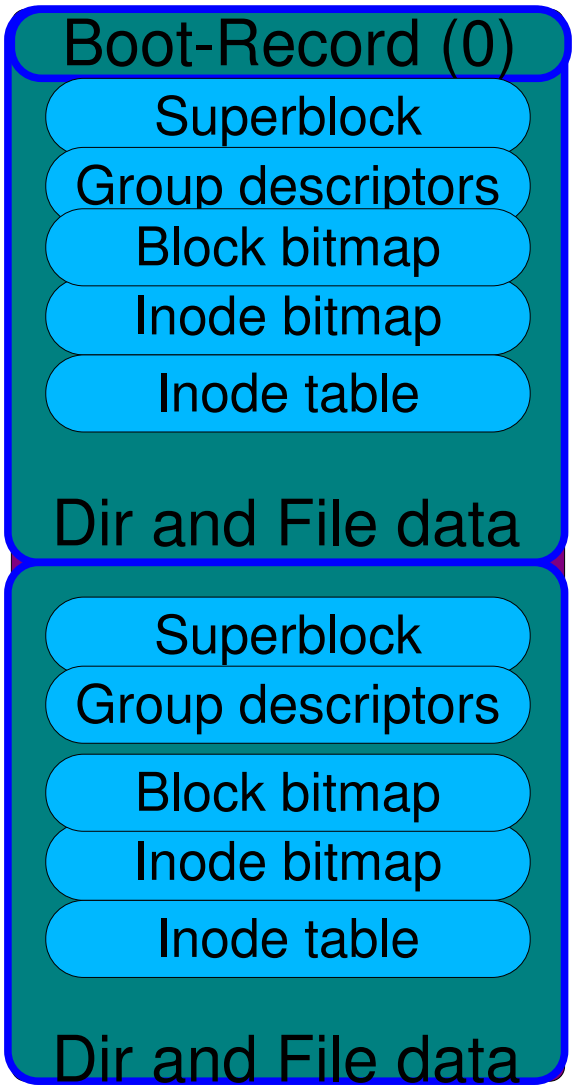
Základní úkoly OS

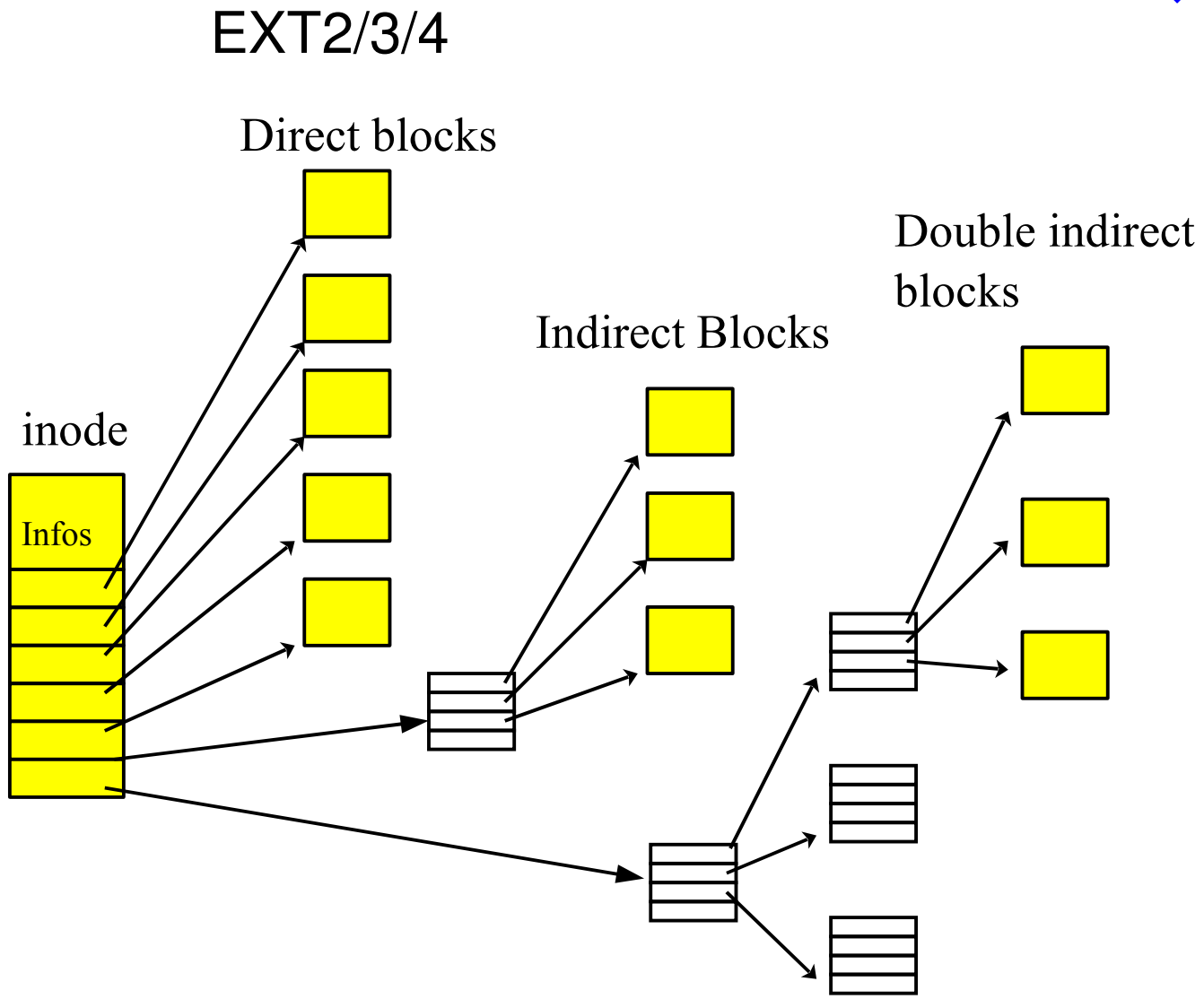
- Jednotné, přenositelné API
- Jednotný přístup k datům, periferiím, komunikaci
- Organizace dat a periferií v (hierarchických) jmenných prostorech
  - Souborový systém (FS) – open, read, write, close
  - Znaková a bloková zařízení většinou přístupná přes globální jmenný prostor (Unix/Linux /dev/*, Windows NT+ \\.\\, …)
  - IPC (FIFO, Unix Socket, Shared MEM, Semaphores), privátní i sdílené, POSIX nevyžaduje pro vše jeden jmenný prostor s FS
  - Síťování a sockety – BSD jiný jmenný prostor ale ostatní operace shodné, Plan 9 vše jednotný prostor, Windows přes IOCTL
- Správa paměti, stránkování – Linux pagecache a swapcache
- Správa procesů, oddělení paměťových prostorů, ochrana paměti a hierarchie oprávnění - capabilities

