Co by měl každý programátor vědět o počítačové paměti

Petr Holášek
holasekp@gmail.com

Univerzita Mikuláše Koperníka

23. února 2013

umk

Popis předmětu IMEM

Přednášky:

- Jedna přednáška 23.2.2013; 10:00

Zkouška:

- První a poslední termín zkoušky 24.2.2013
- Full-text + sada otázek na ABCD

Hodnocení:

- 0-49 b. = F
- 50-59 b. = E, 60-69 b. = D, 70-79 b. = C, 80-89 b. = B, 90-100 b. = A

Obsah přednášky

1. Paměťová hierarchie
2. Operační systém
3. Virtuální paměť
4. Práce s pamětí z pohledu programátora

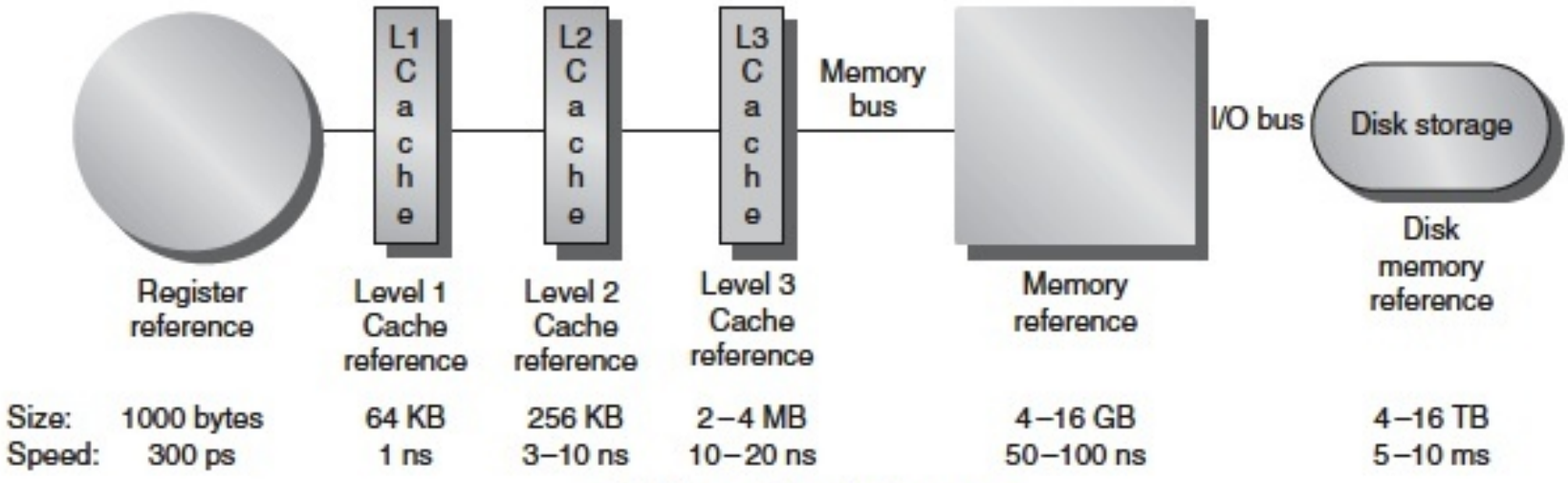
Co je to paměť?

- Prostor k uložení dat
- Různé velikosti a různé rychlosti, platí nepřímá úměra (viz. Paměťová hierarchie)
- V dnešních operačních systémech je nutné použít pokročilé techniky správy paměti

