13. Virtuální paměť v praxi

1. Princip lokality

Architektura paměti se projevuje na několika úrovních:
adresování – TLB (D-TLB, C-TLB), tabulka stránek
plnění – cache L1 a L2 (jádra), cache L3 (CPU), lokální paměť, vzdálená paměť (NUMA), virtuální paměť

Dosažení plného výkonu procesoru vyžaduje dodržení principu lokality při práci s daty:

- často používané proměnné musí být alokovány blízko sebe (aby byly pokud možno v jednom řádku cache = 64/128 byte, v jedné stránce)
- málo používané proměnné by měly vzdáleny alespoň o délku řádku cache od často používaných (není problém ve strukturách a třídách, ale je to na programátorovi)
- při práci s velkými datovými strukturami je třeba pracovat vždy s co největším souvislým úsekem dat, neskákat po jednotlivých slabikách po paměti (přístup třeba k jedné slabice znamená načtení celého řádku cache!)

Pro paralelní programy je třeba dát pozor na **falešné sdílení**:

```
static int i, j;    /* nejspíš bude ve stejném řádku cache */
for (i=0; i<N; i++){          for (j=0; j<n; j++) {
    paralelní výpočet                 paralelní výpočet
}                             }
```

Jak rychle poběží?
Překvapení: paralelně může běžet výrazně pomaleji, než sekvenčně! Každá inkrementace *i*,*j* znamená synchronizaci sdíleného řádku cache L1,L2 mezi procesory – tedy efektivně zamykání! Může nastat také při alokaci dynamických datových struktur.

Operační systém může ovlivnit efektivitu až od úrovně nahrazovacího algoritmu, resp. plnění TLB (tam kde to dělá jádro OS), nižší úrovně hierarchie paměti má ve své moci programátor (a kompilátor).

Jádro systému poměrně obtížně odhaduje chování programu, je schopno detekovat poslední použití stránek, případně chování programu (náhodný přístup - FreeBSD prefetch 7+8 okolních stránek, sekvenční - prefetch až 15 následujících). Nemůže ale znát budoucí chování – musí pomoct programátor!

BSD systémy – volání jádra *madvise()*

POSIX 1003.1-2001 zavádí volitelně *posix_madvise():*

```
int posix_madvise(void *addr, size_t len, int
advise);
```

Indikace jak bude program přistupovat k adresám od *addr* v délce *len*:

POSIX_MADV_NORMAL	bez indikace – heuristika pro detekci sekvenčního přístupu
POSIX_MADV_SEQUENTIAL	sekvenčně – má smysl dopředné čtení, nižší adresy se mohou odkládat dříve
POSIX_MADV_RANDOM	náhodně – nemá smysl dopředné čtení
POSIX_MADV_WILLNEED	data budou použita – stránky by neměly být odkládány
POSIX_MADV_DONTNEED	data nebudou znovupoužita – stránky není třeba držet v paměti

2. Paměťové nároky jádra

- Jádro systému vyžaduje dynamickou alokaci paměti.
- Paměť je často alokována při V/V – nelze čekat, až se uvolní nahrazovacím algoritmem.
- Velikost paměti pro stránkování × paměť pro jádro.

Musí být zajištěna rezerva volné paměti – odkládání stránek je aktivováno už při nedostatku stránek, ne až při výpadku stránky.

Velikost paměti pro stránkování x V/V vyrovnávací paměti:

- pevně stanovený počet vyrovnávacích pamětí nemusí využít paměť při malém počtu procesů,
- proměnný počet – minimum pevné, aktuální počet podle počtu volných stránek,
- integrovaná stránkovací a vyrovnávací paměť pro V/V.

Příklad FreeBSD (Linux obdobně):
Seznam zamčených stránek, aktivních, neaktivních a volných.
Při nedostatku volných stránek spuštěn stránkovací démon *vm_pageout*, ten projde seznam neaktivních:

- stránky s nastaveným příznakem použití přesune na konec aktivních,
- stránky s nulovým příznakem použití uvolní a přesune na konec seznamu volných.