Základní operace nad „souborem“

Unix/Linux	file_operations	Windows	C library	Java
open,creat O_NONBLOCK	open	CreateFileFILE_FLAG_OVERLAPPED	fopen	java.nio.channels.FileChannel
close	release*	CloseHandle	fclose	Channel.close
read, aio_read	read, aio_read	ReadFile	fread, fprintf	ByteChannel.read
write, aio_read	write, aio_write	WriteFile	fwrite, fscanf	ByteChannel.write
ioctl	ioctl, unlocked_ioctl	DeviceIoControl	ioctl, tcgetattr, ...	No portable way
fsync	flush	FlushFileBuffers	fflush	
select, poll, (epoll, kevent)	poll	WaitForMultipleObjects, select		java.nio.channels.Selector
opendir, fdopendir, readir, closedir	readdir	FindFirstFile, FindNextFile		
fcntl, flockfile, funclockfile	lock	LockFile, UnlockFile		
mmap	mmap	MapViewOfFile		java.nio.channels.FileChannel

Organizace dat souborového systému





Množství dalších FS pro bloková zařízení– FAT, NTFS, BTRLFS, SquashFs, ISO9660 ale i virtuálních – dočasné TmpFS nebo zpráva systému sysfs, proc, cgroup, debugfs, usbfs, fuse

Průběh volání open

- Na vyšší úrovni Java nebo `fopen("/home/franta/dopis.txt","w+")`
- Po zpracování/alokaci stavových informací na vyšších úrovních dojde k volání `open("/home/franta/dopis.txt", O_RDWR|O_CREAT, S_IRWXU)`
- Volání dále prochází (G)LIBC/CRT *glibc/sysdeps/unix/sysv/linux/open64.c*
  `INLINE_SYSCALL (open, 3, file, oflag | O_LARGEFILE, mode);`
- Makro nahradí mnemonické určení služby „open“ za číslo systémového volání a předá řízení jádru OS. Číslo je nadefinováno v souboru *linux-2.6.x/include/asm-generic/unistd.h*
  `#define __NR_open 1024`
  `__SYSCALL(__NR_open, sys_open)`
- Makro `INTERNAL_SYSCALL` je již architekturně závislé, pro 32-bit Intel x86 systém je definováno v souboru *glibc/sysdeps/unix/sysv/linux/i386/sysdep.h*
  `INTERNAL_SYSCALL(name, err, nr, args...)`
  `… LOADARGS_##nr; movl %1, %%eax; int $0x80; RESTOREARGS_##nr`
- Protože je volání přes IDT na x86 velmi neefektivní, preferuje se **SYSENTER**
- CPU přejde do systémového režimu (x86 Ring 3 → Ring 0) a volá vstupní bod jádra **system_call** v *linux-2.6.x/arch/x86/kernel/entry_32.S*, v této fukci je podle čísla systémového volání (předáno v EAX na x86) nalezena položka v tabulce **sys_call_table**

Implementace služby open

- **open** náleží do skupiny **souborových služeb** → implementace v adresáři *fs linux-2.6.32/fs/open.c*

```
SYSCALL_DEFINE3(open, const char __user *, filename, int,
flags, int, mode)
    …; ret = do_sys_open(AT_FDCWD, filename, flags, mode);
```

- Protože není dopředu jasné, na jakém disku a souborovém (nebo virtuálním) filesystému se dané adresáře nacházejí, jsou veškeré souborové služby nejdříve zpracovávané obecně - **VFS** (virtual filesystem). Když je cesta nalezena ve vyhledávacích strukturách **dcache** (d=directory), je již znám filesystém. Je zkontrolovaná existence/vytvoření souboru, založena stavová struktura se stavem k jeko konkrétnímu otevření (`struct file`) a je zavolaná „metoda“ daného filesystému *linux-2.6.x/fs/ext3/file.c:*

```
struct file_operations ext3_file_operations = { …, .open =
generic_file_open, ...};
```

