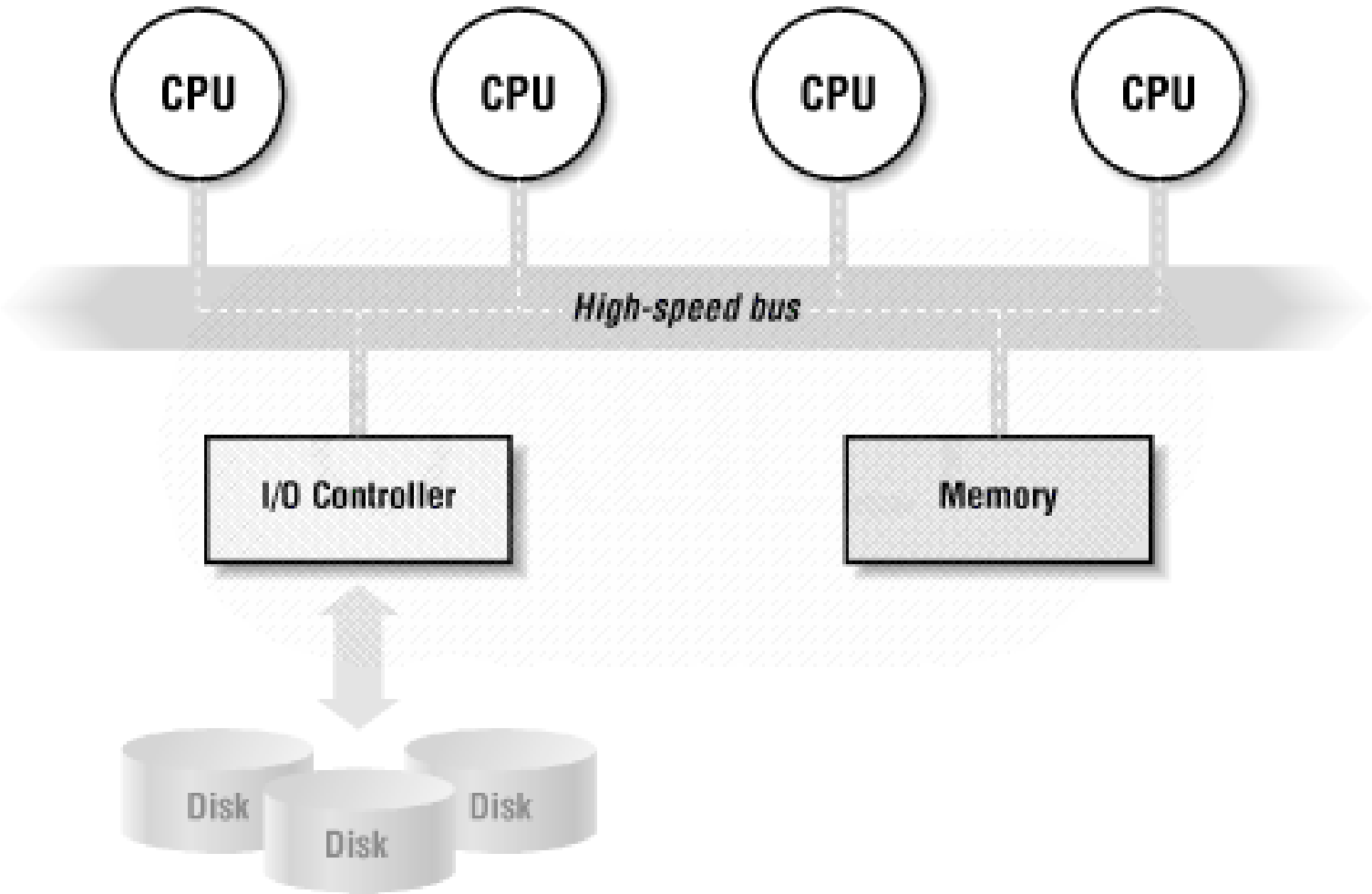
IB109 Návrh a implementace paralelních systémů

Programování v prostředí se sdílenou pamětí

Jiří Barnat

HW model prostředí se sdílenou pamětí



HW platformy

Paralelní systémy se sdílenou pamětí

- Systémy s více procesory
- Systémy s více-jadernými procesory
- Systémy s procesory se zabudovaným SMT
- Kombinace

Rizika paralelních výpočtů na soudobých procesorech

- Mnohé optimalizace na úrovni procesoru byly navrženy tak, aby zachovávaly sémantiku sekvenčních programů.

Pozor zejména na

- Přeuspořádání instrukcí
- Odložené zápisy do paměti

Simultání multithreading (SMT)

Princip

- Procesor využívá prázdné cykly způsobené latencí paměti k vykonávání instrukcí jiného vlákna.
- Vyžaduje duplikaci jistých částí procesoru (např. registry).
- Vlákna sdílí cache.