Pokud je málo neaktivních (méně než 30% aktivních), projde seznam aktivních:

- stránky s nastaveným příznakem použití přesune na konec seznamu a vynuluje příznak použití,
- stránky s nulovým příznakem použití přesune do neaktivních.

Při výpadku stránky se prohledávají (hash) všechny seznamy, pokud zůstala stránka v paměti, nemusí se zavádět.

3. Zamykání stránek

- stránka, která obsahuje data pro přímý V/V, musí být po dobu operace trvale v paměti (DMA) – zajišťuje jádro,
- bezpečnost - citlivá data by se neměla odkládat na disk (nahrazovacím algoritmem), nebezpečí prozrazení,
- systémy reálného času – kritická doba reakce nesmí být ovlivněna stránkováním – zamykání procesů, stránek v paměti, musí si zajistit programátor.

POSIX 1003.1b:

```
#include <sys/mman.h>
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

Použití omezeno na privilegované procesy, může snadno vést k uváznutí (viz přidělování prostředků). Neprivilegovaným procesům je dnes povoleno zamykat malý rozsah stránek (cca. 64 KB) pro práci s citlivými daty (privátní klíče, hesla v otevřené podobě, apod.).

4. Spouštění procesů

```
if ((id = fork()) == 0) {  /* dětský proces */
    execl("/prog", ARG0, ARG1, ...);
} ...
```

- sémantika *fork()* – musí vzniknout kopie procesu = kopie obsahu adresového prostoru do vytvářeného procesu,
- pokud následuje *exec()*, přepíše se adresový prostor kódem a daty spuštěného programu, kopie ve *fork()* je zbytečná!

Řešení:
BSD 4.3 - *vfork()*
Dětský proces používá (sdílí) adresový prostor rodiče, rodič je pozastaven do té doby, než potomek provede *exec()* nebo skončí.

Omezení: vzniklý proces nesmí změnit obsah sdíleného adresového prostoru (např. návrat z funkce volající *vfork()* by mohl poškodit obsah zásobníku, volání knihovny std. C změnit statická data v knihovnách, apod.). Není definováno chování ve vícevláknovém programu (obvykle skončí špatně)!

Využití stránkování – Copy-On-Write (COW)
- oba procesy mohou sdílet počátečně stejně naplněnou tabulku stránek,
- zakáže se zápis do všech platných stránek,
- první pokus o zápis způsobí výpadek paměti – zjistí se, že se jedná o legální zápis do oblasti COW, vytvoří se kopie stránky a povolí zápis do originálu a kopie v obou procesech.

Výhody:
- data se kopírují až v okamžiku, kdy je třeba,
- lze sdílet i celé tabulky stránek (u víceúrovňových).

5. Přístup k souborům virtuální pamětí

Klasický přístup k souborům:

```
    lseek(fd, offset, SEEK_SET);
    read(fd, buf, n);
```

→ jádro, čtení bloku - *bread()*, kopie z AP jádra do AP procesu

Myšlenka: zamapovat soubor do adresového prostoru procesu a pro přístup použít virtuální paměť:

```
fd = open("data", O_RDWR);
stat(fd, &st);          /* informace o souboru */
m = mmap(0, st.st_size, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, (off_t)0);
close(fd);
...
memcpy(kam, m+offset, n);    /* čtení souboru */
memcpy(m+offset, co, n);     /* zápis souboru */
```

MAP_SHARED – změny jsou ukládány zpět do soboru,
MAP_PRIVATE – změny se neukládají zpět do souboru (při nahrazování stránek se použije normální swap).

Windows/NT – otevřít soubor pomocí *CreateFile()* a namapovat:

```
m = MapViewOfFile(fmh, FILE_MAP_WRITE, ofset_hi,
    ofset_lo, length);
```

Výhody:

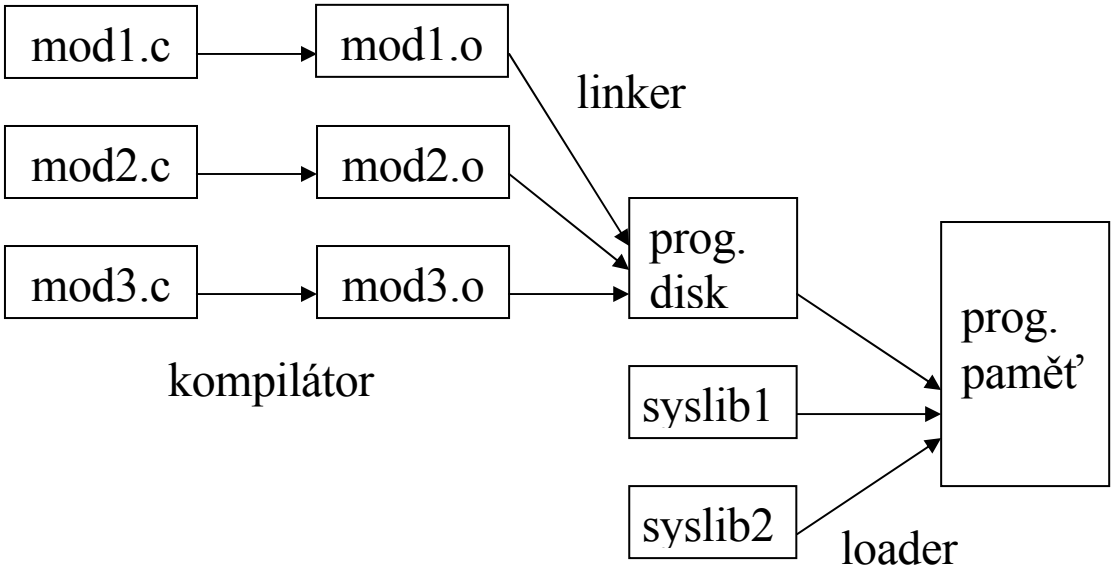
- Rychlejší přístup k datům, může odpadnout kopie přes systémové vyrovnávací paměti (záleží na implementaci V/V).
- Pracovní množina procesu tvoří zároveň vyrovnávací paměť (pokud je dostatek volné paměti, může být celý soubor trvale v paměti).
- Bezprostřední sdílení změn dat ve více procesech (pokud mapují stejný soubor, sdílí stejné fyzické stránky, do kterých jsou zavedeny části souboru).

6. Sdílení stránek

implicitní – sdílení kódu procesů, sdílených knihoven
explicitní – zajišťuje si programátor sám

a) Sdílení kódu

Problém relokace kódu před spuštěním:



Možnosti relokace (address binding):
- při překladu (asembler na pevné logické adresy)
- při sestavení (statická relokace na pevnou logickou adresu)
- při spuštění (dynamická relokace)

Možnosti sestavování (vyřešení neuspokojených externích referencí):
- při sestavení (staticky)
- při spuštění (odloženo sestavení do spuštění)

Sdílení kódu: kód musí být konstatní a přístupný pouze pro čtení – reentrantní (nesmí být při spuštění a za běhu modifikován).

Sdílení kódu procesů – procesy řízené stejně relokovaným programem

- vliv na stránkování – vícenásobné použití stránek
- datové struktury – správa sdílených částí LAP

Sdílení kódu sdílených knihoven:

- knihovny staticky relokované na konstantní adresu (SYSV3)
- dynamické knihovny – lze zavést na libovolnou adresu, kód musí být dynamicky přemístitelný (relokovatelný), v jednom procesu může být zobrazen na adrese X, v jiném na adrese Y.

Problém odkazů na externí vstupní body – jejich adresa není dopředu známá, statické adresy se musí relokovat až při použití knihovny (nutné taky dynamické sestavení).
U dynamických knihoven přibývá problém adresování kódu a dat v rámci knihovny (adresy jsou závislé na umístění knihovny v LAP).
Unix System V Relase 4 (SVR4) – formát ELF (Extended Link Format), nutná podpora překladačů – PIC (Position Independent Code) a sestavovacích programů:

```
gcc –c –fpic mod1.c mod2. mod3.c
ld –shared –o lib.so mod1.o mod2.o mod3.o
```

Program s dynamickými knihovnami:

```
ldd /usr/bin/vi
  libcurses.so.2 => /usr/lib/libcurses.so.2
  libtermcap.so.2 => /usr/lib/libtermcap.so.2
  libc.so.3 => /usr/lib/libc.so.3
```

Zavedení a relokace dynamických knihoven – při startu programu, dynamický sestavovací program je buď součástí *libc*, nebo je automaticky namapován (*mmap*) při spuštění programu.

Dynamická knihovna obsahuje:

- adresy, které je třeba relokovat podle místa zobrazení knihovny v procesu (Global Offset Table = GOT),
- seznam vstupních bodů (pro použití z hl. programu a jiných knihoven), všechny externí odkazy do dynamické knihovny musí jít nepřímo (nepřímý skok, volání přes PLT),
- seznam externích odkazů z knihovny (na závislé knihovny),
- seznam závislých knihoven (řetězové závislosti).

Výhody sdílených knihoven:

- redukce velikosti programů uložených na disku,
- sdílení kódu na úrovni knihoven (častější než u sdílení kódu celých programů),
- možnost oprav knihoven beze změny programu (verzování knihoven).

Příklad:

```
int v;                  /* static, abs. adresa */
extern int e, g(int p);   /* externí adresa */
int f(int a)
{
        return g(a) + v + e;
}
```

gcc –S –fpic –c –O elf.c

```
f:  pushl %ebp
    movl %esp,%ebp # ebp = adresa rámce zásobníku
    pushl %ebx
    call .L3
.L3:popl %ebx      # získání skutečné adresy L3
    addl $_GLOBAL_OFFSET_TABLE_+[.-.L3],%ebx
    movl 8(%ebp),%eax # ebx=adresa GOT knihovny
    pushl %eax     # parametr a
    call g@PLT     # skok přes pomocnou spojku
    movl v@GOT(%ebx),%edx  # statická adresa
    addl (%edx),%eax   # proměnná v
    movl e@GOT(%ebx),%edx  # externí adresa
    addl (%edx),%eax   # proměnná e
    movl -4(%ebp),%ebx
    leave
    ret
```

- dump hlaviček ELF viz *readelf*
- pro IA32 je rezervován registr EBX jako báze GOT (tabulka adres všech globálních/externích statických dat knihovny)
- PLT (Procedure Linkage Table) – generuje sestavovací program v kódové části knihovny (programu)

```
.PLT0:  pushl 4(%ebx)       # který vstupní bod
        jmp *8(%ebx)        # adresa dyn.sest.prog
.PLT1:  jmp *name1@GOT(%ebx) # počátečně *+4
        pushl $offset       # ofset vstupního bodu
        jmp .PLT0@PC
.PLT2:  jmp *name2@GOT(%ebx)
        pushl $offset
        jmp .PLT0@PC
```

Průběh volání funkce z/v dynamické knihovně:
1. Skok na položku PLT,
2. Nepřímý skok přes položku v GOT,
3. Počátečně je položka v GOT naplněna adresou následující instrukce v PLT, což je pomocný kód, který přípraví identifikaci volaného vstupního bodu na zásobník a zavolá dynamický sestavovací program pro vyhledání tohoto vstupního bodu a umístění jeho adresy do GOT

Proč? odložení relokace funkcí, není třeba relokovat (modifikovat) binární kód modulů, volání funkcí se neliší mezi statickými/dynamickými knihovnami

Windows – PE (Portable Executable)

Vychází ze starší verze formátu binárních modulů Unix SVR3 – COFF (Common Object File Format). Podporuje dynamickou relokaci sdílených knihoven při zavedení do paměti, ale bez pozičně nezávislého kódu. Pokud je tedy sdílená knihovna (DLL) namapována v různých procesech na různé adresy, musí být v paměti více kopií kódové části. Implicitně je knihovna mapována na doporučenou logickou adresu. Pokud ji nelze v daném procesu použít, protože je obsazená něčím jiným, nastaví se pro kód příznak COW (copy on write) a sestavovací program relokuje všechny pozičně závislé instrukce na jinou bázovou adresu.

Při použití sdílených knihoven v C je třeba používat explicitně klíčová slova *__declspec(dllexport)* a *__declspec(dllimport)* (volání jdou opět nepřímo přes pomocný kód) nebo seznam vstupních bodů v *import library* (*.lib).

b) Explicitní sdílení

- pro komunikaci mezi procesy
- různá rozhraní – SVR3 IPC, SVR4+POSIX mmap()

SVR3:

První rozhraní pro sdílení pamětí, součást IPC:

```
shmid = shmget(key, size, IPC_CREAT|SHM_R|SHM_W);
m = shmat(shmid, (void *)0, 0)
```

- lze předat mezi nezávislými procesy jménem klíče (key)

BSD 4.4:

Zavedena univerzální operace mapování souboru do adresového prostoru, viz následující kapitola. Lze využít také pro zobrazení sdílené paměti:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
m = mmap(0, size, PROT_READ|PROT_WRITE,
           MAP_SHARED|MAP_ANON, -1, (off_t)0);
```

Adresový prostor lze sdílet pouze pouze následně vzniklými procesy (*fork()*).

SVR4:

Převzata operace *mmap()* z BSD, ale jako zdroj stránek musí být vždy soubor. Lze použít /dev/zero:

```
fd = open("/dev/zero", O_RDWR);
m = mmap(0, size, PROT_READ|PROT_WRITE,
           MAP_SHARED, fd, (off_t)0);
close(fd);
```

POSIX 1003.1b
Odstraňuje omezení sdílení, lze vytvořit pojmenovanou sdílenou paměť *name*, kterou mohou sdílet i nezávislé procesy:

```
fd=shm_open(name,O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);
m = mmap(0, size, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, (off_t)0);
close(fd);
```

- jméno sdíleného úseku je globální v rámci systému, nevyžaduje záložní soubor (odkládání na swap)

Windows/NT
Funguje obdobně jako *shm_open()* + *mmap()*, jméno *name* je globální v rámci systému (sezení):

```
fmh=CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
    PAGE_READWRITE, szhi, szlo, name);
m = MapViewOfFile(fmh, FILE_MAP_WRITE, ofset_hi,
    ofset_lo, length);
```

Anonymní sdílenou paměť lze vytvořit také pomocí *CreateFile()* pro získání *file handle* (první parametr).

Pokud už je sdílená paměť vytvořena, další procesy ji mohou otevírat voláním:

```
fmh=OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE,
     name);
m = MapViewOfFile(fmh, FILE_MAP_WRITE, ofset_hi,
    ofset_lo, length);
```

7. Alokace logického adresového prostoru

Logický adresový prostor je obsazován dynamicky:

- dynamické knihovny
- sdílená paměť
- dynamická data
- zásobníky pro vlákna

Je třeba přidělovat celistvé úseky o proměnné velikosti, zarovnané na hranice stránek – problém správy úseků o proměnné velikosti

Obsazení LAP pro proces zachycuje mapa adresového prostoru (region map, object map, apod.):

- začátek a délka úseku
- zdroj dat (kód, statická data ze souboru, ostatní volné stránky)
- odkládací prostor (swap, datový soubor)
- ochrana (čtení, zápis, provádění)
- stav sdílení (které procesy sdílí daný objekt)
- stav COW (obvykle na úrovni celých úseků)

Z tabulky obsazení LAP se pak sestavuje tabulka stránek, případně při výpadcích stránek naplňuje.

Tabulka obsazení fyzické paměti:

Pro každý rámec:

- kterému objektu patří stránka zavedená do paměti
- číslo stránky v rámci objektu
- počet použití při sdílení
- stav zamčení
- stav zavedení, odložení
- zařazení do seznamu volných, neaktivních stránek

Rychlé hledání dvojice (objekt, stránka) ve volných, neaktivních - hashování