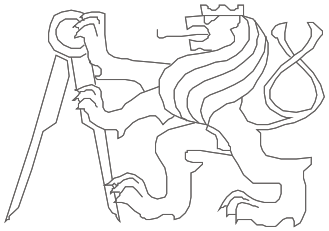
Pokročilé architektury počítačů
Konzistence paměti, Synchronizace

Michal Štepanovský
Pavel Píša



České vysoké učení technické v Praze
Fakulta elektrotechnická
Fakulta informačních technologií

Ver.1.00

Terminologie dnešního tématu podrobněji

- Pravidla pro provádění paměťových operací,
  - Paměťová koherence – minulá přednáška
    - Pravidla pro přístupy k paměťovým místům
  - Paměťová konzistence – dnešní přednáška
    - Pravidla pro všechny paměťové operace
- zajištění sekvenční konzistence,
- slabší modely paměťové konzistence
  - Konzistence dosahovaná s pomocí synchronizace, resp. synchronizačních instrukcí.

Definice koherentní paměti (obvyklá)

- Řekneme, že paměťový systém multiprocesorového systému je **koherentní**, jestliže výsledek jakéhokoli provádění programu je takový, že pro každé paměťové místo je možné sestavit myšlené sériové pořadí čtení a zápisů k tomuto paměťovému místu a platí:

- 1. Paměťové operace k tomuto paměťovému místu pro každý proces jsou provedeny v pořadí, ve kterém byly spuštěny tímto procesem.
- 2. Hodnoty vrácené každou operací čtení jsou hodnotami naposledy provedené operace zápis do tohoto paměťového místa vzhledem k sériovému pořadí.

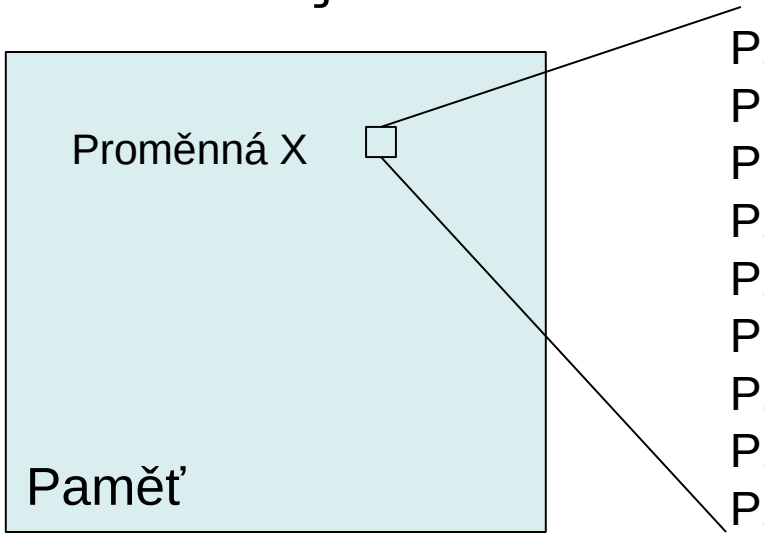
Koherence

Proces P1:

```
X=0;
if(X ==0) {
  y=fun();
  X = 1;
}
```

Proces P2:

```
X=0;
while(X ==0)
  { ; }
X = 2;
```



P2: X=0;
P1: X=0;
P1: read(X)
P2: read(X)
P2: read(X)
P1: X=1;
P2: read(X)
P2: read(X)
P2: X=2;

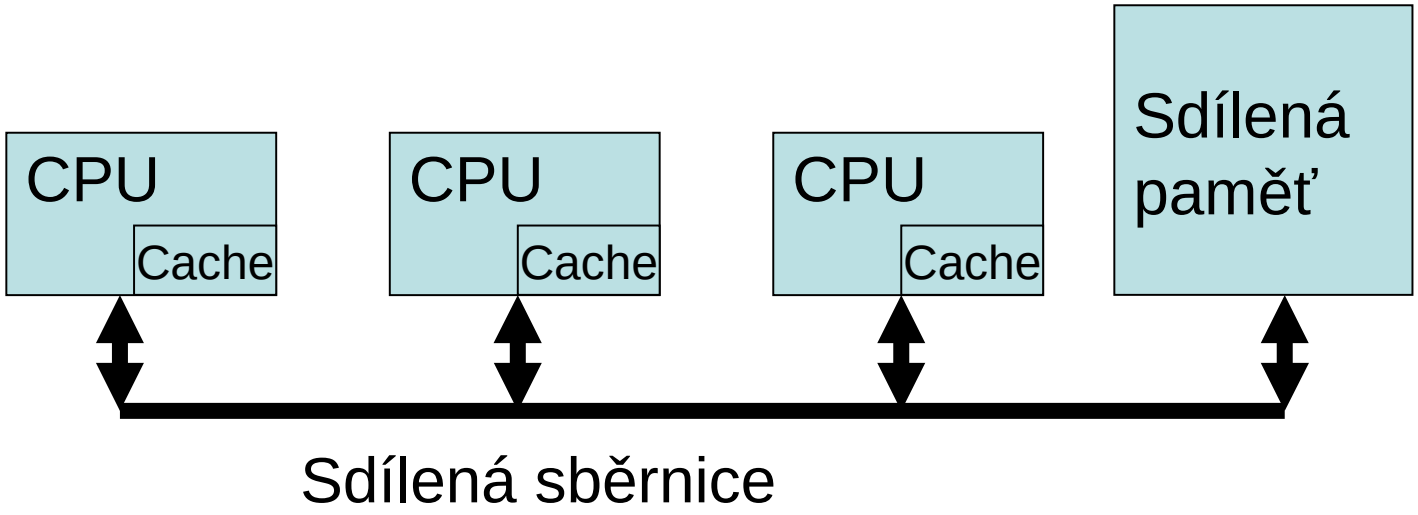
P2: read(X)
P2: X=0;
P1: X=0;
P1: read(X)
P2: read(X)
P1: X=1;
P2: read(X)
P2: read(X)
P2: X=2;

Máme jistotu, že když P2 uvidí X==1, bude funkce fun() volaná procesem P1 se všemi důsledky vykonána?

