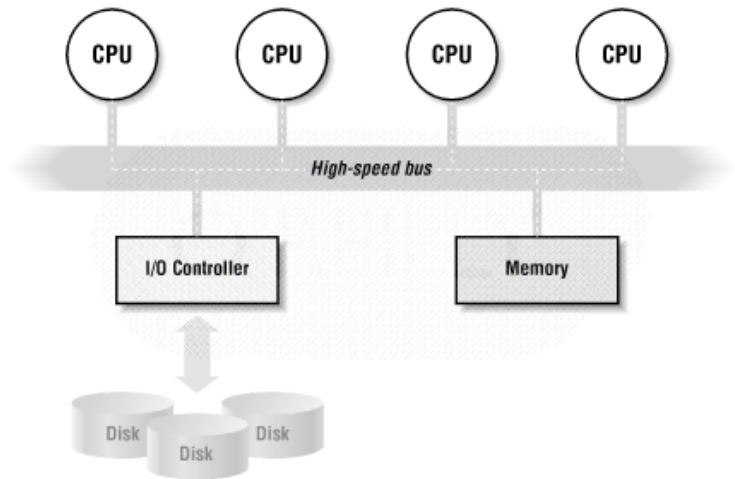


IB109 Návrh a implementace paralelních systémů

Programování v prostředí se sdílenou pamětí

Jiří Barnat

HW model prostředí se sdílenou pamětí



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

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 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.
- Jádra sdílí datovou cestu do paměti.

Budoucnost

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

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 vylučné.
- 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.

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.

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         )
```

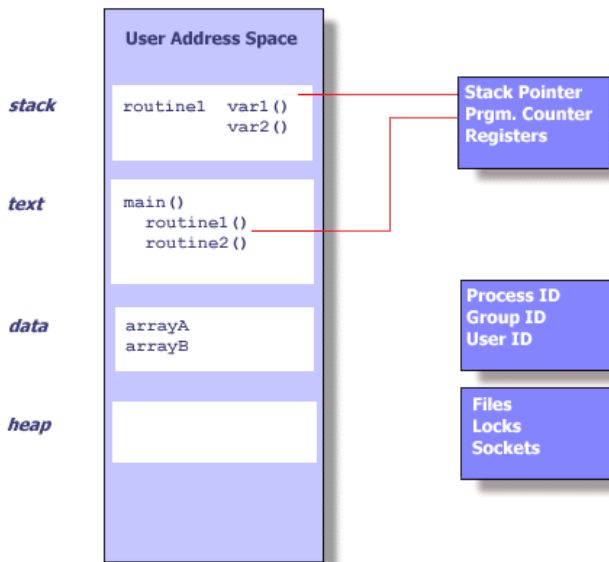
Proces

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

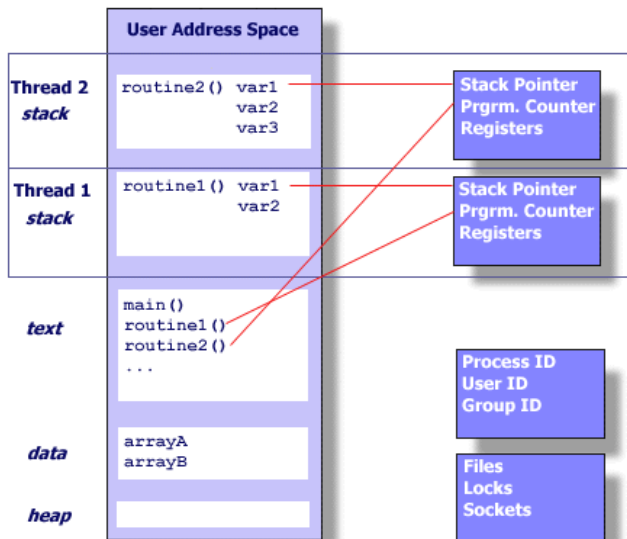
Vlákna mají privátní

- Zásobník
- Registry
- Frontu signálů

Proces v operačním systému



Vlákna v rámci procesu



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

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.

Související pojmy

- Cache line – atomický paměťový blok uložený v cache.
- “Hit ratio” – číslo vyjadřující úspěšnost obslužení 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)).

Specifikace

- Paralelní systém s více procesory
- Program s několikanásobně vícero vláknů
- Pole hodnot `int pole[nr_of_threads]`
- Vlákná 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

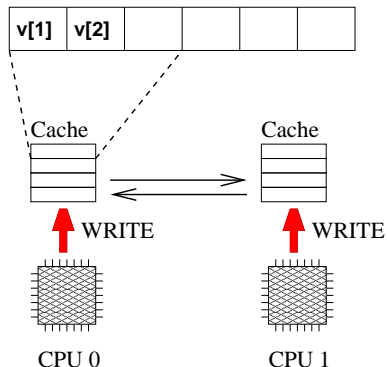
- Která implementace bude pomalejší a proč?

False sharing