- Protože specializace FS není v tomto případě potřeba je zavolaná obecná implementace
- Skutečná specializace většiny metod běžných filesystémů je totiž řešena až na úrovni souborových uzlů inode `static const struct address_space_operations ext3_{ordered|writeback|journalled}_aops`

Vlastní čtení a zápis dat

- U znakových zařízení a virtuálních filesystémů dané „metodami“ `read` a `write` v `struct file_operations`
- U běžných souborů na disku je potřeba zajistit **paralelní přístup** přes **více otevření** souboru k jeho datům, která musí být nejdříve načtena někam do paměti. Zároveň paralelně běžící přístupy musí výsledek vidět tak, jako by přímo po bytech měnily data na disku
- Úplně přímý **přístup** na disk **po bytech není možný**, i načtení, úprava a uložení bloku je extrémně pomalé, nejvíce času zabírá vystavování hlaviček. Je potřeba spravovat vyrovnávací paměť, přitom musí být zajištěné, že v paměti ke každým datům existuje jen jedna kopie a její modifikace mohou probíhat v podstatě paralelně a změny musí být okamžitě vidět pro ostatní přístupy
- Existují dvě logické možnosti, kam vložit vyrovnávací paměti – diskové buffery (buffer heads mezi diskem a FS) a **pagecache** mezi FS a aplikacemi
- Původně Linux používal především diskové buffery, vzhledem k požadavku na možnost mapování souborů do paměťových prostorů (mmap) se data mohla objevit ve dvou kopiích nebo se složitě po mmapu upravovaly buffer heads tak aby směřovaly do stránek. Postupem času se zcela přešlo na **pagecache**

Pagecache - stránková vyr. paměť

- Data nejsou v paměti udržovaná podle ~~**pozice na disku**~~, ale podle **příslušnosti a offsetu** v daném **souboru** (**inode**). Přitom jsou offsety a data zarovnány tak aby odpovídaly násobkům velikosti **paměťové stránky** (4kb na x86).
- Protože buffer heads jsou používané opravdu jen pro přenosy dat z a na disk, musí být i data adresářů a množství dalších rozšiřujících informací filesystémem udržované v pagecache v interních stránkových sadách, které příslušejí buď k adresáři nebo **superbloku** FS (globálnímu stavu filesystému)
- Na první pohled to vypadá nevýhodně, ale umožňuje to úplnou koherenci/**konsistenci** mezi daty čtenými a modifikovanými voláními **read a write** a stránkami **mapovanými do adresních prostorů** (mm context) procesů. Téměř veškerá paměť může podle potřeby sloužit jako cache disků/souborů a užití se přizpůsobuje zátěži.
- Práce souborových systémů a základních operací je silně provázaná se zprávou paměti

Správa paměti

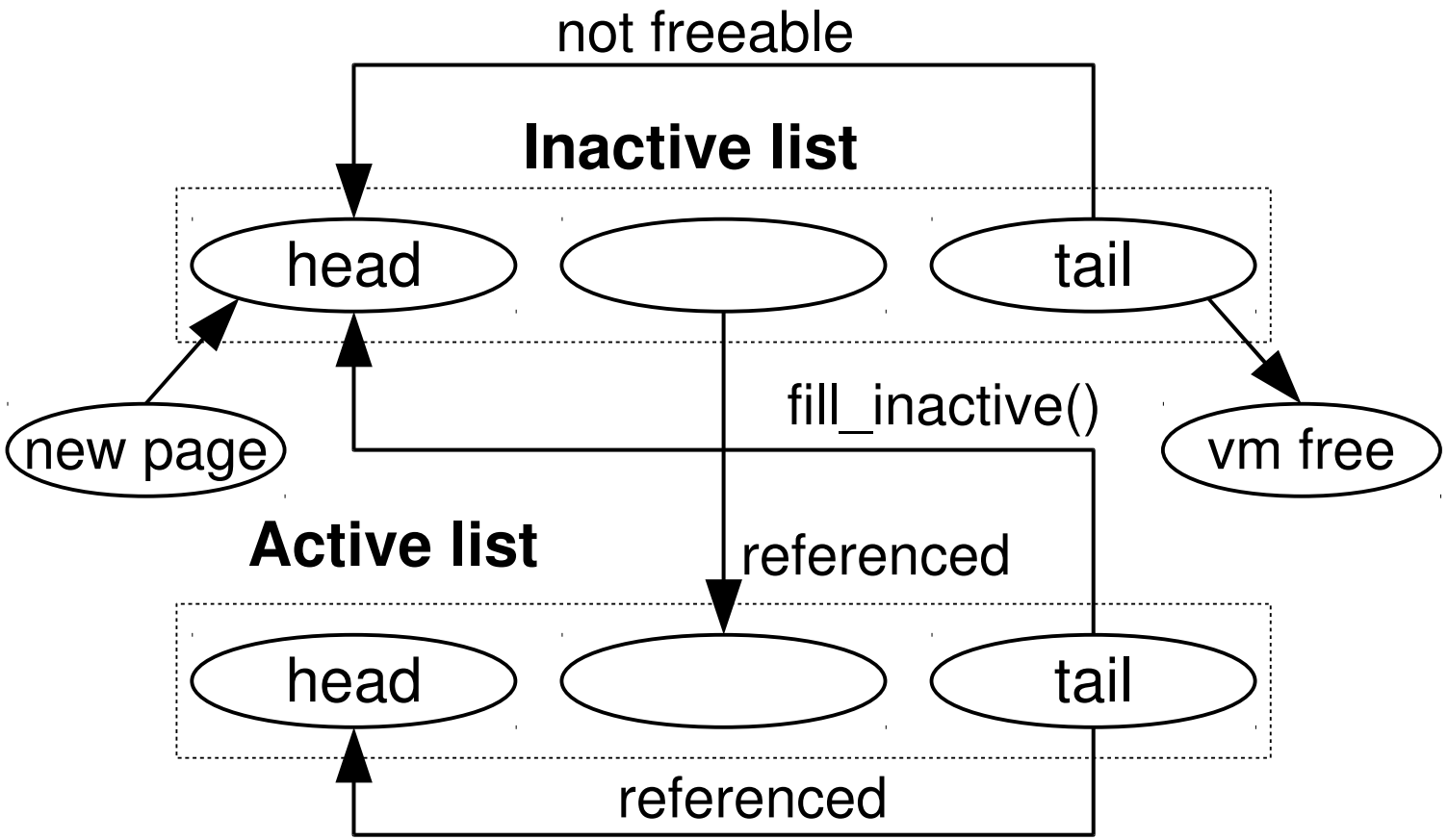
- Informace o stavu obsazení fyzické paměti a odkládacích (Swap) oddílů a souborů
- Alokace volných stránek pro účely jádra a všech ostatních subsystémů
- V Linuxu slouží v podstatě většina fyzické paměti jako
  - stránková/pagecache pro zpřístupnění a mapování souborů, které se nalézají na souborových systémech, FS pak vlastně slouží k rozdělení kapacity blokových zařízení mezi soubory
  - pro anonymní privátní a sdílené stránky, které se odkládají na swap zařízení – obecně swapcache
- Připomenutí hierarchie paměti:

  zápisníková paměť – registry, běžná transparentní L1 až Lx cache, hlavní/fyzická paměť, soubory a data na disku

Kritické požadavky na zprávu pam.

- Žádný úsek (stránky) spustitelného souboru/programu ani sdílených knihoven (.so, DLL) není „nikdy“ načten do paměti pokud ho není potřeba – do paměti je pouze mapován
- I pokud je stejná stránka potřeba ve více paměťových prostorech, je ve skutečnosti mapována do všech prostorů stejná stránka fyzické paměti nebo je ve všech prostorech položka stránkovacích tabulek invalidována a zpracování výpadku zajistí její natažení ze souboru (inode, backing store)
- Celkový součet paměťových požadavků může řádově přesahovat množství fyzické paměti. Nepoužívané (LRU) stránky, které odpovídají obsahu backing store jsou vyřazeny, modifikované sdílené stránky zapsány zpět a modifikované anonymní nebo privátně mapované stránky jsou odkládány na swap úložiště
- Pro výkonnost systému je kritický dobře navržený algoritmus pro výměnu stránek – nahrání stránek které jsou potřeba (způsobí výjimku nebo jsou odhadnuté pro readahead), které vyžaduje zajištění volného místa realizované vyhledáním a vyjmutím těch fyzických stránek, která již nejsou (s největší pravděpodobností) potřeba
- Díky sdílení stránek mezi paměťovými kontexty je potřeba nalézt vhodný algoritmus a invalidaci všech odpovídajících položek v stránkových tabulkách