Konzistence

- **Konzistence** oproti koherenci specifikuje v jakém pořadí jednotlivé procesy spouštějí své paměťové operace, či jak se toto pořadí jeví ostatním procesům.
- Uvažuje sekvenční pořadí všech paměťových operací.
- **Koherence** uvažuje myšlené sekvenční pořadí **pouze vůči jednotlivým paměťovým místům**, nikoli mezi přístupy do různých paměťových míst.
- Konzistence definuje co je korektní chování sdílené paměti z pohledu čtení a zápisů

Na tomto MP systému simulujte provedení programu



Počáteční hodnoty oba procesy: `x=0, y=0`

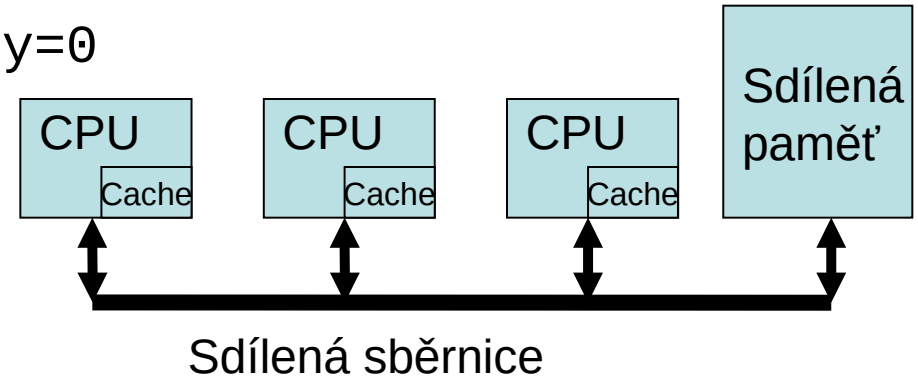
```
P1:          P2:
x = 1;       while(y==0) {;}
y = 1;       print(x);
```

Očekáváme, že *print(x) vytiskne 1.*

Na tomto MP systému simulujte provedení programu

Počáteční hodnoty oba procesy: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Možný scénář:

1. Procesor P2 nenajde y v cache a vydá požadavek na čtení z paměti. Nejprve ale musí získat sběrnici.
2. Procesor P2 spekulativně spustí čtení proměnné x – řádek „print(x)“. Tu najde v cache s hodnotou 0. Spekulace předpokládá y==1.
3. Procesor P1 získá sběrnici a provede zápis do proměnné x „x=1“. Nyní je v jeho cache označena jako M (MESI protokol) a zneplatněna v cache procesoru P2.
4. Procesor P1 získá sběrnici a zapíše y=1 do paměti.
5. Procesor P2 získá sběrnici a načte hodnotu y. To potvrdí „správnost“ spekulace a spekulativním instrukcím je umožněno dokončení.
6. Procesor P2 vytiskne 0.

Stačí koherence k ***rozumnému*** chování sdílené paměti?

- Proměnná *y* indikuje, že proměnná *x* byla změněna.
- Paměťová koherence ovšem nijak nespecifikuje v jakém pořadí jednotlivé procesy P1 a P2 spouštějí své paměťové operace (read, write) a nijak nespecifikuje v jakém pořadí uvidí P2 zápisy do x a y.
- Koherence pouze zajistí, že P2 se nakonec dozví nové hodnoty *x* a *y,* ale nijak nespecifikuje v jakém pořadí tyto nové hodnoty obdrží.

- **Proto ani na počítači s koherentním paměťovým systémem není vyloučeno, že P2 vytiskne starou hodnotu *x* (tj. 0).**

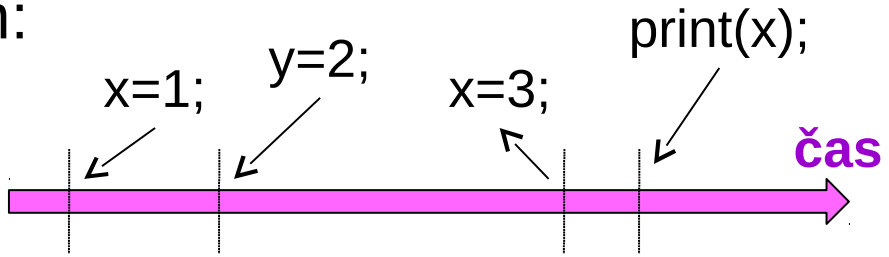
Koherence skrytých pamětí je nezbytná k zajištění datové (paměťové) konzistence v multiprocesorovém systému.

- Koherence – co vrátí operace čtení (jakou hodnotu)
- Konzistence – kdy bude zapsaná hodnota vrácena čtením.

Striktní konzistence

- Jednoprocesorový systém:

```
x = 1;
y = 2;
x = 3;
print(x);
```



(**Jakékoliv čtení z paměti z adresy *x* vrátí hodnotu uloženou při posledním zápisu na adresu *x*.**)

- Víceprocesorový systém:
  - podmiňuje existenci přesného globálního času ve všech uzlech a okamžité propagování změn
  - nerealistický až absurdní požadavek

Sekvenční konzistence (sequential consistency)

- Definice (Lamport, 1979): “Počítač je **sekvenčně konzistentní**, jestliže je výsledek provádění programu stejný, jako kdyby operace na všech procesorech byly provedeny v nějakém sekvenčním pořadí a operace každého jednotlivého procesoru se objevují v této posloupnosti v pořadí daném jejich programem.”