Příklad: Intel Pentium 4

- Hyper-Threading Technology (HTT)
- OS s podporou SMP vidí systém se SMT/HTT jako více procesorový systém.
- Až 30% nárůst výkonu, ale vzhledem ke sdílené cache může být rychlost výpočtu jednoho vlákna nižší.

Více-jaderné procesory (multicores)

Více plnohodnotných procesorů v jednom chipu.

Výhody

- Efektivnější cache koherence na nejnižší úrovni.
- Nižší náklady pro koncového uživatele.

Nevýhody

- Víc jader emituje větší zbytkové teplo.
- Takt jednoho jádra bývá nižší.
- Automatické dočasné podtaktování/přetaktování.
- Jádra sdílí datovou cestu do paměti.

Realita

- Více-jádrové procesory se SMT.
- Intel Core-i7 (hexa-core se SMT = 12 paralelních jednotek)

Paralelismus v prostředí se sdílenou pamětí

Idealizovaný model

- **Na této úrovni se řeší návrh paralelního algoritmu.**
- Jednotlivá výpočetní jádra paralelního systému pracují zcela nezávisle.
- Přístupy k datům v paměti jsou bezčasové a vzájemně výlučné.
- Komunikace úloh probíhá atomicky přes sdílené datové struktury.

Realita

- **Na této úrovni musí programátor řešit technickou realizaci paralelního algoritmu.**
- Přístup do paměti přes sběrnici je pro CPU příliš pomalý.
- Registry procesoru a cache paměti – rychlé kopie malého množství dat na různých místech datové cesty.
- Problém koherence dat.

Paralelní úlohy v kontextu OS – Procesy a vlákna

Procesy

- Skrývají před ostatními procesy své výpočetní prostředky.
- Pro řešení paralelní úlohy je potřeba mezi-procesová komunikace (IPC).
  - Sdílené paměťové segmenty, sokety, pojmenované a nepojmenované roury.

Vlákna

- Existují v kontextu jednoho procesu.
- V rámci rodičovského procesu sdílí výpočetní prostředky.
- Komunikace probíhá přes sdílené datové struktury.
- Účelem interakce je spíše synchronizace než transport dat.
- Subjekty procedury plánování.

Příklad použití vláken v rámci procesu

Vlákno

- Realizuje výpočet, tj sekvenci instrukcí.
- Každý proces je tvořen alespoň jedním vláknem.
- Hlavní vlákno procesu vytváří další vlákna.

Příklad

```
1 for (i=0; i<n; i++)
2   for (j=0; j<n; j++)
3     m[i][j] = create_thread(
4                 product(getrow(i),getcol(j))
5               )
```