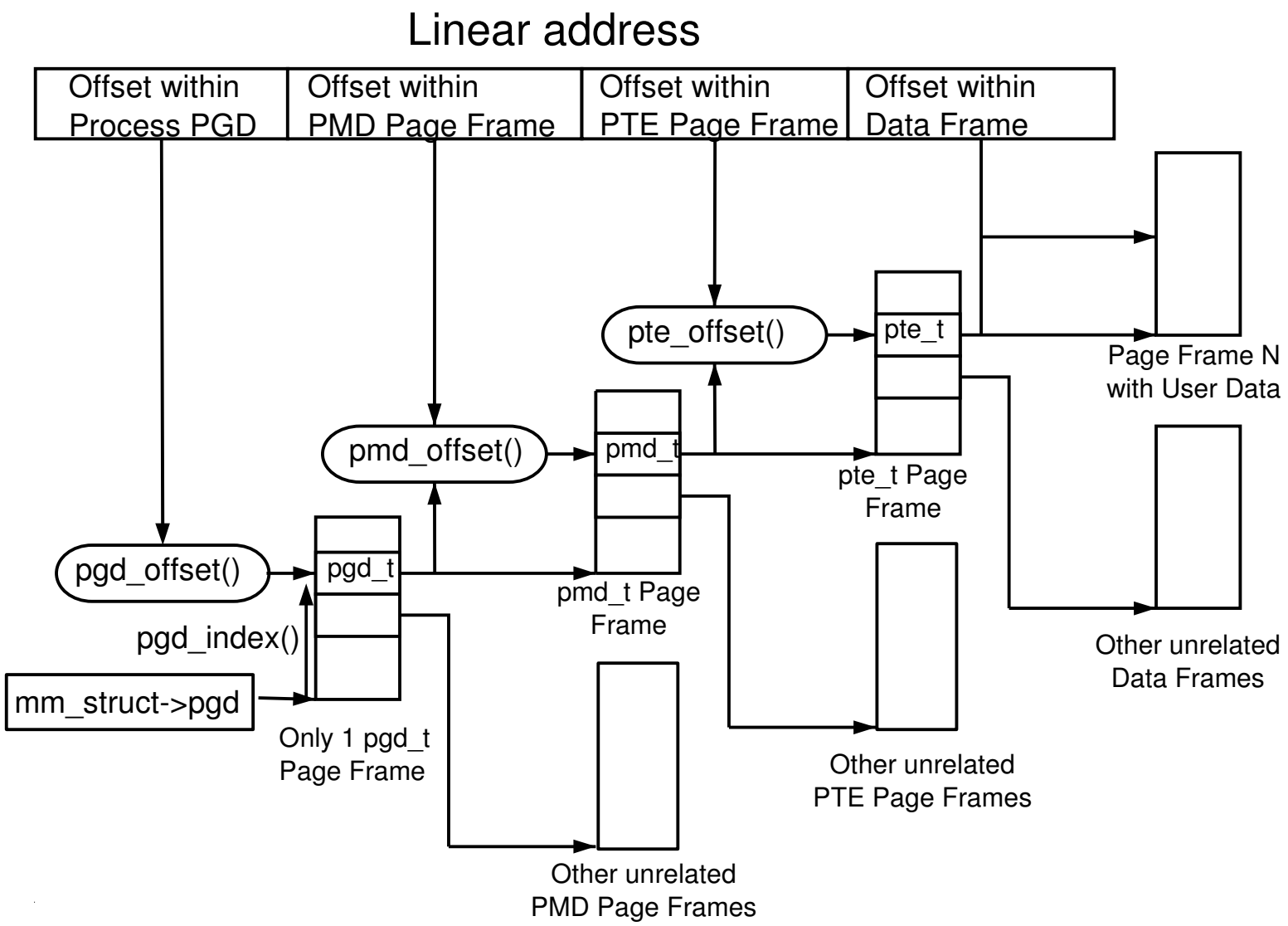
Správa běžných stránek – Linux 2.6.x



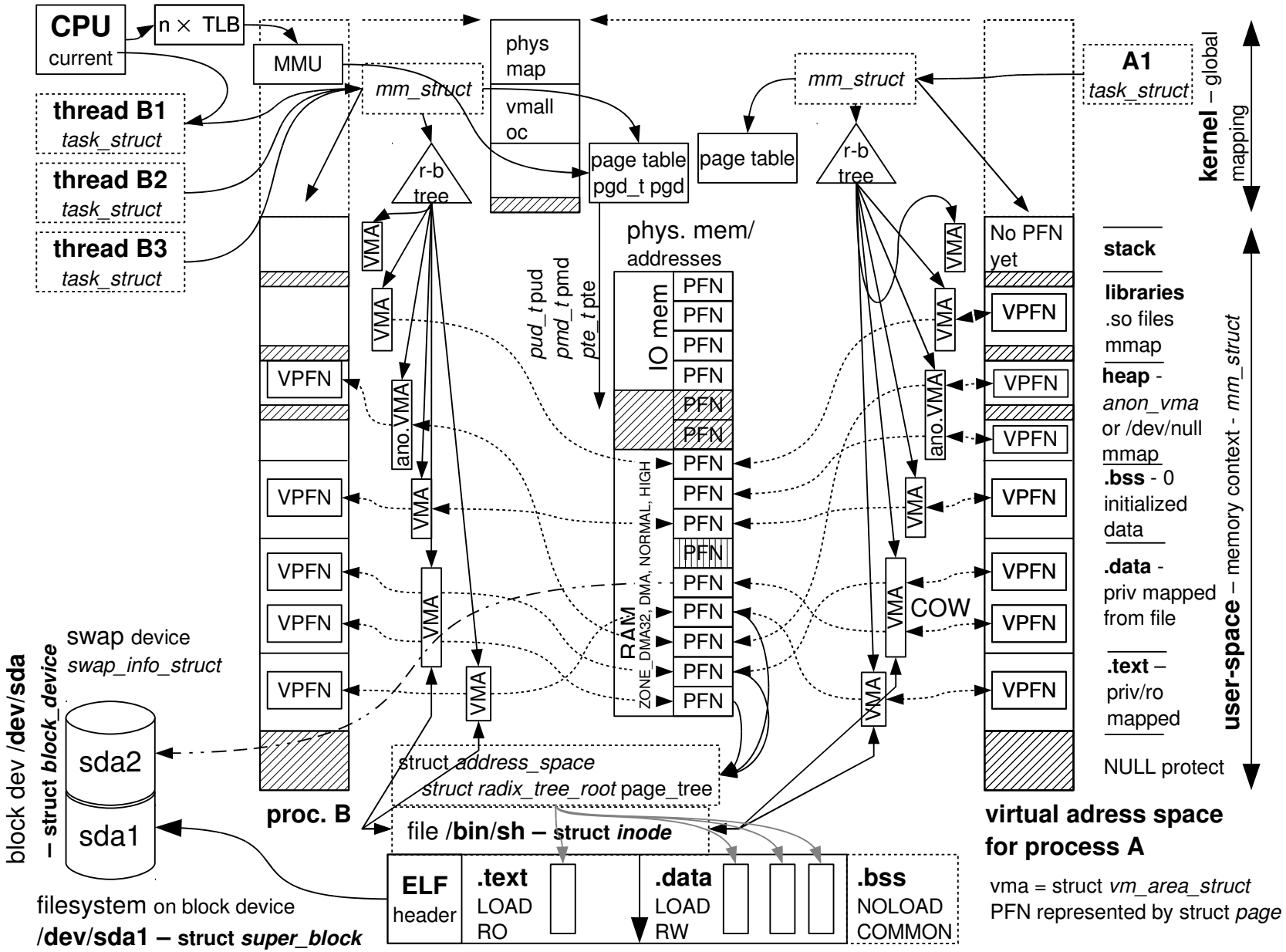
klíčovým bodem je ve spodní úrovni funkce __get_free_pages()

paměť je z důvodů omezenosti některých architektur a sběrnic dělena na zóny – GFP_DMA, GFP_DMA32, __GFP_HIGHMEM, NORMAL, způsob plnění požadavku – GFP_NOWAIT, GFP_ATOMIC, GFP_NOIO, GFP_NOFS, GFP_KERNEL, GFP_TEMPORAR