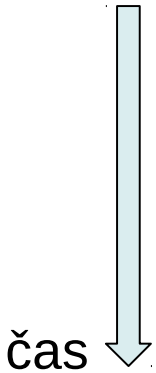
- Sekvenční konzistence je slabší model než striktní konzistence, avšak implementovatelná…
- Jestliže procesy běží na různých procesorech, je povoleno libovolné prokládání jejich instrukcí, avšak s podmínkou, že všechny procesy vidí stejné pořadí změn paměti. Změny nejsou propagovány okamžitě, je pouze zaručeno pořadí (následek nepředchází příčinu).

Sekvenční konzistence (sequential consistency)

Předpokládejme, že na začátku platí a=0, b=0, c=0.

```
P1:          P2:          P3:
  a=1;         b=1;         c=1;
  print(b,c);  print(a,c);  print(a,b);
```

Může to dopadnout takto:




```
a=1;
b=1;
c=1;
print(b,c);
print(a,c);
print(a,b);
```

Výstup: 111111

```
a=1;
print(b,c);
b=1;
print(a,c);
c=1;
print(a,b);
```

Výstup: 001011

atd.

Existuje 6! různých variací proložení instrukcí, ale ne všechny splní podmínku sekvenční konzistence.

6! / 8 = 90

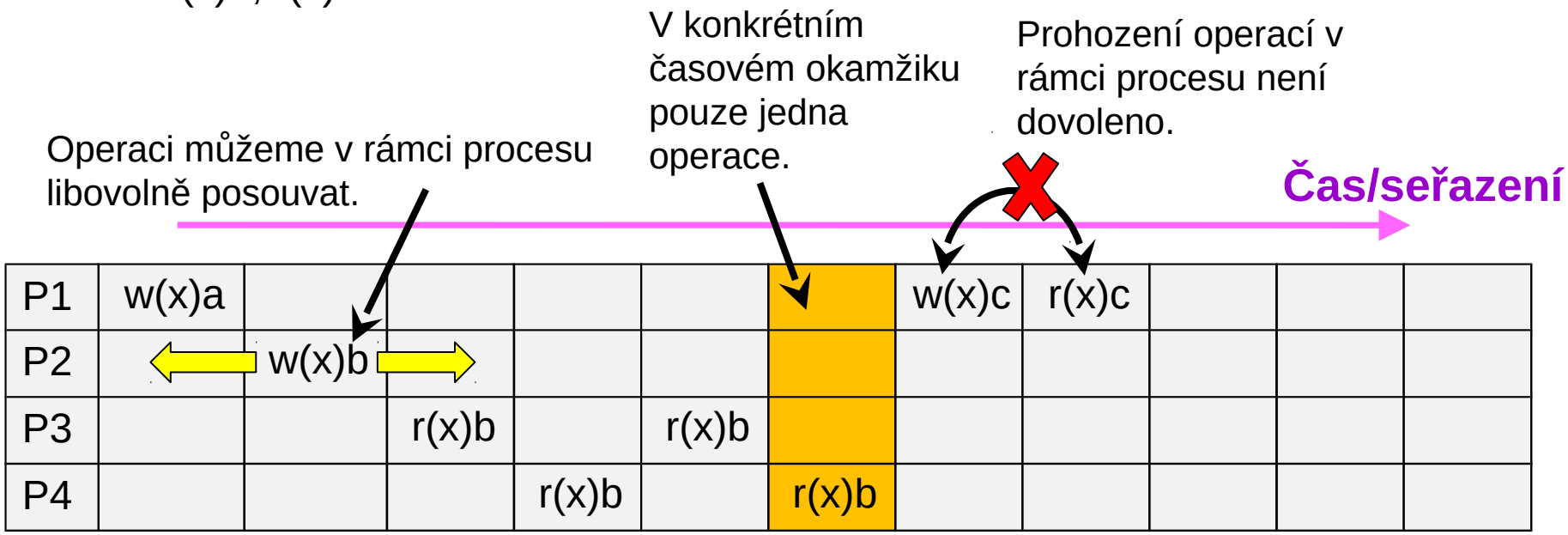
Sekvenční konzistence

Značení:

- Zápis hodnoty „**a**“ na adresu „**x**“: **w(x)a**
- Čtení z adresy „**x**“. Vrácená hodnota je „**a**“: **r(x)a**

Příklad – uvažujme 4 procesory (procesy), které běží paralelně a vykonávají:

- P1: w(x)a, w(x)c, r(x)?
- P2: w(x)b
- P3: r(x)?, r(x)?
- P4: r(x)?, r(x)?



Postačující podmínky pro zajištění SC

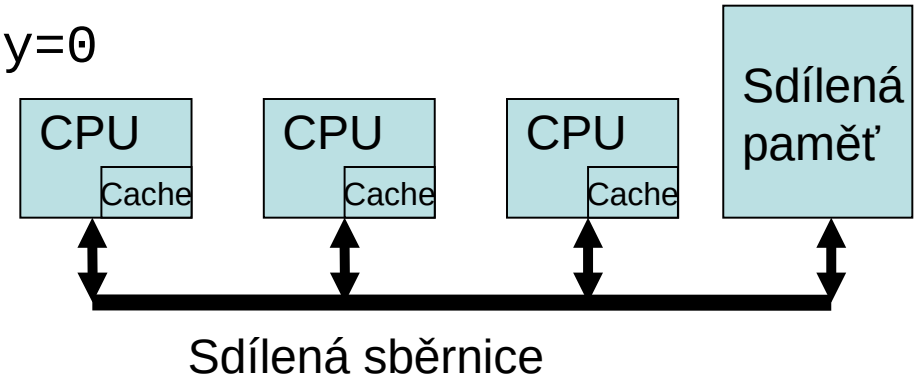
I. Každý procesor P(i) spouští paměťové operace v programovém pořadí.
II. Procesor P(i), který spustí operaci Write, nespustí další paměťovou operaci dříve, než se tato dokončí.
III. Procesor P(i), který spustí operaci Read, nespustí další paměťovou operaci dříve, než se tato dokončí a než se dokončí (vzhledem ke všem ostatním procesorům, nejenom vzhledem k procesoru Pi) operace Write, jejíž hodnotu vrací operace Read -> write atomicity