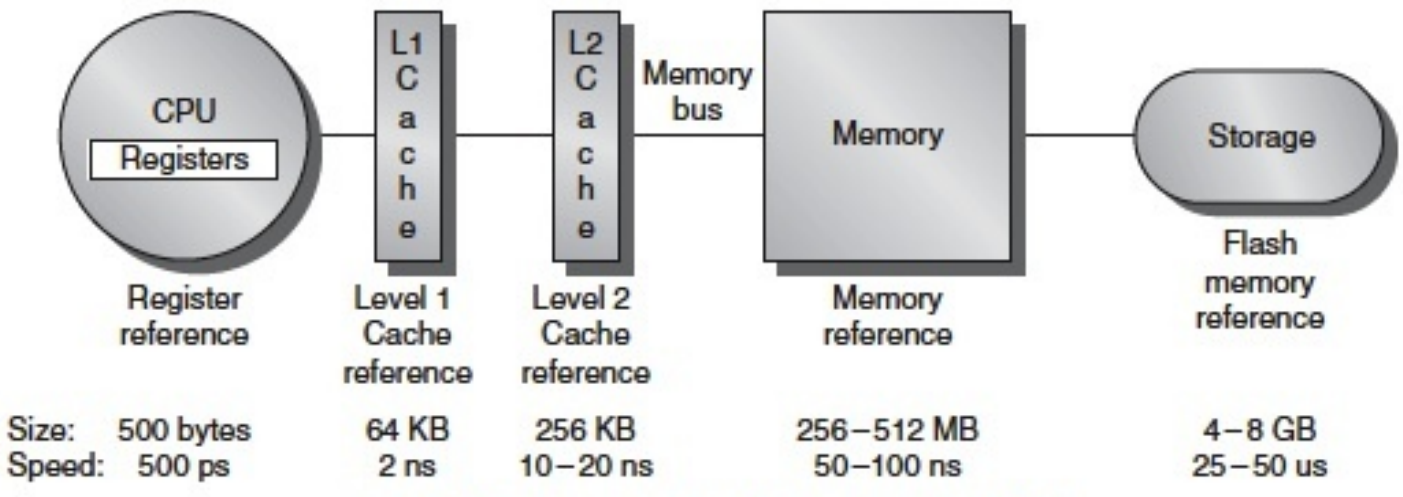
Paměťová hierarchie

- Uspořádání paměťových úložišť:
  - Vzestupně podle kapacity
  - Sestupně podle rychlosti
- Data se pohybují oběma směry
- Stupně:
  - Registry procesoru
  - Cache (L1, L2, L3)
  - Hlavní paměť (RAM - Random Access Memory)
  - Sekundární úložiště (disk, páska)

Paměťová hierarchie



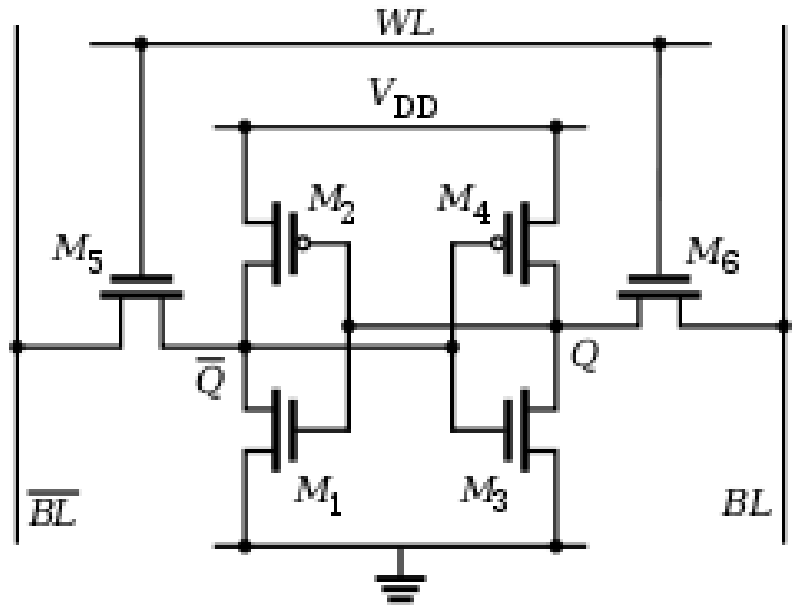
(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

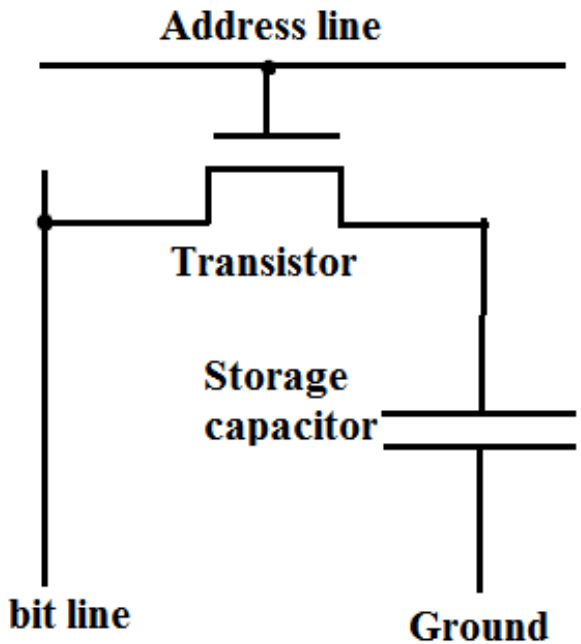
Statická RAM aka SRAM

- SRAM je obvykle použita v registrech a cache
- Výhodnější vlastnosti než DRAM, ale má vyšší spotřebu a její výroba je dražší
- Stav buňky není potřeba obnovovat, ale je třeba dodávat konstantní proud