Na úrovni CPU

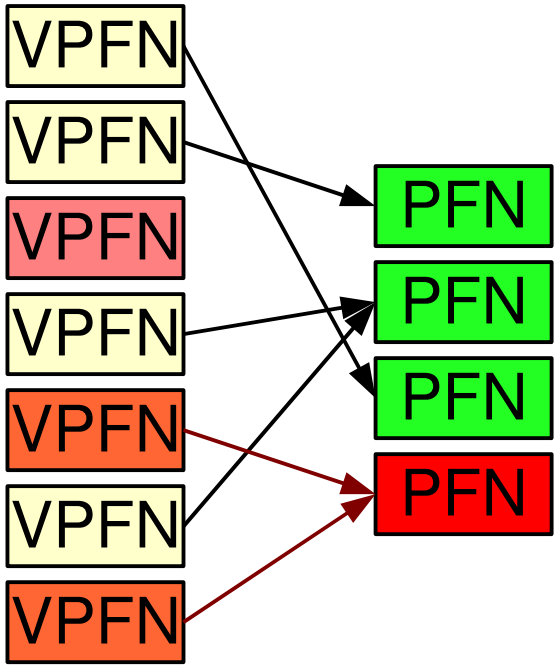


Správa virtuální paměti



Hlavní komplikace - reverse mapping

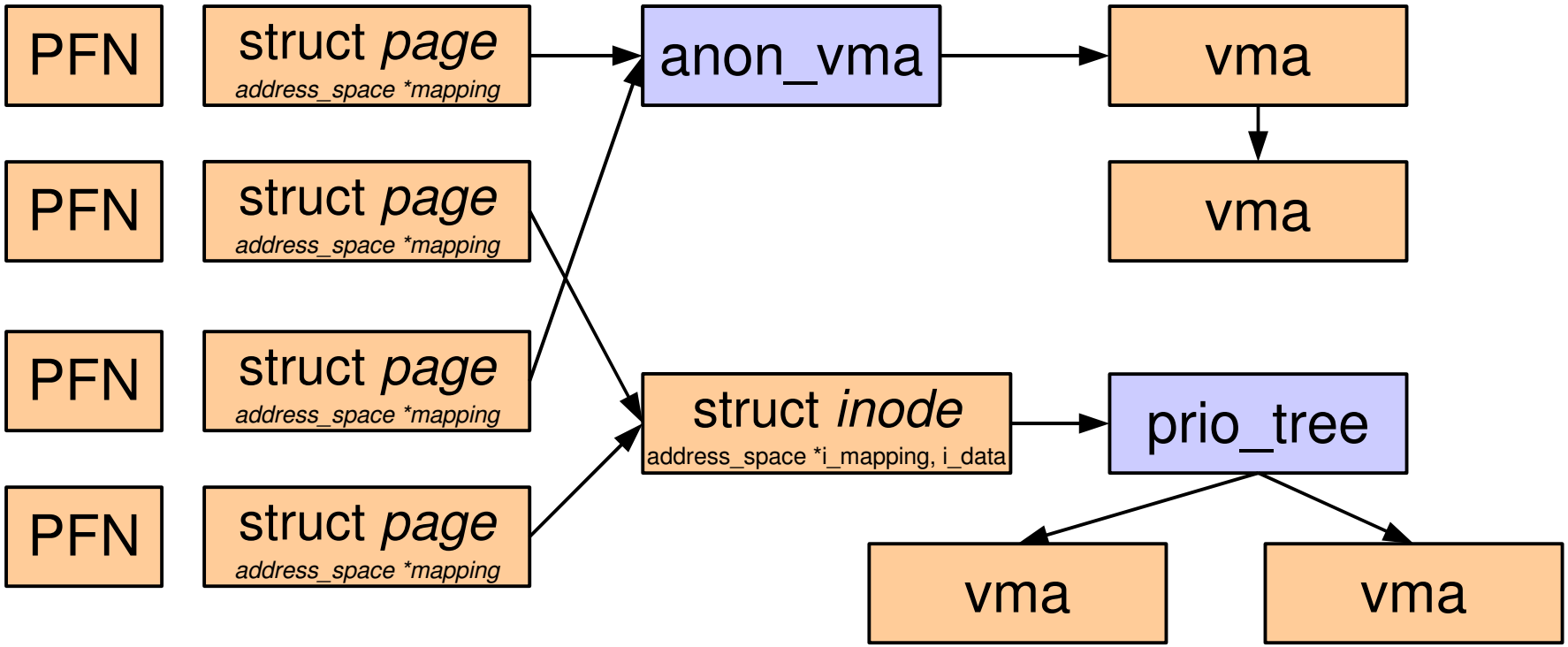
- Virtutální stránky (VPFN) z více a dokonce i z jednoho paměťového kontextu mohou odkazovat na jednu fyzickou stránku (PFN)
- Stránky virtuální paměti stojí pouze položku (PTE) ve stránkových tabulkách (4/8 byte) + něco málo ve vyšších úrovních tabulek (PGD, PUD, PMD) + popis rozsahu (*vm_area_struct*)
- Fyzické stránky jsou kritickým zdrojem, každá je popsaná svojí struct *page* a referencovaná svým umístěním page-frame-number
  PFN = virtual_address >> PAGE_SHIFT