- Je důležité poznamenat, že compiler nesmí změnit pořadí paměťových operací – jak je obvyklé při optimalizaci programu na jednoprocesorový systém.

Na tomto MP systému simulujte provedení programu

Počáteční hodnoty oba procesy: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Předpokládejme sekvenční konzistenci

Narušení podmínky III.

Možný scénář:

1. ✓ Procesor P2 nenajde y v cache a vydá požadavek na čtení z paměti. Nejprve ale musí získat sběrnici.
2. ✗ ~~Procesor P2 spekulativně spustí čtení proměnné x – řádek „print(x)“. Tu najde v cache s hodnotou 0. Spekulace předpokládá y==1.~~
3. ✓ Procesor P1 získá sběrnici a provede zápis do proměnné x „x=1“. Nyní je v jeho cache označena jako M (MESI protokol) a zneplatněna v cache procesoru P2.
4. ✓ Procesor P1 získá sběrnici a zapíše y=1 do paměti. To zneplatní y v cache P2.
5. ✓ Procesor P2 získá sběrnici a načte hodnotu y. ~~To potvrdí „správnost“ spekulace a spekulativním instrukcím je umožněno dokončení.~~
6. ✓ Procesor P2 vytiskne ~~0~~ 1. Před tím si ale bude muset „x“ znovu načíst – bylo zneplatněno v kroku č.3.

Sekvenční konzistence a spekulace

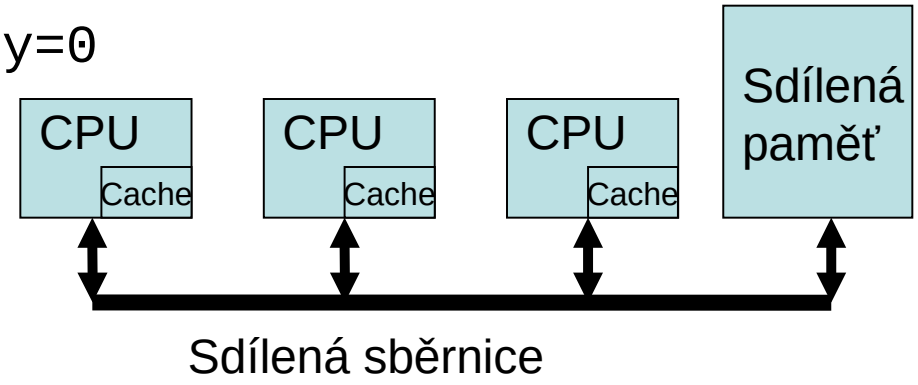
- Jak jsme viděli, zakázaní spekulací problém vyřešilo..

- Jiným řešením by bylo izolovat procesory pokud/dokud se nevyskytne sdílení dat – absence koherenčních aktivit indikuje, že procesor může přeuspořádat paměťové operace a tedy, povolit spekulaci
- Nicméně, stále je potřeba dodržet pořadí paměťových referencí z pohledu cache missů a slídění (snooping)
- Takže:
  - Umožníme spekulativní vykonávání
  - Všechny použité adresy během spekulace si budeme pamatovat
  - Pokud některá z těchto adres koliduje s koherenční aktivitou, zrušíme celou větev spekulace

Na tomto MP systému simulujte provedení programu

Počáteční hodnoty oba procesy: `x=0, y=0`

```
P1:        P2:
x = 1;     while(y==0){;}
y = 1;     print(x);
```



Předpokládejme sekvenční konzistenci

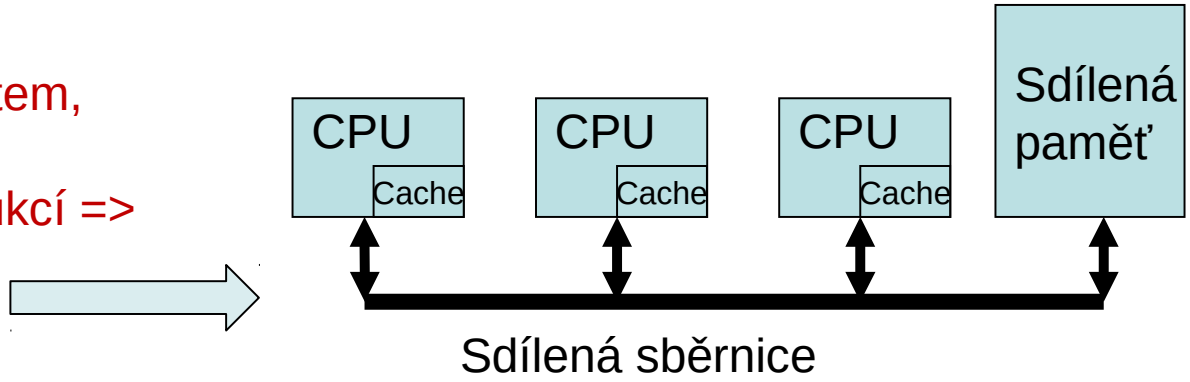
Možný scénář:

1. Procesor P2 nenajde y v cache a vydá požadavek na čtení z paměti. Nejprve ale musí získat sběrnici.
2. Procesor P2 spekulativně spustí čtení proměnné x – řádek „print(x)“. Tu najde v cache s hodnotou 0. Spekulace předpokládá y==1.
3. Procesor P1 získá sběrnici a provede zápis do proměnné x „x=1“. Nyní je v jeho cache označena jako M (MESI protokol) a zneplatněna v cache procesoru P2. To koliduje s adresou, kterou jsme si poznamenali během spekulace. Spekulace je zrušena.
4. Procesor P1 získá sběrnici a zapíše y=1 do paměti.
5. Procesor P2 získá sběrnici a načte hodnotu y.
6. Procesor P2 získá sběrnici, zažádá o x, v P1 přejde z M do S, zároveň do paměti a k P2, kde bude taky S. P2 vytiskne 1.

Zajištění konzistence v SMP se sdílenou pamětí?

- Definice (Lamport, 1979): “Počítač je **sekvenčně konzistentní**, jestliže je výsledek provádění programu stejný, jako kdyby operace na všech procesorech **byly provedeny v nějakém sekvenčním pořadí** a operace každého jednotlivého procesoru se objevují v této posloupnosti v pořadí daném jejich programem.”

Právě sběrnice je tím místem, kde se vytváří „nějaké“ sekvenční proložení instrukcí => serializace



Pokud tedy procesor (program) dodrží podmínky sekvenční konzistence a budeme mít paralelní systém se sběrnicí, **pak dosáhneme modelu sekvenční konzistence**. V procesu arbitrace (získání) sběrnice se rozhodne o pořadí paměťových operací – může to pokaždé dopadnout jinak.

Konzistence – otázka synchronizace – příklad

Problém:
Předpokládejme dva procesy P1 a P2 a sdílenou proměnnou A.
P1: A = A+1; P2: A = A +2;

Pokud by operace součtu byly atomické, pak A bude mít hodnotu A+3.
Jenomže:

```
P1:   load R1, A        P2:   load R1, A
      addi R1,R1,1            addi R1,R1,2
      store R1,A              store R2,A
```

Možná sekvence, která produkuje A+1 jako výsledek:

```
P1:   load R1, A
                        P2:   load R1, A
                              addi R1,R1,2
                              store R1,A
      addi R1,R1,1
      store R1,A
```

Toto proložení instrukcí **splňuje** model sekvenční konzistence, ale produkuje **„neočekávaný“ výsledek.**

Konzistence – otázka synchronizace – příklad

Problém:
Předpokládejme dva procesy P1 a P2 a sdílenou proměnnou A.
P1: A = A+1; P2: A = A +2;

Řešení:

1. **SW přístup.** Část kódu, která se stará o inkrementaci A „uchráníme“ před interakcí -> **vzájemné vyloučení, kritická sekce**.
   - Vzájemné vyloučení v sekvenčně konzistentním modelu paměti lze realizovat i pouze za pomoci atomických operací Read a Write.
   - **Dekkerův algoritmus** – první známé správné řešení – garantuje vzájemné vyloučení, neuváznutí v deadlocku a přidělení prostředků.

Petersonův algoritmus: počáteční hodnota wants_to_enter = { false, false}

```
P1: wants_to_enter[0] = true;
    turn = 1;
    while(wants_to_enter[1] && turn==1)
       ; // busy waiting
    // critical section
    A=A+1;
    // end of critical section
    wants_to_enter[0] = false;
```

```
P2: wants_to_enter[1] = true;
    turn = 0;
    while(wants_to_enter[0] && turn==0)
       ; // busy waiting
    // critical section
    A=A+2;
    // end of critical section
    wants_to_enter[1] = false;
```

Konzistence – otázka synchronizace – příklad

Problém:
Předpokládejme dva procesy P1 a P2 a sdílenou proměnnou A.
P1: A = A+1; P2: A = A +2;

Řešení:

2. **SW+HW přístup.** Část kódu, která se stará o inkrementaci A „uchráníme“ před interakcí -> **vzájemné vyloučení, kritická sekce**.
   - Čistě SW přístup je příliš komplikovaný. Raději bychom psali pouze:

```
while(!acquire(lock)) { čekací algoritmus }
výpočet nad sdílenými daty
release(lock)
```

   Protože několik procesů může chtít získat (acquire) zámek současně, proces získávání zámku musí byt atomický.
   - čekací algoritmus: ***busy waiting*** nebo ***blocking waiting***. Busy waiting – neustále zkouší získat zámek, blokující čekání – proces uspí sám sebe, uvolní procesor; vzbudí se až bude zámek uvolněn. Taktéž kombinace obou technik..

Konzistence – otázka synchronizace

Problém získání zámku lze v nejjednodušším případě řešit synchronizační sdílenou proměnnou (uloženou v paměti), která může mít hodnotu 0 (zámek je volný) nebo 1 (obsazen).
Problém získání zámku pak spočívá v otestování na 0 a nastavení na 1. Toto však musí být **nedělitelné**!

Potřebujeme tedy instrukci, která:

přečte, modifikuje a zapíše hodnotu do paměti bez interference.

ISA: ***test-and-set*** - všechny dnešní procesory ji buď podporují přímo nebo poskytují primitiva sestavení; je nejjednodušším případem atomické operace. Zapíše do paměti 1 (set), ale vrátí její starou hodnotu.