Dynamická RAM aka DRAM

- Použití v hlavní paměti počítače
- Stav buňky uchován v kapacitoru C
- Nutnost obnovování obsahu buňky každých několik desítek milisekund
- Buňky jsou obvykle uspořádány v matici, důvodem je velikost paměti ($4GB = 2^{32}$ adres)



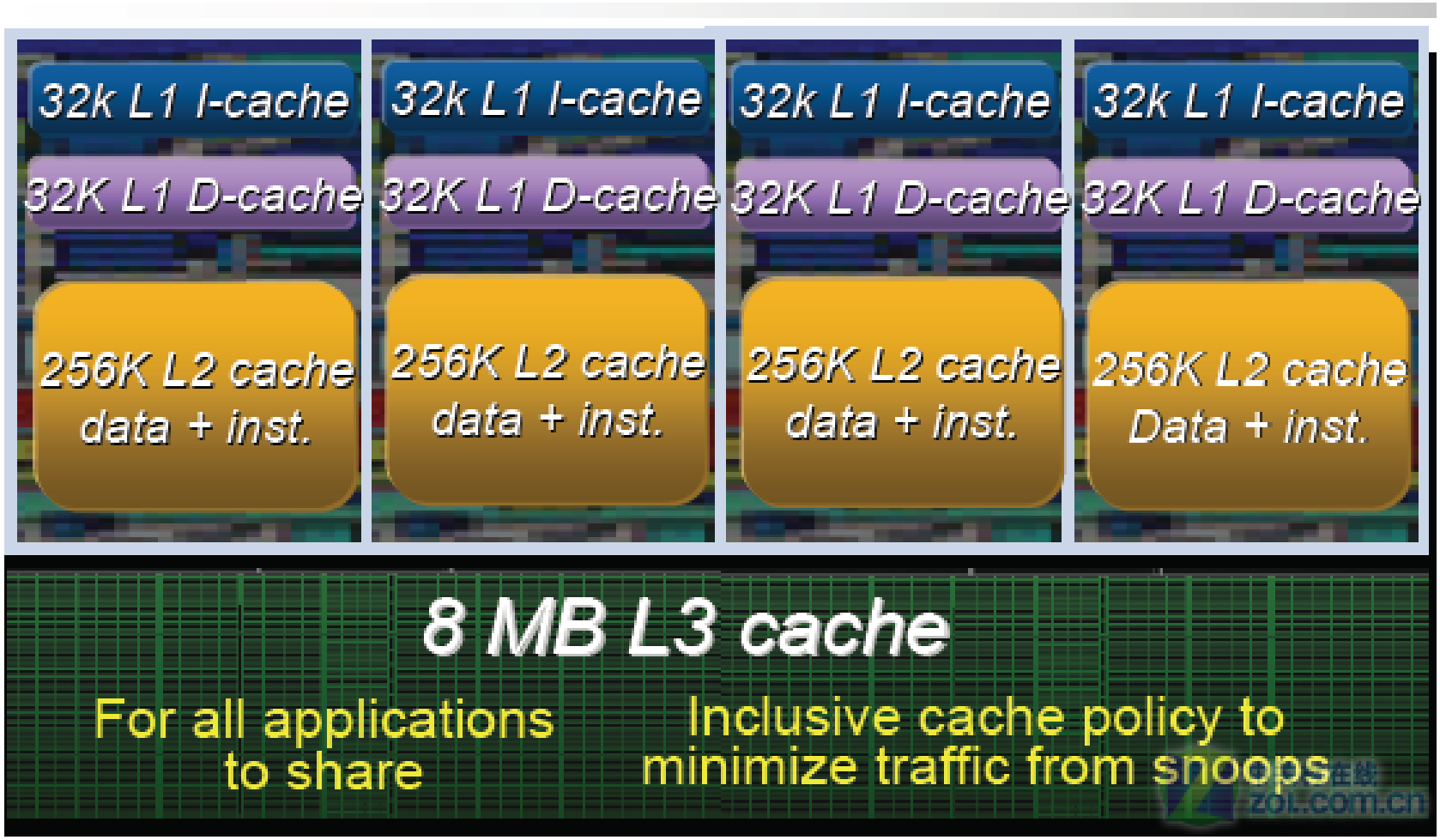
Registry CPU

- Velice malé kusy paměti uvnitř CPU (typicky 32-64 bit podle architektury)
- eax, ebx, eip, esp, ... (asemblerovi labužníci znají, odvážnější linuxisté určitě někdy viděli)
- Nové sady SIMD instrukcí typu SSE nebo AVX mají svoje vlastní