- Těžká operace je nalezení všech PTE k danému PFN
- Pokud by se seznam držel ke každé stránce, tak při 8 byte na položku seznamu, přitom na x86 32-bit je k dispozici je ~900M lowmemory
- Při 1000 mapujících 2G sdílené (shared) paměti by seznamy vyšly na 1000*2*1024*1024*1024/4096*8/1024/1024/1024 = 3.9 GiB

Reverzní mapování přes objekty

- Rešení: objrmap+anon_vma+prio_tree



- Dohledání konkrétních PTE položek v stránkových tabulkách je již formalita, pokud známe VMA a tím i *mm_struct*

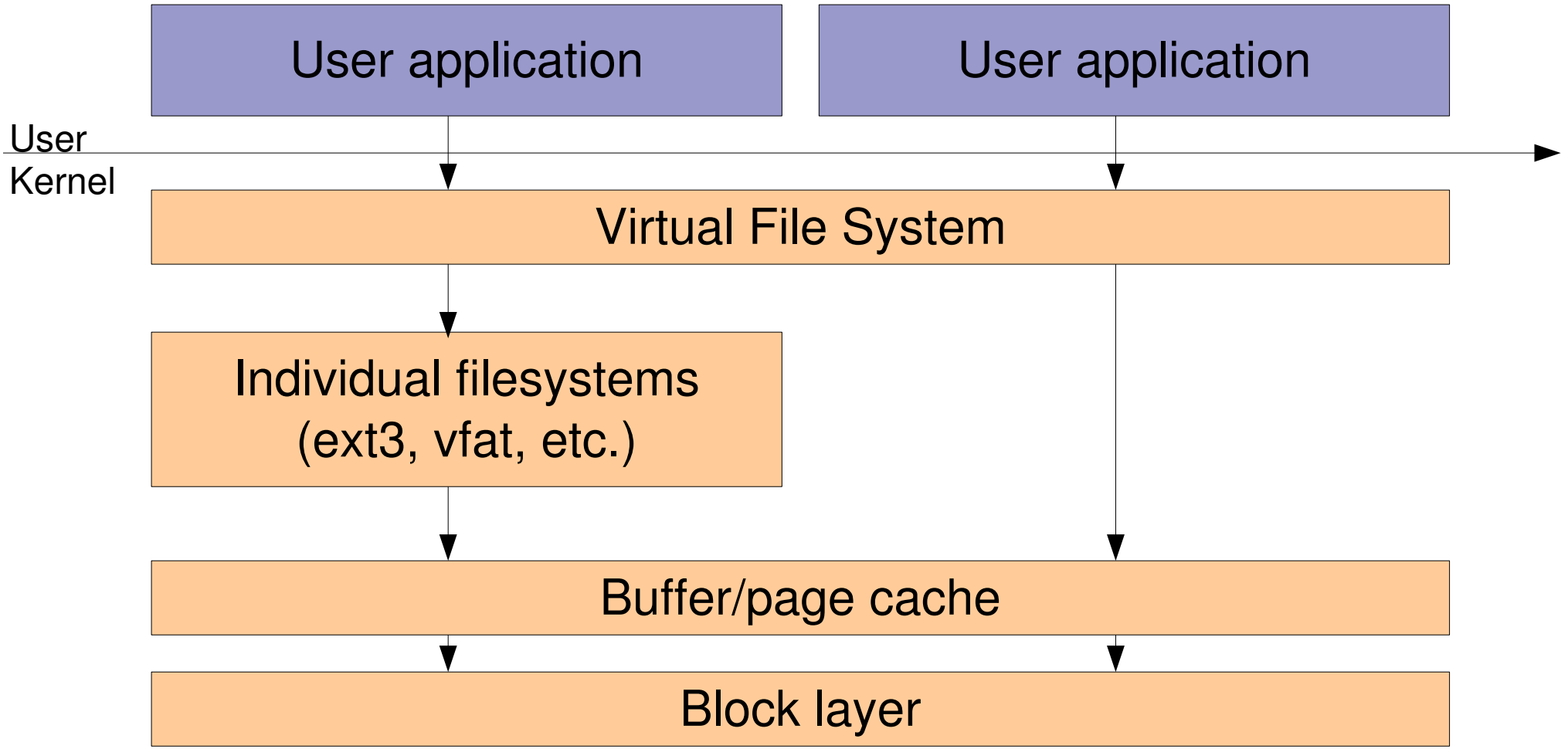
Datové struktury

```
struct vm_area_struct {
     struct mm_struct * vm_mm;      /* The address space we belong to. */
     unsigned long vm_start;        /* Our start address within vm_mm. */
     unsigned long vm_end;          /* The first byte after our end address
                              within vm_mm. */

     /* linked list of VM areas per task, sorted by address */
     struct vm_area_struct *vm_next;

     pgprot_t vm_page_prot;         /* Access permissions of this VMA. */
     unsigned long vm_flags;        /* Flags, see mm.h. */

     struct rb_node vm_rb;

     union {
          struct {
               struct list_head list;
               void *parent;  /* aligns with prio_tree_node parent */
               struct vm_area_struct *head;
          } vm_set;

          struct raw_prio_tree_node prio_tree_node;
     } shared;

     struct list_head anon_vma_node;     /* Serialized by anon_vma->lock */
     struct anon_vma *anon_vma;     /* Serialized by page_table_lock */

     /* Function pointers to deal with this struct. */
     const struct vm_operations_struct *vm_ops;

     /* Information about our backing store: */
     unsigned long vm_pgoff;        /* Offset (within vm_file) in PAGE_SIZE
                                       units, *not* PAGE_CACHE_SIZE */
     struct file * vm_file;         /* File we map to (can be NULL). */
     void * vm_private_data;        /* was vm_pte (shared mem) */
     unsigned long vm_truncate_count;/* truncate_count or restart_addr */
};
```