- zobecněním *test-and-set* je exchange-and-swap a compare-and-swap
- příklad: compare-and-exchange na x86: CMPXCHG s prefixem LOCK

Konzistence – otázka synchronizace

```
while(!acquire(lock)){ ; }
výpočet nad sdílenými daty
release(lock)
```

Použitím ***test-and-set*** by mohl náš program pro P1 vypadat takhle:

```
loop: test-and-set R2, lock  // testuje lock, hodnotu dá do R2 a nastaví lock=1
      bnz R2, loop           // pokud R2 není 0 vrátí se na loop
      load R1, A
      addi R1, R1, 1
      store R1, A
      store #0, lock         // uvolní zámek zapsáním 0.
```

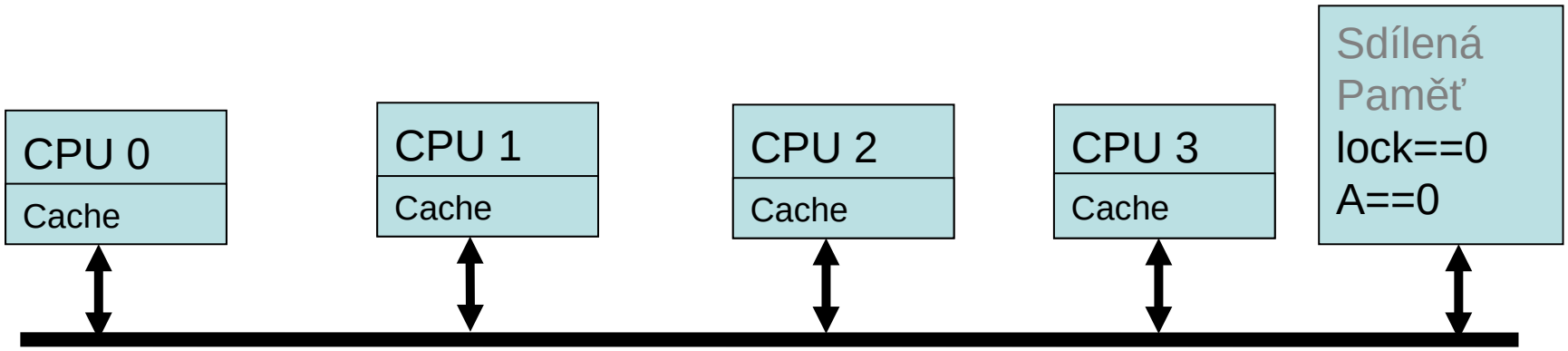
Instrukce **test-and-set R2, lock**, atomicky vykoná: **{load R2,lock; store #1,lock}**

Dalším typem atomické operace je fetch-and-xx operace (fetch-and-increment, fetch-and-add, fetch-and-store,…). Program pak bude vypadat takhle:

P1: fetch-and-inc A; P2: fetch-and-inc A;

MESI protokol a realizace zámku pomocí test-and-set

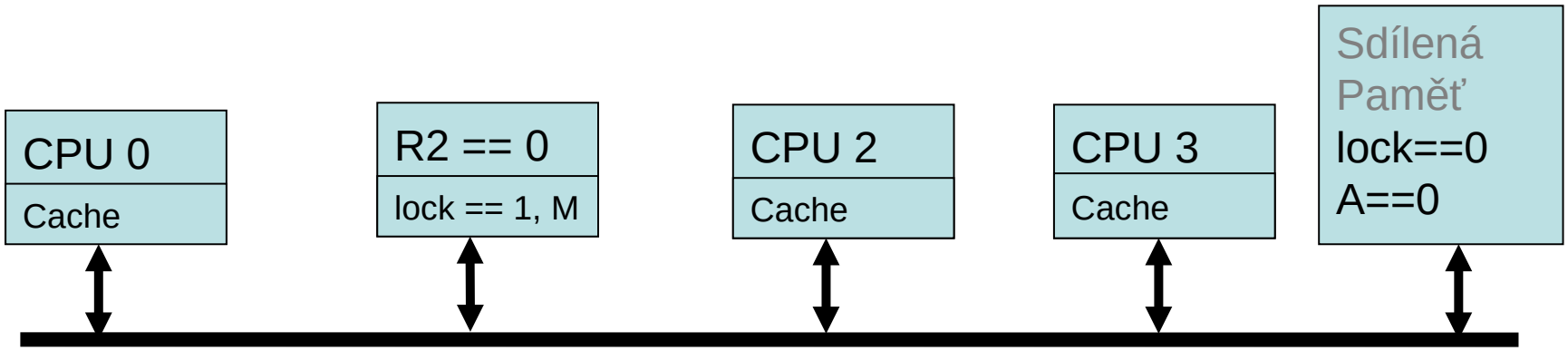
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;
store #1,lock}	store #1,lock}	store #1,lock}	store #1,lock}
bnz R2, L	bnz R2, L	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- Všechny CPU se pokusí vykonat instrukci test-and-set. Proto zažádají o sběrnici. Jeden z nich ji dostane.