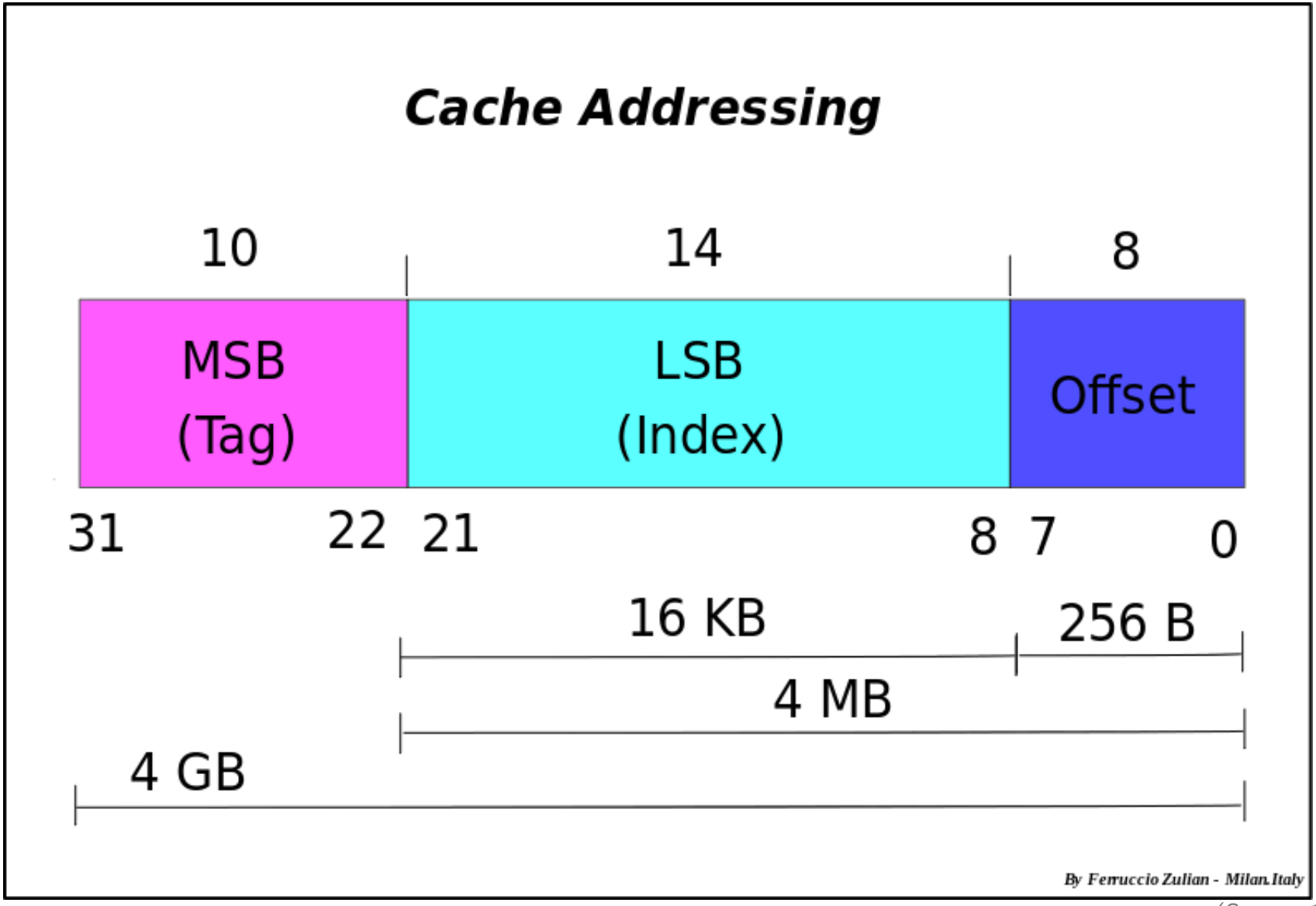
Cache

- Rychlá a malá paměť mezi CPU a hlavní pamětí pro překlenutí velkého rychlostního rozdílu
- V nejnovějších procesorech obvykle třístupňová L1-L3
- Využívá principů *časové* a *místní lokality*