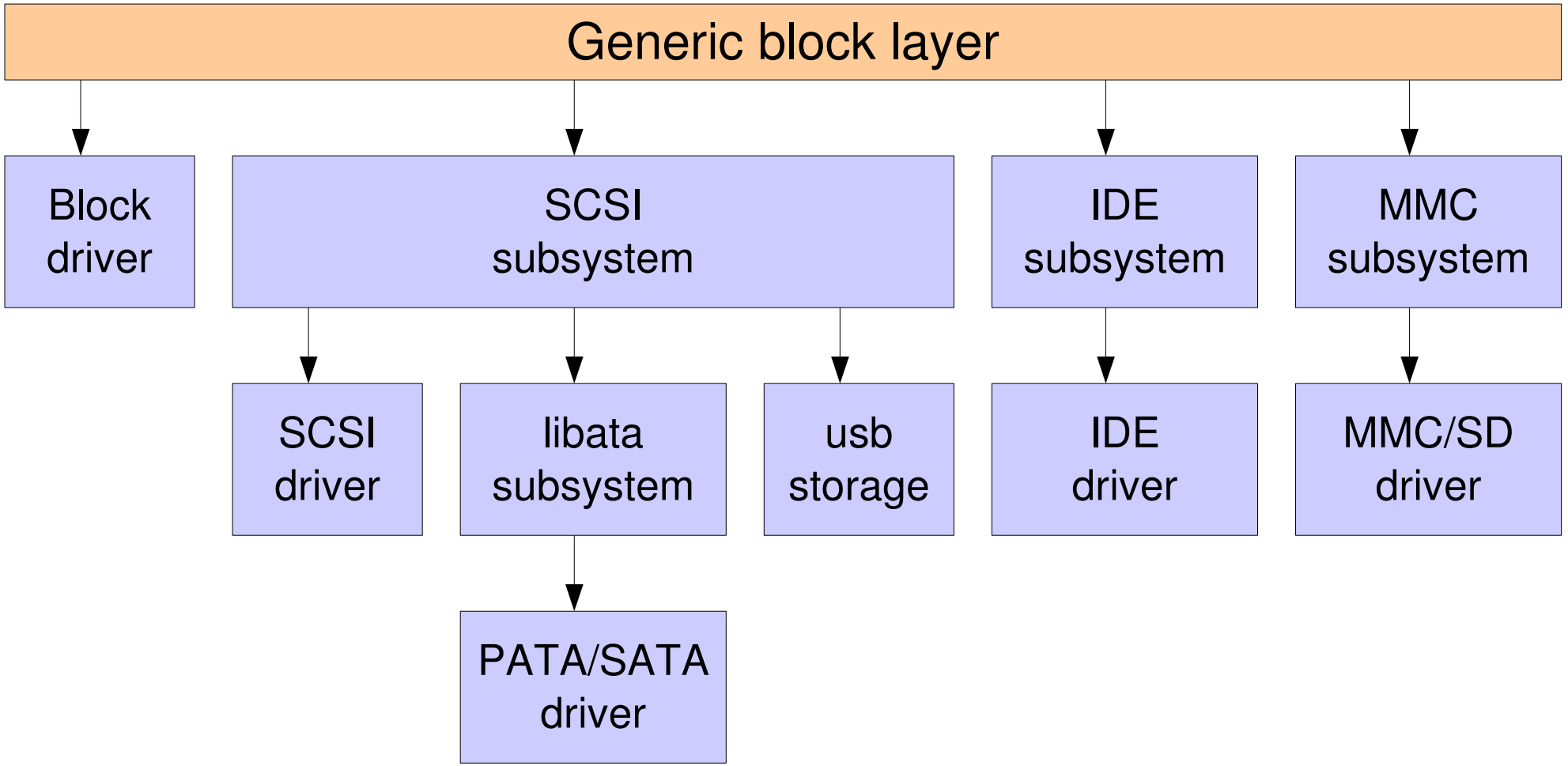
A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma list, after a COW of one of the file pages. A MAP_SHARED vma can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack or brk vma (with NULL file) can only be in an anon_vma list.

For areas with an address space and backing store, linkage into the address_space->i_mmap prio tree, or linkage to the list of like vmas hanging off its node, or linkage of vma in the address_space-> i_mmap_nonlinear list.

FS a bloková zařízení



Různé subsystémy



Datové struktury