MESI protokol a realizace zámku pomocí test-and-set

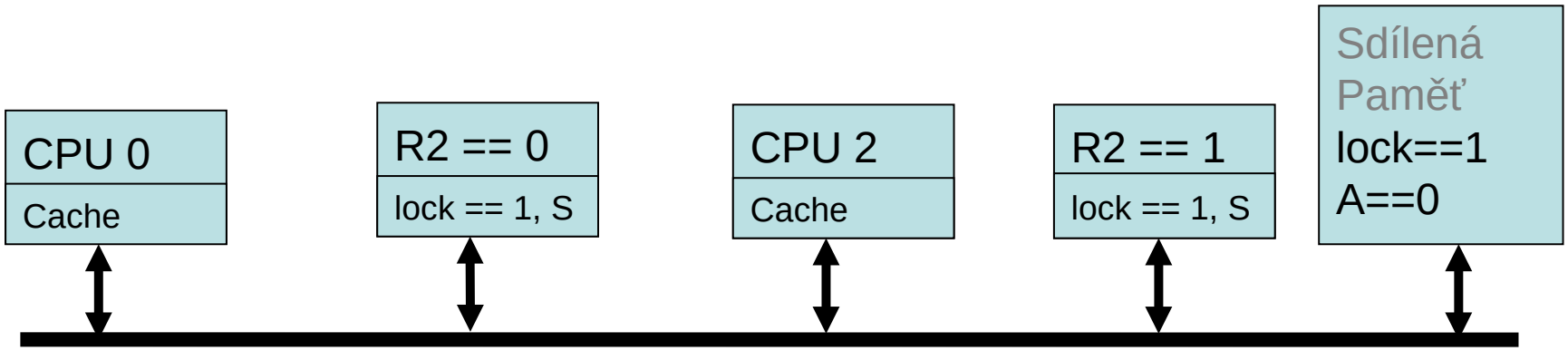
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	**L: {load R2,lock;**	L: {load R2,lock;	L: {load R2,lock;
store #1,lock}	**store #1,lock}**	store #1,lock}	store #1,lock}
bnz R2, L	bnz R2, L	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 1 získal sběrnici. Načte z paměti hodnotu „lock“ do R2 a přepíše jí na 1. To se projeví pouze v jeho cache. Řádek je označen M (modified).

MESI protokol a realizace zámku pomocí test-and-set

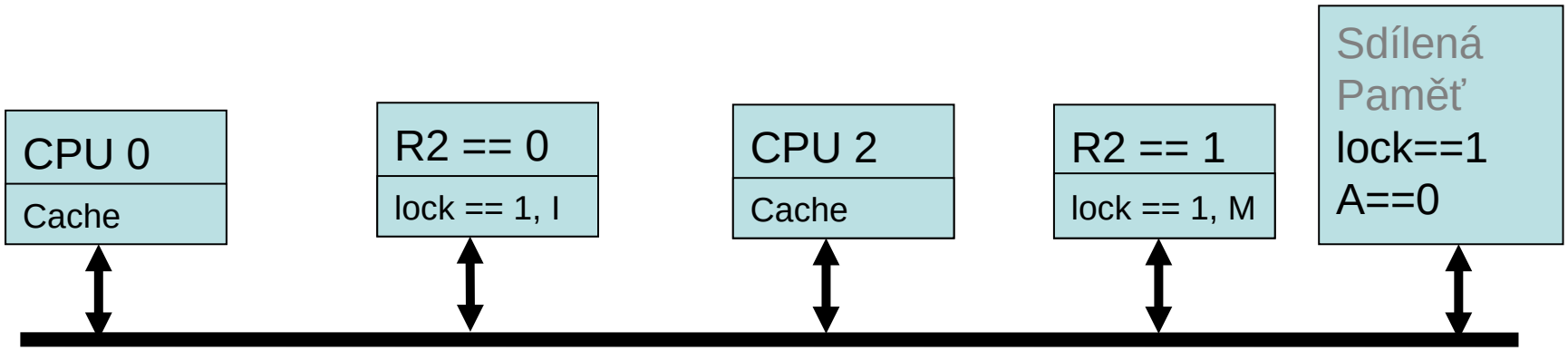
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	**L: {load R2,lock;** ⇦
store #1,lock}	store #1,lock}	store #1,lock}	**store #1,lock}**
bnz R2, L	**bnz R2, L**	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 3 získal sběrnici. Chce načíst z paměti hodnotu „lock“ do R2. Nicméně slídící CPU 1 tuto akci MemRead vidí, poskytne data, zapíše do paměti a přejde do S. Žádající CPU 3 obdrží data a přejde do S.

MESI protokol a realizace zámku pomocí test-and-set

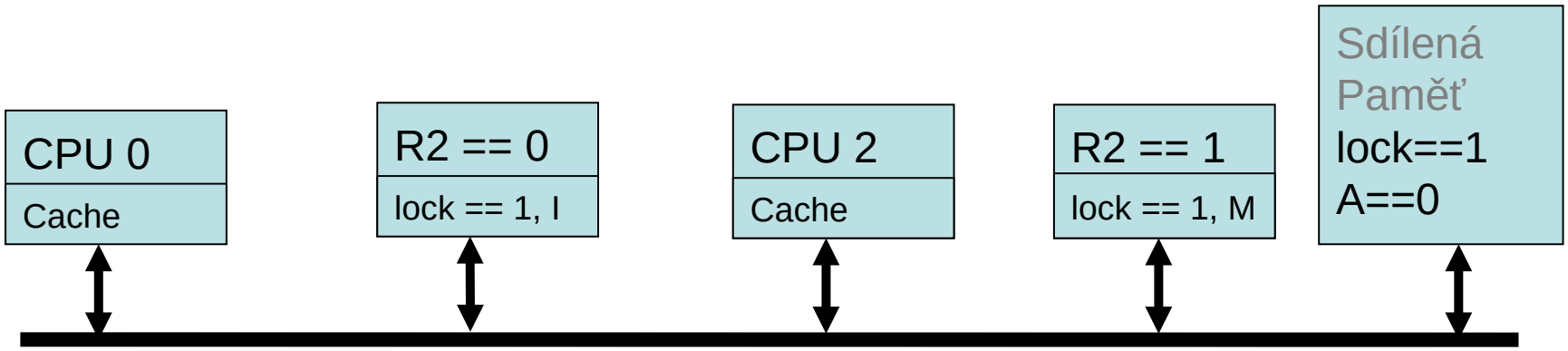
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	**L: {load R2,lock;**
store #1,lock}	store #1,lock}	store #1,lock}	**store #1,lock}** ⇦
bnz R2, L	bnz R2, L	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 3 stálé má sběrnici. Dalším krokem atomické akce test-and-set je zápis 1 do paměti na adresu lock. To vidí slídící CPU 1. Jde do stavu I (invalid), zatímco CPU 3 do M. CPU3 uvolní sběrnici.