Procesy versus vlákna

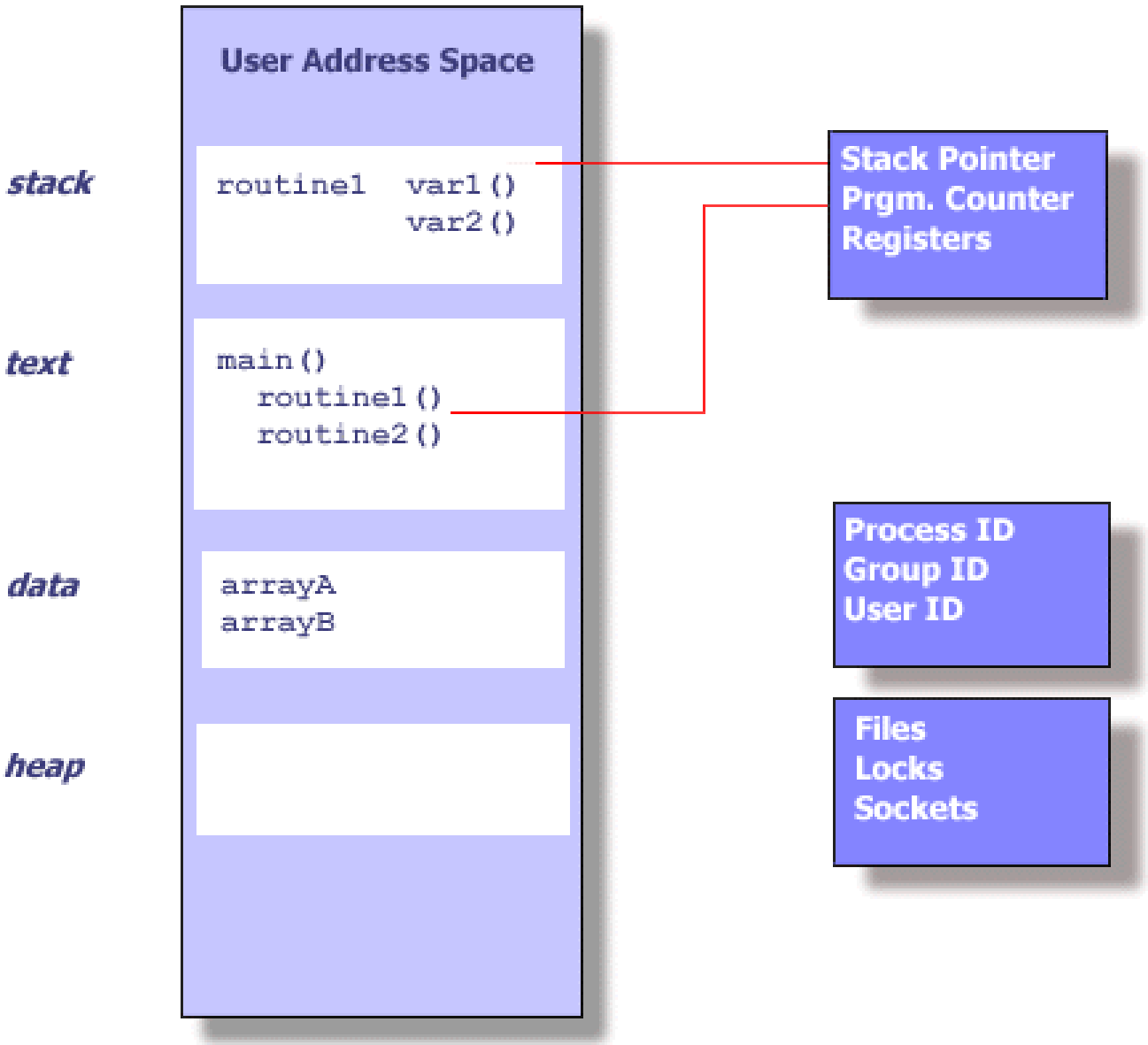
Proces

- Identifikátory procesu a vlastníka
- Proměnné prostředí, pracovní adresář
- Kód
- Registry, Zásobník, Halda
- Odkazy na otevřené soubory a sdílené knihovny
- Reakce na signály
- Kanály IPC