Uspořádání pamětí cache v dnešních procesorech



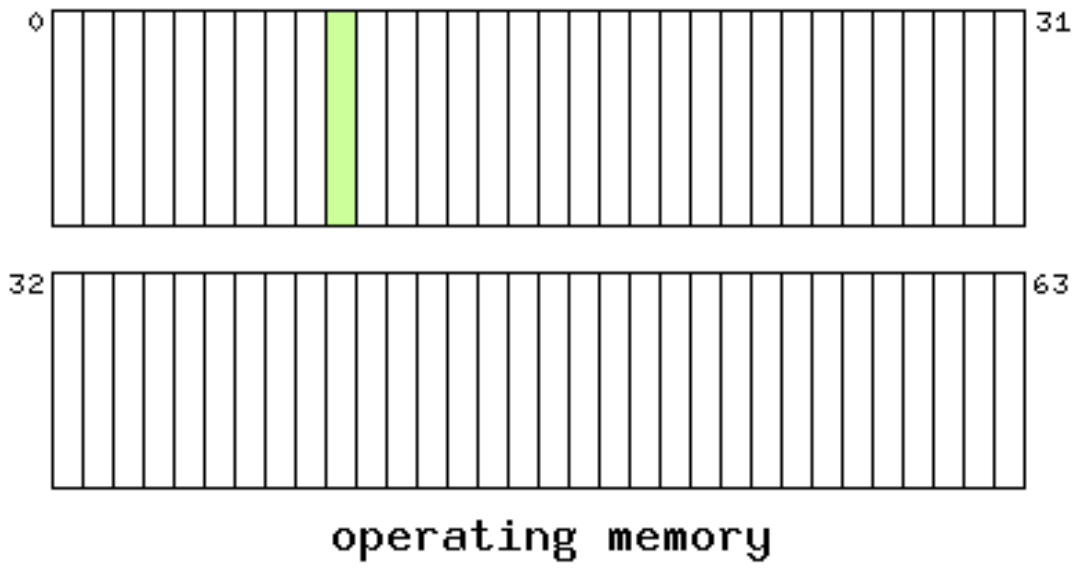
Adresování dat v cache



Asociativita cache

- plně asociativní - není třeba index, blok může být umístěn kdekoli v cache, flexibilní co se týče umístění bloků, použitelné pouze u velmi malých cache kvůli počtu porovnání tagů
- přímo mapovaná - každý blok může být umístěn pouze na jednom místě v cache, neflexibilní umístění bloků, klasicky modulo
- kompromisem je *n-cestná cache* - index najde cestu, tag konkrétní blok, kompromisní řešení

Asociativita cache



Metody propagace dat z/do cache

write-through Data jsou zapsány zároveň do bloku cache i do bloku paměti na nižší úrovni.

write-back Data jsou zapsány pouze do bloku cache. Tento modifikovaný blok je zapsán do hlavní paměti až při jeho výměně.

Metriky používané pro paměti cache

- *miss rate*
- *hit rate*
- *instruction cycles per miss*
- *instruction cycles per hit*
- Pro experimenty s Vašimi binárkami:
  `valgrind --tool=cachegrind ./muj_program`

Cachegrind

Ukázkový výstup pro quicksort na poli náhodně generovaných 10000 prvku:

```
==9471== I   refs:      18,321,256
==9471== I1  misses:    814
==9471== LLi misses:    808
==9471==
==9471== D   refs: 6,219,587 (3,828,621 rd + 2,390,966 wr)
==9471== D1  misses: 3,723  (2,754 rd   +  969 wr)
==9471== LLd misses: 1,494  (635 rd   +  859 wr)
==9471==
==9471== LL refs:   4,537  (3,568 rd   +  969 wr)
==9471== LL misses: 2,302  (1,443 rd   +  859 wr)
```

Operační systém

- Vrstva mezi HW a uživatelských SW
- Poskytuje programátorům iluzi, že:
  - paměti pro jejich program je nekonečně mnoho
  - adresy jdou hezky spojitě od $0 - N$
  - jejich proces běží na svém (nebo svých) CPU sám a nepřetržitě

... nicméně jádro OS se zapotí, aby této iluze dosáhlo.

Proces

- Jde o zapouzdření běžící aplikace (`bash`, `openoffice`, `gdb`) z pohledu OS
- Na systému jich může běžet "zároveň" velmi mnoho (desítky tisíc)
- Každý proces má vlastní lineární paměťový prostor, který může používat:

  32-bit 4GB (proč právě tolik?)