```
typedef struct {  
    long long iter;  
  
} thread_private_data_t;
```

```
thread_private_data_t v[16];
```

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

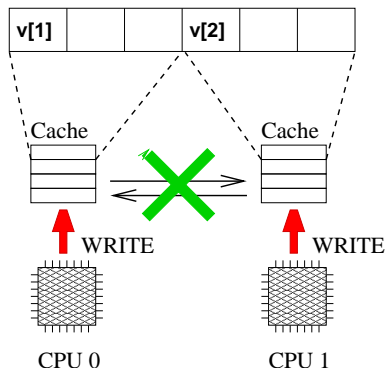


False sharing

```
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 ++;  
    ...  
}
```

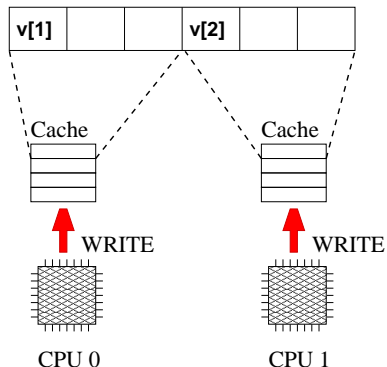


False sharing

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

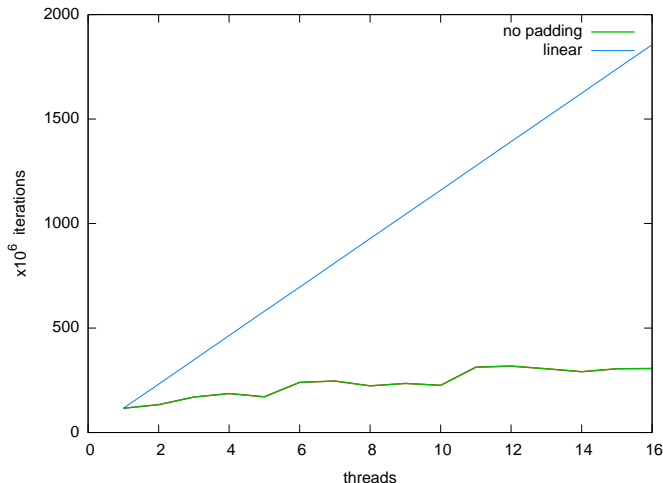
```
thread_private_data_t v[16];
```

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



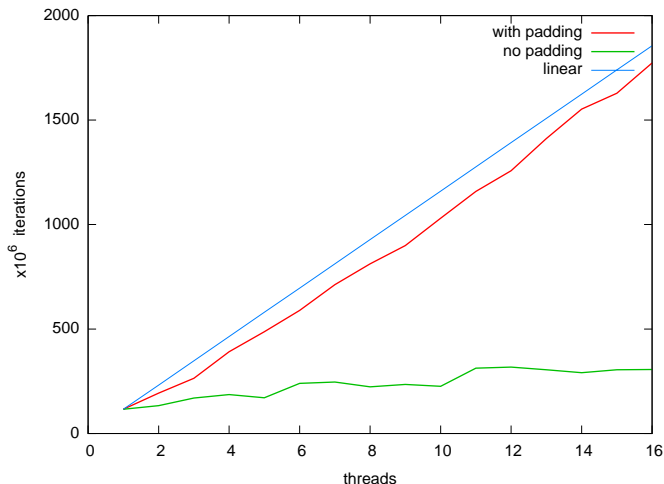
False sharing

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



False sharing

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



POZOR!

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

Důsledek

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

Pozorování

- Udržování koherence cache paměti 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 {  
    while ( i==0 );  
}
```

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

Problém

- Vlákno P0 neskončí, neboť nezaznamená změnu proměnné `i`.
- Může záviset na stupni optimalizace překladače, například
 - chyba se neprojeví při překladači pomocí `g++ -O0`
 - chyba se projeví při překladači 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 {  
    while ( i==0 );  
}
```

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

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.

Příklad

```
volatile int i=0;
```

```
Proc {  
    for (int j=0;  
        j<100000;  
        j++)  
        i=i+1;  
}
```

```
main {  
    run_thread Proc as T1;  
    run_thread Proc as T2;  
    wait_on T1;  
    wait_on T2;  
    print i;  
}
```

Výstup

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

Zdůvodnění

- 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**.

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.

Příklad

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

P0 {
    ...
    while (i==0);
    (*p)=5;
    ...
}

P1 {
    ...
    p = new int;
    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.

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)`

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)

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 *ptr, type oldval, type newval, ...)
- type `__sync_fetch_and_add` (type *ptr, type value, ...)