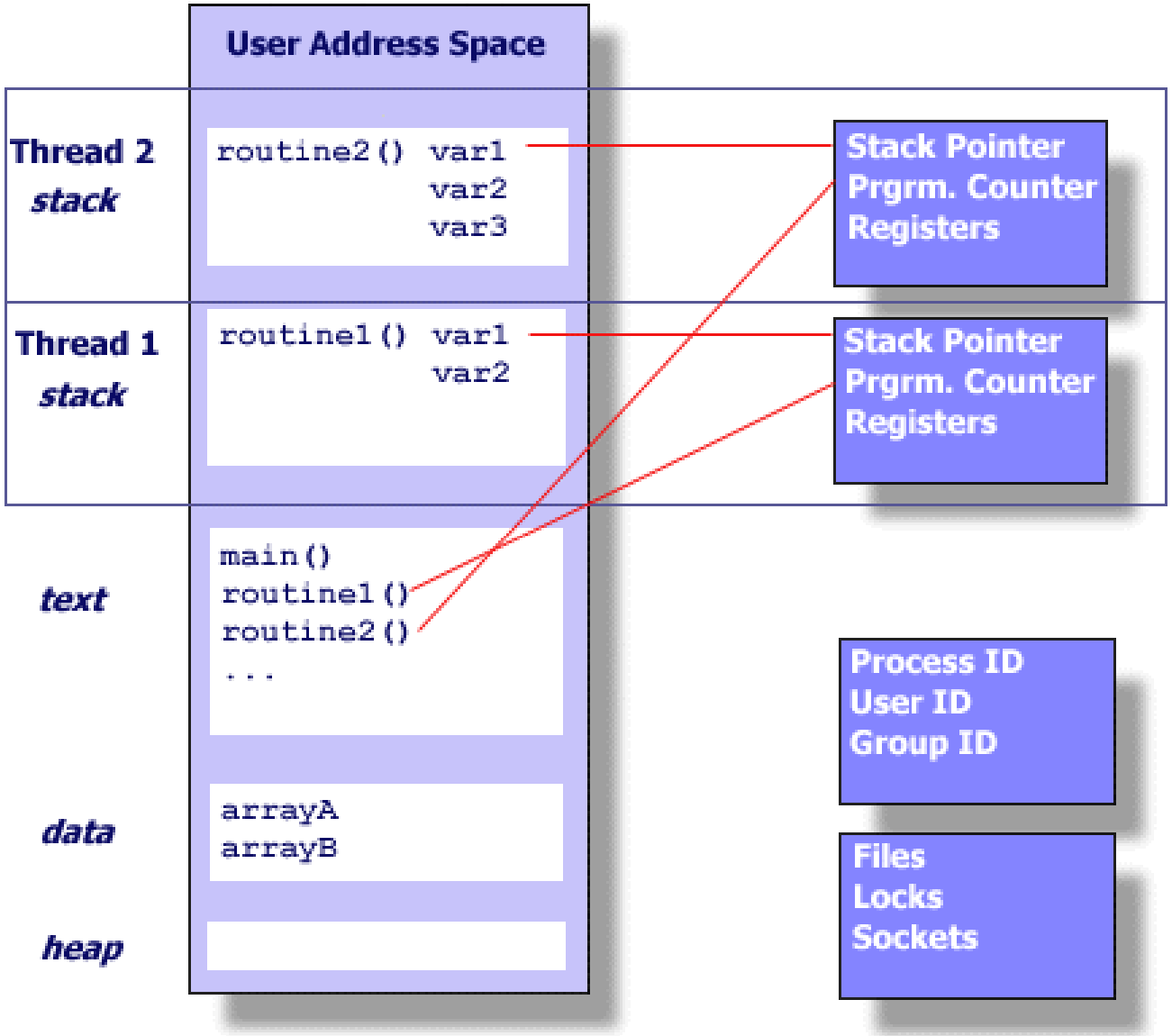
Vlákna mají privátní

- Zásobník
- Registry
- Frontu signálů

Proces v operačním systému



Vlákna v rámci procesu



Argumenty pro používání vláken místo procesů

Výkon aplikace

- Vytvoření procesu je výrazně dražší než vytvoření vlákna.
- Změna dat provedená v rámci jednoho vlákna je viditelná v kontextu celého procesu.
- Komunikace mezi vlákny spočívá v předávání reference na data, nikoliv v předávání obsahu.
- Předávání reference se děje v rámci jednoho procesu, operační systém nemusí řešit skrývání dat a přístupová práva.

Nevýhody

- Vlákna nemají žádné "soukromí".
- Sdílené globální proměnné.

Efektivní využití cache

Cache

Princip cache

- Menší ale rychlejší paměť na pomalé datové cestě.
- Při prvním čtení se okolí čtené informace uloží do cache.
- Při následujícím čtení z okolí původní informace se čte pouze z cache.

Koherence

- Soulad dat uložených v paměti počítače (potažmo v cache).
- Je zajištěno, že existuje právě jedna platná hodnota asociovaná s daným paměťovým místem.
- Z důvodu rychlosti jsou zápisy procesoru do paměti odkládány a sdružovány, bližší specifikace chování procesoru v tomto ohledu je dána **paměťovým modelem** daného CPU.

Cache – související pojmy

Související pojmy

- Cache line – atomický paměťový blok uložený v cache.
- "Hit ratio" – číslo vyjadřující úspěšnost obsloužení požadavků na data daty uloženými v cache.
- "Vylití cache" – procedura aktualizace dat v paměti hodnotami uloženými v cache.

Zásady efektivního použití cache

- Časová lokalita – přístupy v malém časovém intervalu.
- Prostorová lokalita – přístup k datům uložených adresně blízko sebe.
- Zarovnaná alokace paměti (např. memalign (GNU C)).

False sharing – příklad

Specifikace

- Paralelní cache koherentní systém s více procesory.
- Program s několikanásobně vícero vlákny.
- Pole hodnot `int pole[nr_of_threads]`.
- Vlákna počítají výslednou hodnotu typu `int` a k výpočtu si ukládají mezivýsledek typu `int`.

Varianty implementace

A) Každé vlákno opakovaně zapisuje do datového pole integerů na pozici určenou jeho ID.

B) Každé vlákno zapisuje do lokální proměnné a před skončením nakopíruje hodnotu do pole na pozici určenou jeho ID.

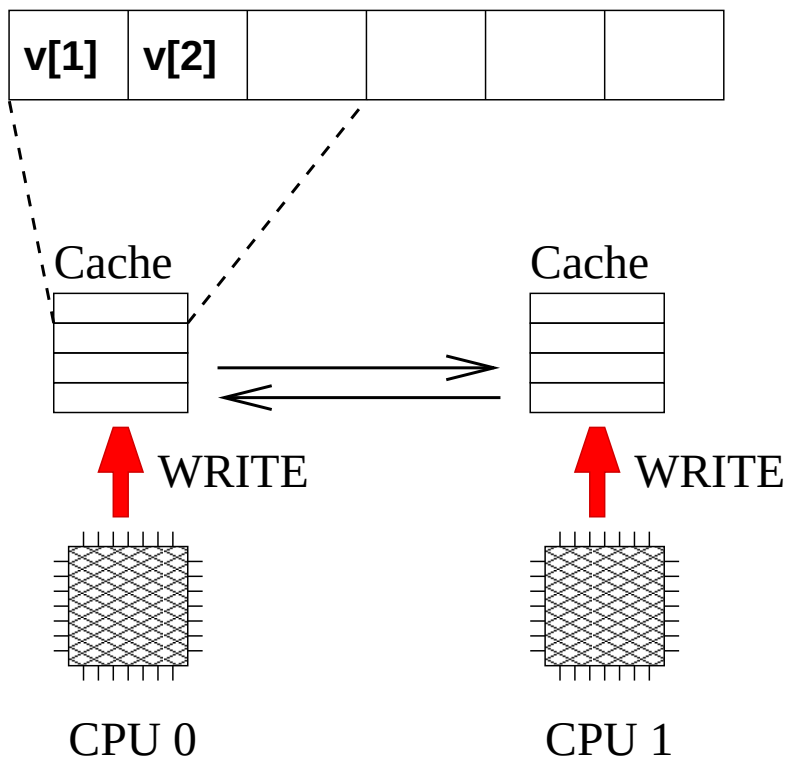
Otázka

- Která implementace bude pomalejší a proč?