  64-bit teoreticky 16EB, prakticky v Linuxu 128TB
- Procesy jsou postupně přepínány plánovačem jádra tak, aby se každý dostal na určitý čas k procesu

Stránkování

- Na všech moderních OS je nejmenší použitelné množství paměti 1 stránka
- Typicky 4kB, existují však i větší (viz. hugepages v Linuxu)
- Z pohledu OS rozlišujeme dva typy stránek:
  - Virtuální stránky - viditelné pro běžící procesy a tedy programátory
  - Fyzické rámce - paměť v RAM

Virtuální paměť

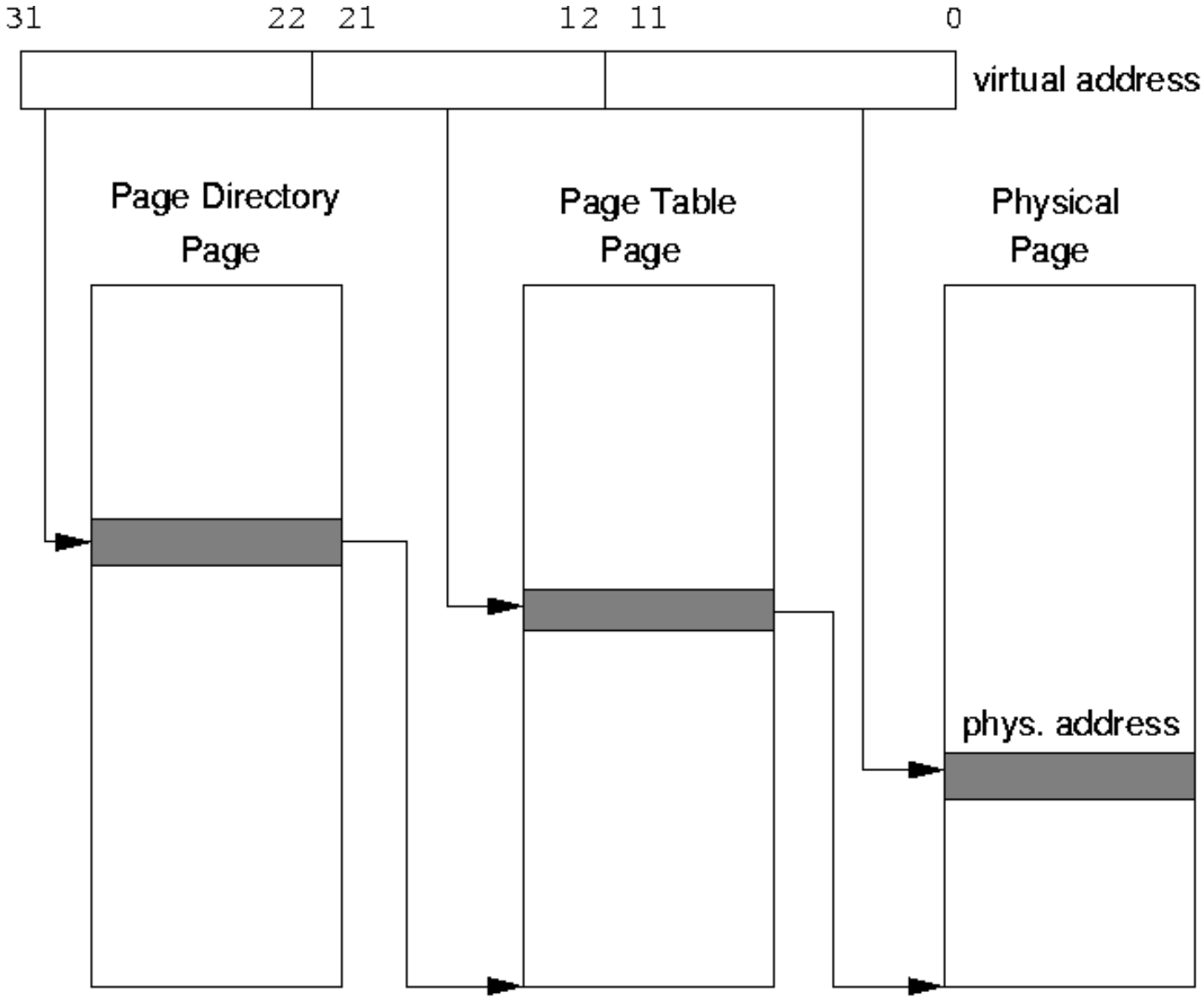
- Koncept řešící současný běh více procesů na jednom systému
- Každý běžící proces má k dispozici 4GB adresového prostoru rozděleného na jednotlivé *stránky*
- Virtuální paměť řeší přiřazení fyzických rámců ke stránkám virtuálních (ne nutně 1:1 - u sdílených knihoven)
- Adresa `0x89f92ad2` u procesu 1 ukazuje do jiné buňky paměti než stejná adresa u procesu 2