False sharing

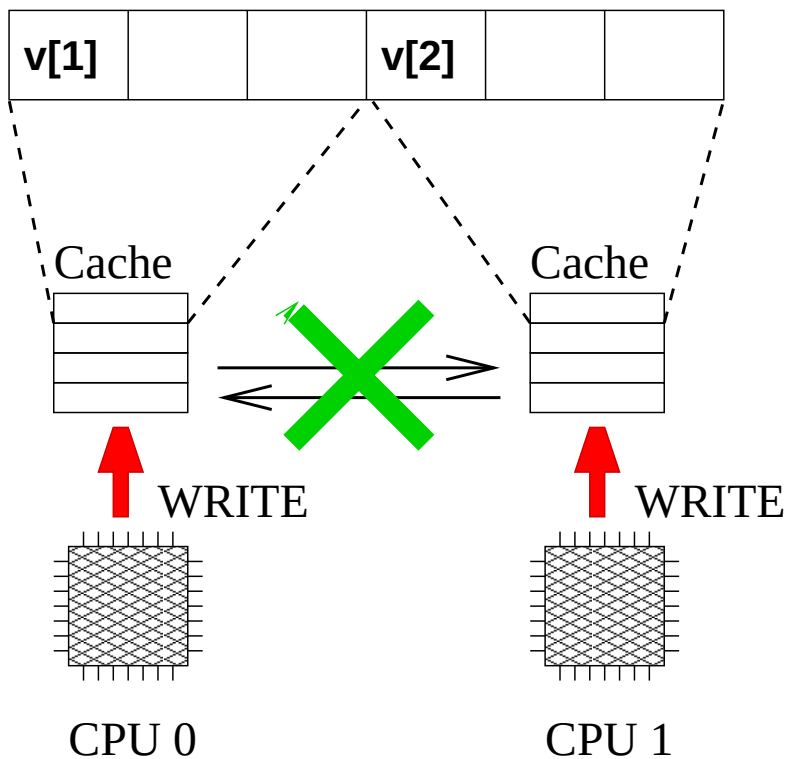
```
typedef struct {
    long long iter;

} thread_private_data_t;


thread_private_data_t v[16];
```

```
my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



False sharing

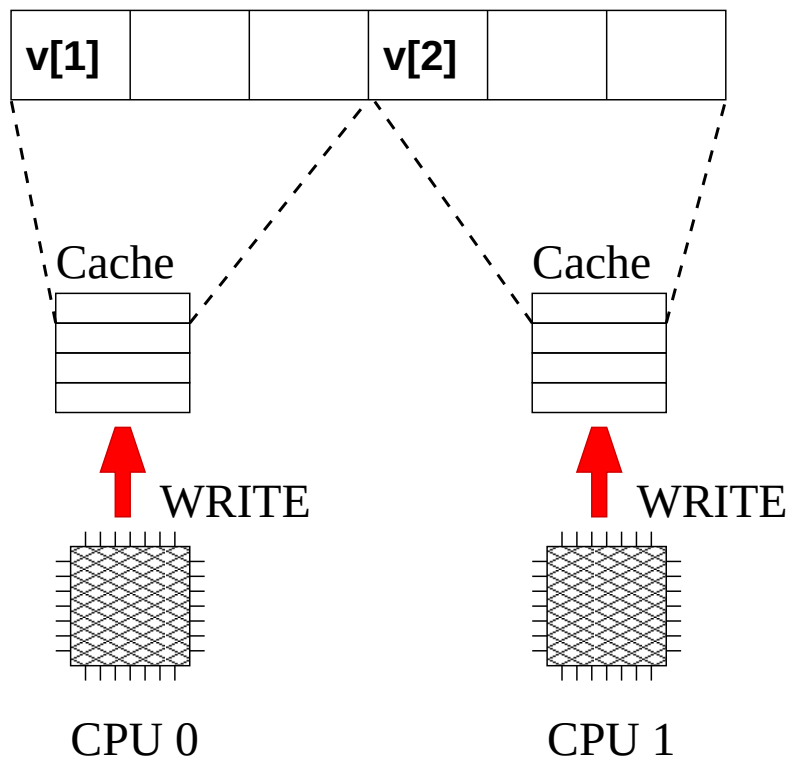
```
typedef struct {
    long long iter;
    char cache_line_filler[2000];
} thread_private_data_t;

thread_private_data_t v[16];