Tabulka stránek

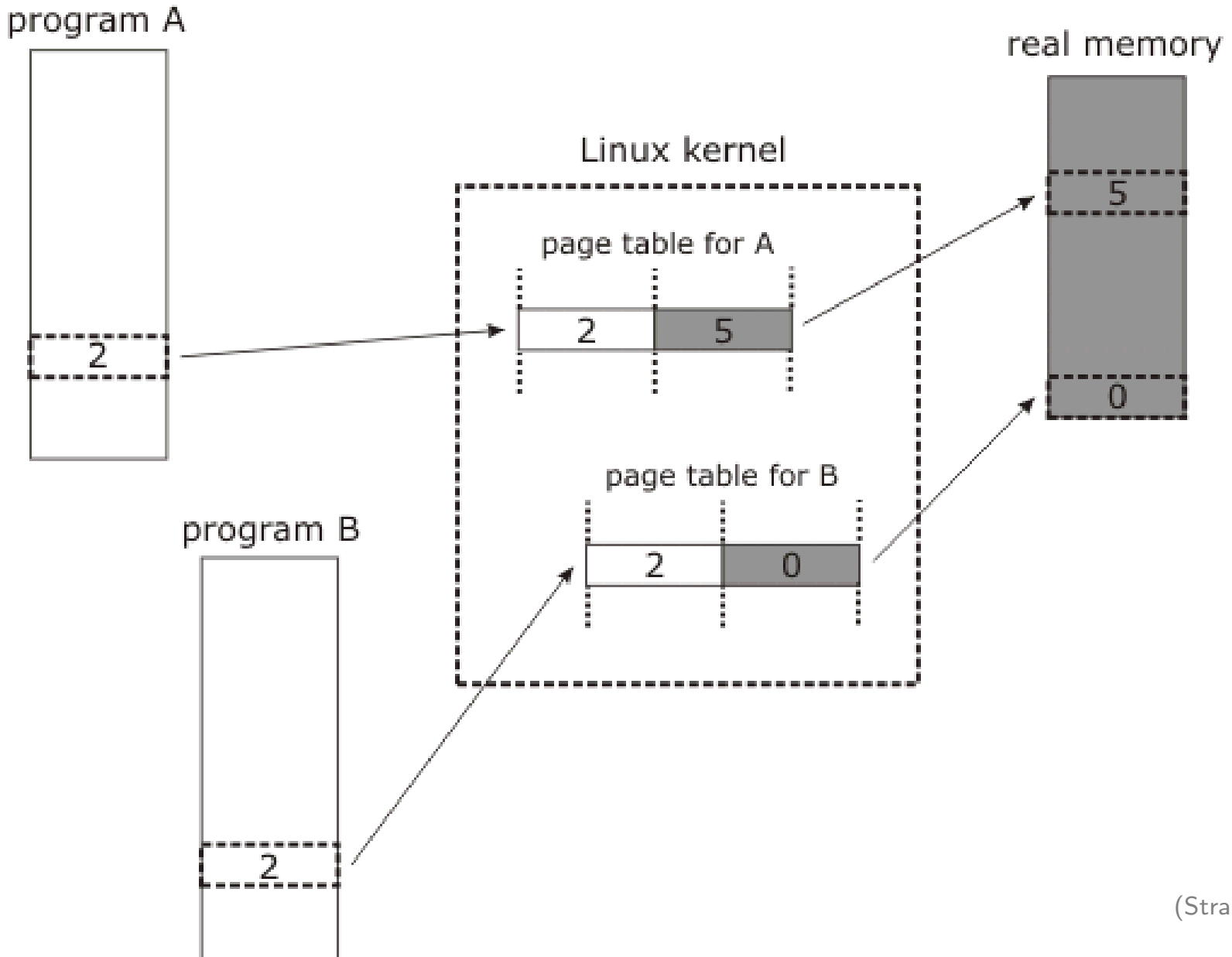
- Udržuje mapování virtuálních stránek na fyzické rámce pro **každý** běžící proces
- Adresa je rozdělena na několik částí podle počtu stupňů TS a offset
- Hledání v tabulce je časově náročné, existuje tedy malá cache TLB (Translation lookaside buffer), která obsahuje poslední úspěšné překlady mapování
- Při přepnutí procesu je TLB vyprázdňována

TODO: Obrázek!

Tabulka stránek



Překlad adres



Swapování stránek

- Řeší záhadu proč může každý proces používat 4GB paměti i když je fyzicky nainstalováno méně RAM
- Pokud je hlavní paměť RAM plná, jádro OS vybere nejméně potřebnou stránku a uloží ji na swap oddíl, typicky na sekundárním úložišti:
  - v Linuxu obvykle samostatný diskový oddíl typu swap
  - ve Windows stránkovací soubor viditelný přímo na disku
- Poté toto uvolněné místo ve fyzické paměti nahradí novou požadovanou stránkou
- Při přístupu na stránku umístěnou ve swapu je stránka opět nahrána do hlavní paměti, kde opět vystřídá jinou, "nejméně" potřebnou, pokud není stále volné místo.
- V designu paměťového podsystému je vždy velká snaha pomoci použití sofistikovaných algoritmů omezit použití swapu na minimum

Virtuální mapování uživatelských procesů

- Každý spuštěný proces v OS obsahuje ve své virtuální paměti tyto namapované segmenty:

text	instrukce procesu
data	statické data a literály
heap	"halda", paměť určená k uživatelské alokaci, např. `malloc()/free()`
stack	zásobník používaný pro argumenty funkcí a lokální proměnné

- Pomocí utility `pmap` lze jednotlivé segmenty paměti jednoduše rozpoznat.

Použití pmap

```
[root@thinkpad-work imem_slides]# pmap 729
729:   /usr/sbin/irqbalance
0000000000400000     32K r-x--  /usr/sbin/irqbalance
0000000000607000      4K r----  /usr/sbin/irqbalance
0000000000608000      8K rw---  /usr/sbin/irqbalance
000000000250a000    132K rw---    [ anon ]
000000326d800000     40K r-x--  /usr/lib64/libnuma.so.1
000000326da0a000      4K rw---  /usr/lib64/libnuma.so.1
0000003904600000     16K r-x--  /usr/lib64/libcap-ng.so.0
0000003904604000   2044K -----  /usr/lib64/libcap-ng.so.0
...
00007fffb7c02000    132K rw---    [ stack ]
00007fffb7c67000      8K r-x--    [ anon ]
ffffffffff600000      4K r-x--    [ anon ]
 total            21076K
```

Operace nad pamětí v jazyce C

malloc()	alokace paměti na "haldě", přidání/rozšíření virtuálního mapování, vrací pointer
free()	uvolnění alokované paměti, odstranění mapování
mmap()	obecná funkce pro vytvoření mapování, lze použít pro anonymní paměť, pro soubory, i pro zařízení
munmap()	zrušení mapování
mlock()	označená paměť určená pointerem a délkou bude vždy umístěna v hlavní paměti, nebude nikdy odswapována
munlock()	paměť jde opět odswapovat

Co se stane po zavolání malloc()?

1. Jádro OS ověří, zda proces nepřekročil povolenou celkovou velikost a počet mapování
2. Vytvoří nebo rozšíří některé z existujících mapování stejného typu - tedy anonymní
3. Dokud do nově alokované paměti nebude přistupováno, fyzicky se nic alokovat nebude
4. Po zápisu do této paměti se přiřadí volný fyzický rámec k virtuální stránce a vytvoří se záznam v tabulce stránek
5. Pokud je první přístup čtení, odkazuje tabulka stránek k obecné vynulované stránce (zero-page)

Ukázka

```
int main(void)
{
char *mem;

mem = (char *) malloc(4096 * sizeof(char));
if (!mem)
return 1;

bzero(mem, SIZE);
...
free(mem);
```

Kolik stránek bude programem na haldě fyzicky naalokováno po volání `bzero`? Kolik stránek bude programem na haldě fyzicky naalokováno po zakomentování `bzero`?

Literatura

- What Every Programmer Should Know About Memory, Ulrich Drepper, 2007 - ke stažení na `www.akkadia.org/drepper/cpumemory.pdf`
- Computer Architecture, Fifth Edition: A Quantitative Approach, Hennessy & Patterson, 2011
- `http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html`
- `http://www.kernel.org/` (adresář `mm/`)