my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



False sharing

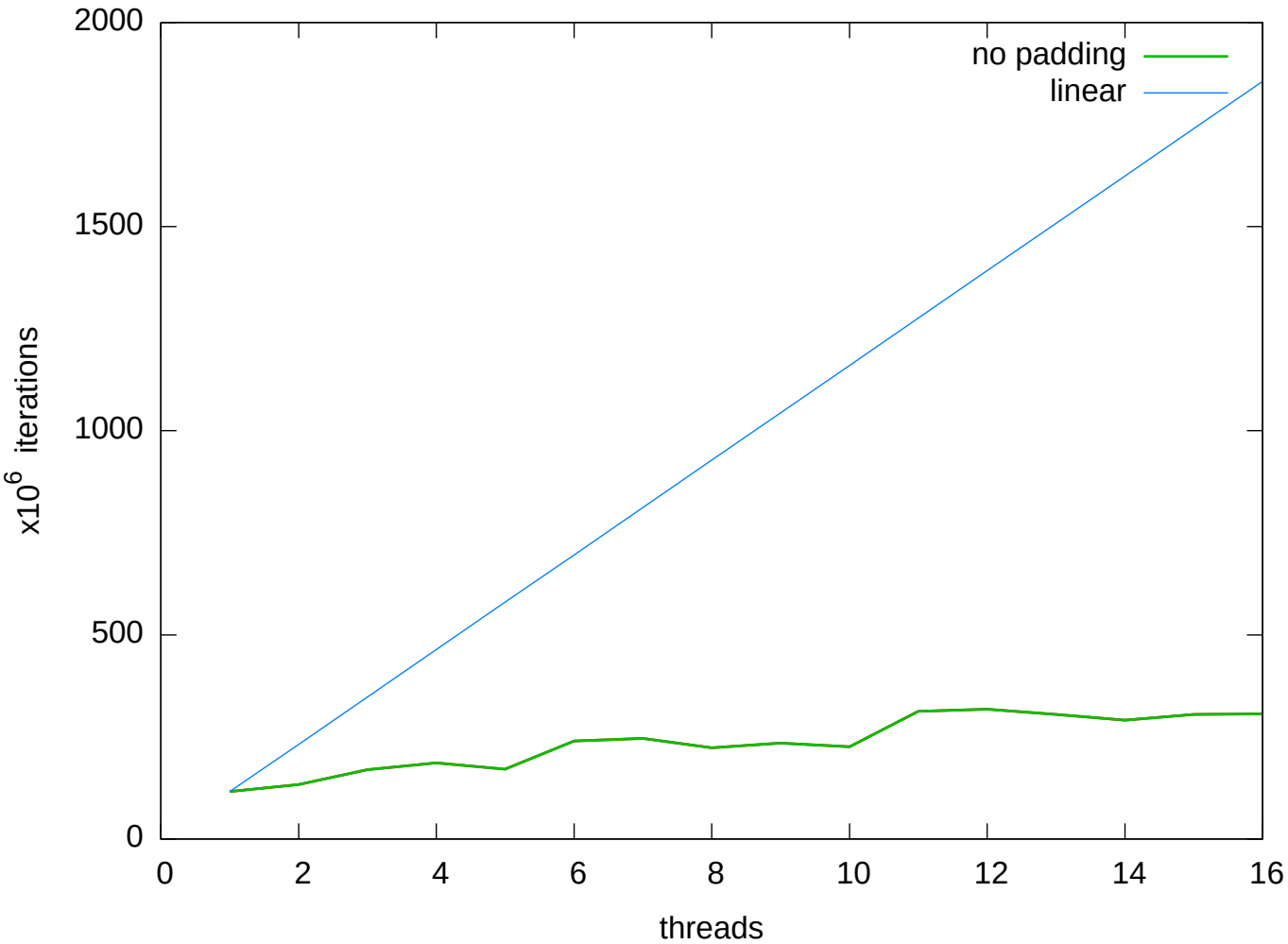
```
typedef struct {
    long long iter;
    char cache_line_filler[2000];
} thread_private_data_t;


thread_private_data_t v[16];
```

```
my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



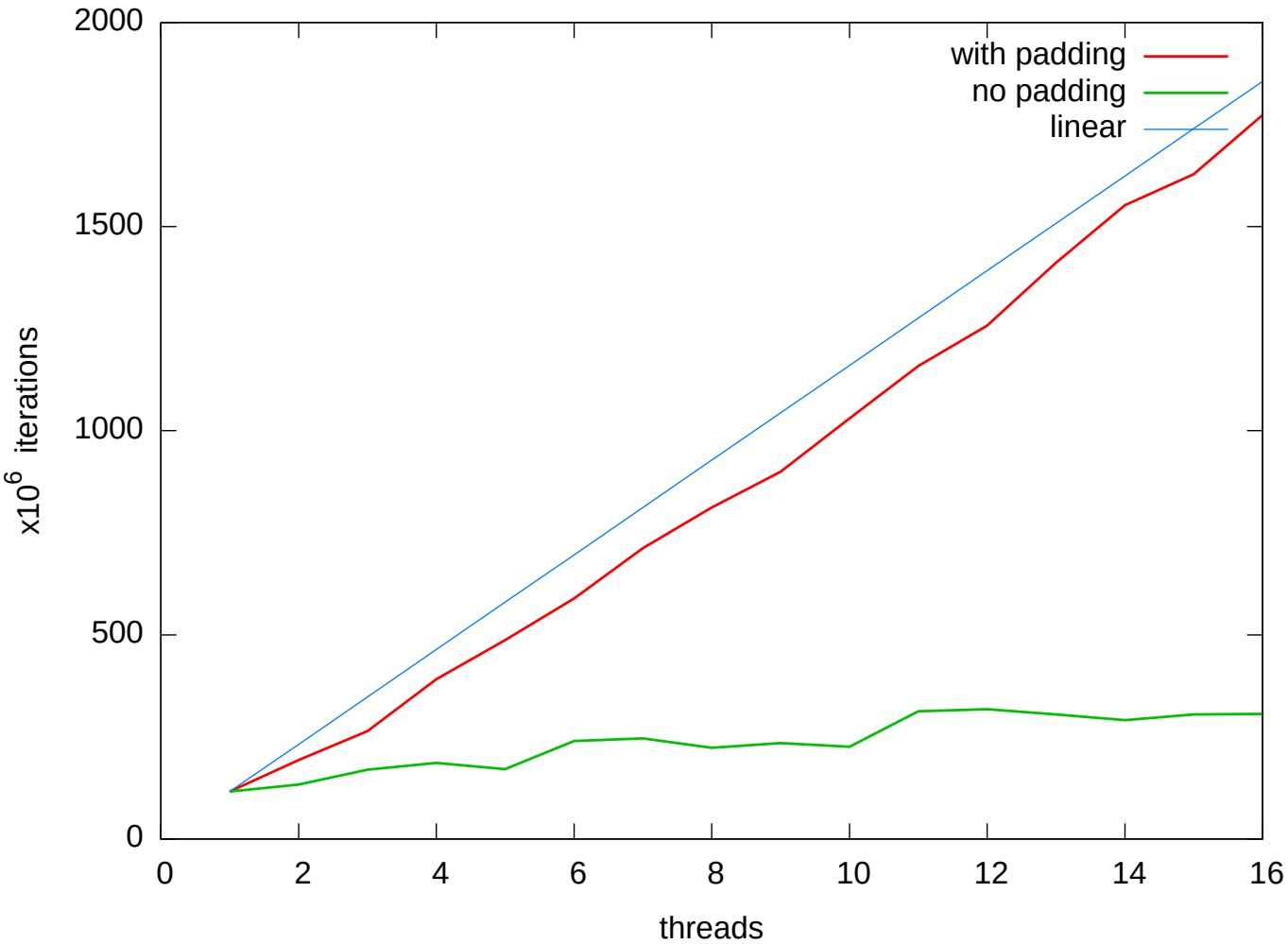
False sharing

1 thread = 119 $x10^6$ iterations (1.00x)
10 threads = 231 $x10^6$ iterations (1.94x)
16 threads = 313 $x10^6$ iterations (2.63x)



False sharing

1 thread = 119 $x10^6$ iterations (1.00x)
10 threads = 1055 $x10^6$ iterations (8.86x)
16 threads = 1815 $x10^6$ iterations (15.25x)



Přehnaná důvěra v mentální model

POZOR!

- Není garantováno, že všechny informace uložené do explicitně definovaných proměnných budou zapsány do paměti.
- Proměnné mohou být realizovány registrem procesoru.

Důsledek

- Posloupnost hodnot uložených do jedné proměnné v rámci jednoho vlákna může být viděna jiným vláknem jen **částečně** nebo **vůbec**.

Nestálé proměnné

Pozorování

- Udržování koherence cache pamětí je nákladné.
- Soudobé překladače nevynucují, aby každá vypočítaná hodnota byla uložena do paměti (nevygenerují instrukci ukládající obsah registru do paměti).

Důsledek

- Modifikace sdílené proměnné provedená v jednom vlákně na daném CPU se nemusí projevit v jiném vlákně na jiném CPU.

Příklad

```
int i=0;

P0 {                          P1 {
  while ( i==0 );               if (i==0) i++;
}                             }
```

Nestálé proměnné

Problém

- Vlákno P0 neskončí, neboť nezaznamená změnu proměnné `i`.
- Může záviset na stupni optimalizace překladu, například
  - chyba se neprojeví při překladu pomocí `g++ -O0`
  - chyba se projeví při překladu pomocí `g++ -O2`

Řešení

- Je nutné označit proměnnou jako tzv. nestálou proměnnou.
- C,C++: klíčové slovo `volatile`.
- Překladač zajistí, aby daná proměnná nebyla realizována pouze na úrovni registrů CPU, ale před a po každém použití byla načtena/uložena do paměti.

Příklad

```
volatile int i=0;

P0 {                          P1 {
  while ( i==0 );               if (i==0) i++;
}                             }
```

Nestálé proměnné – volatile

Použití klíčového slova

- Umístěno před nebo za datový typ v definici proměnné.
- Rozlišujeme
  - nestálou proměnnou:
    ```
    volatile T a
    T volatile a
    ```
  - ukazatel na nestálou proměnnou:
    ```
    volatile T *a
    ```
  - nestálý ukazatel na nestálou proměnnou:
    ```
    volatile T * volatile a
    ```

Případy, kdy je nutné použít volatile:

- Proměnná sdílená mezi souběžnými vlákny/procesy.
- Proměnná zastupující vstup/výstupní port.
- Proměnná modifikovaná procedurou obsluhující přerušení.

Přehnaná důvěra ve volatile

Příklad

```
volatile int i=0;

Proc {                          main {
  for (int j=0;                   run_thread Proc as T1;
       j<100000;                  run_thread Proc as T2;
       j++)                       wait_on T1;
    i=i+1;                        wait_on T2;
}                                 print i;
                                }
```

Přehnaná důvěra ve volatile – pokračování

Výstup

- Na systému s jednou výkonnou jednotkou výstup vždy 200000.
- Na paralelních architekturách výstup často menší než 200000.

Zdůvodnění

- Přičtení není realizováno jednou instrukcí, riziko proložení vláken.
- Zápisy hodnot do paměti (do cache) nejsou prováděny v okamžiku zpracování instrukce, ale jsou odkládány a shlukovány.
- Přesný popis toho, jak procesor manipuluje se zápisy do paměti je součástí specifikace procesoru, jedná se o tzv. **paměťový model**.

Přehnaná důvěra v mentální model

POZOR!

- Pořadí zápisů do paměti dle vykonávané posloupnosti instrukcí procesoru nemusí korespondovat se skutečným pořadím zápisu hodnot do paměti.
- Paměťový model procesoru garantuje korektnost pouze pro sekvenční programy.

Důsledek

- Posloupnost přiřazení hodnot různým sdíleným proměnným provedená v jednom vlákně nemusí korespondovat s pořadím změn hodnot těchto proměnných v jiném vlákně.

Největší "skrčka" paralelního programování

Příklad

```
volatile int i=0;
volatile int * volatile p=0;

P0 {                          P1 {
  ...                           ...
  while (i==0);                 p = new int;
  (*p)=5;                       i=1;
  ...                           ...
}                             }
```

Problém

- V okamžiku přístupu na adresu odkazovanou ukazatelem `p` může mít `p` hodnotu 0.

Pozorování

- Bez nějakého dalšího opěrného bodu na HW úrovni je programování paralelních systémů téměř nemožné.

Paměťová bariéra

Co je to

- HW primitivum pro synchronizaci stavu paměti a stavů procesorů v daném místě programu.
- Na soudobých procesorech realizované instrukcí `mfence`.

Přesný popis

- Na hardwarové úrovni provede serializaci všech *load* a *store* instrukcí, které se vyskytují před instrukcí `mfence`. Tato serializace zajistí, že efekt všech instrukcí před instrukcí `mfence` bude globálně viditelný pro všechny instrukce následující za instrukcí `mfence`.

Realizace

- Není součástí vyšších programovacích jazyků.
- Různé překladače dávají programátorovi jisté možnosti.
  - GCC: `__sync_synchronize()`
  - Intel(R) C++ Compiler: `void_mm_mfence(void)`

HW podpora pro atomické instrukce

Fakta

- Paměťová bariéra neřeší problém atomických instrukcí jako jsou TEST-AND-SET, COMPARE-AND-SWAP, atd.
- Instrukce výše zmíněného typu jsou však pro účely efektivního paralelního programování velmi vhodné.

Další HW podpora

- Alpha, Mips, PowerPC, ARM: instrukce typu LL/SC
- x86 architektura
  - `lock` – následující (do paměti zapisující) instrukce proběhne atomicky a její efekt bude ihned globálně viditelný
  - `XCHG` – prohodí obsah registru a paměťového místa (obsahuje z definice prefix `lock`)

Realizace atomických instrukcí na úrovni kódu

Možnost 1: Jazyk symbolických adres

```
1 int test_and_set(volatile int *s){
2   int r;
3   __asm__ __volatile__(
4     "xchgl %0, %1 \n\t"
5     : "=r"(r), "m"(*s)
6     : "0"(1), "m"(*s)
7     : "memory");          ← paměťová bariéra
8  return r;}
```

Možnost 2: Zabudované funkce překladače (GCC $\geq$ 4.1)

- `type __sync_val_compare_and_swap (...)`
- `type __sync_fetch_and_add (...)`

Možnost 3: Součást programovacího jazyka

- C++ rev. 11, Java, ...