MESI protokol a realizace zámku pomocí test-and-set

CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;
store #1,lock}	store #1,lock}	store #1,lock}	store #1,lock}
bnz R2, L	bnz R2, L	bnz R2, L	**bnz R2, L**
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	**addi R1, R1, 2**	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 0 získal sběrnici. Bude číst lock a zapisovat do něj 1. To se nakonec projeví invalidací ve všech cache-ích, pouze jeho vlastní bude mít M.
- CPU 3 testuje R2 a musí se vrátit na návěstí L a opět se pokusit získat sběrnici..

MESI protokol a realizace zámku pomocí test-and-set

Pozorování:

- Každý pokus získat zámek (úspěšné nebo ne) modifikuje jeho hodnotu a cache line je označena jako M.
- Důsledkem je zneplatnení cache line ve všech dotčených CPU.
- Neúspěšný pokus o získání zámku vede k dalšímu pokusu o získání zámku.
- S rostoucím počtem procesorů roste komunikace na sběrnici kvadraticky – pro čtení i zápis.

Vylepšení č.1:

- Pokud byl pokus o získání zámku neúspěšný, odložme další pokus o nějakou dobu – exponenciální, náhodná.

Vylepšení č.2:

- Vykonání instrukce test-and-set realizuje 2 transakce na sběrnici, z toho jedna zneplatní všechny ostatní cache lines. Proto: Pokusme se získat zámek pouze pokud je volný (jedna transakce). Navíc, cache lines zůstanou S ve všech čekajících CPU (do doby než dojde k uvolnění). Takže další dotazy na stav zámku před pokusem o jeho získání negenerují transakci MemRead – nezatěžuje se sběrnice.

Konzistence – otázka synchronizace

Problém:
Předpokládejme dva procesy P1 a P2 a sdílenou proměnnou A.
P1: A = A+1; P2: A = A +2;

Dalším alternativou je instrukční pár ***load-locked*** (ll) (nebo **load-link, load-linked, load-and-reserve**) a ***store-conditional*** (sc)

- Instrukce ll vrací aktuální hodnotu uloženou v paměti, sc pak uloží novou hodnotu jenom pokud nebyla nikým jiným modifikována – atomická operace je úspěšná – implementace vyžaduje *load address register (LAR)* a speciální *lock flag (LF)*..

```
loop:  ll R1, A      // A načte do R1, adresa A do LAR. LF=1;
       addi R1, R1, 1
       sc R1, A      // if(LF==1) ulož R1 do A;        R1=LF;
       bz R1, loop
```

- IBM PowerPC, DEC Alpha, MIPS, ARM, IA-64

Konzistence – otázka synchronizace

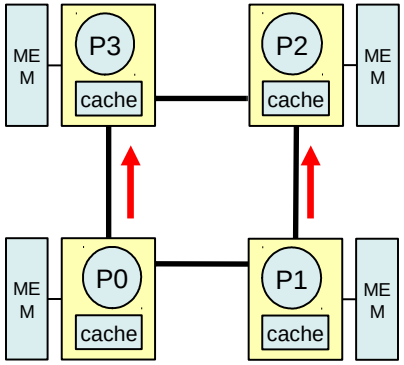
- Implementace ll a sc vyžaduje pouze minimální HW podporu: **registr adresy zámku** LAR a **příznak uzamčení** LF.
- Instrukce ll: nastaví LF a hodnotu LAR – tím je cache line **rezervována**
- Instrukce sc: pokud LF==1, pak ulož data do paměti. Vždy vrať LF.
- **Důležité:** Instrukce sc negeneruje žádnou transakci při neúspěchu = nezneplatňuje cache lines.
- **Přepnutí kontextu nebo interrupt:** vynuluj LF
- **Činnost řadiče cache:**
  - porovnávej adresy RWITM transakcí s adresou v LAR. Při shodě vynuluj LF.
  - Nedovol ale vlastní cache vyhodit cache line v důsledku nahrazování řádků cache (cache replacement policy – například LRU) mezi ll a sc. Nahrazení by nulovalo LF a mohlo by tak způsobit situaci, kdy se nikdy nepodaří sc projít. To by způsobilo zacyklení kódu mezi ll-sc -> **aktivní zablokování - livelock**. Jednoduchým řešením je nepoužívat memory-referencing instrukce = nečíst z paměti, nezapisovat do paměti mezi ll a sc a nedovolit out-of-order execution mezi ll a sc.

Diskuze

- Porovnejte strategie **test-and-set** a instrukční pár **ll-sc**. Která varianta zatěžuje sběrnici méně?

- Stačí paměťově koherentní systém k tomu, abychom splnili model sekvenční konzistence?

- Dnes se již sběrnice nepoužívá pro propojení procesorů / jader. Více žádostí tedy může být v běhu současně…
  - Co když 2 procesory současně provedou RWITM?
  - Co když odpovědi a žádosti dorazí k různým procesorům v různém pořadí?



Řešení:

Serializace (jinak synchronizace) požadavků (nutná kvůli koherenci a konzistenci) – jako na sběrnici … Ale sběrnici nemáme ….

Místo serializace: Domovský uzel – Home Node (viz minulá přednáška)

Diskuze

- V sekvenčním programu bude napsáno:

```
Instr.1:    load R1, A    // čtení proměnné A z paměti do R1
Instr.2:    load R2, B
Instr.3:    store R3, C   //hodnota R3 do C
Instr.4:    load R4, D
…
Instr.N:    store R5, A
```

Otázka č.1

- Vadí, pokud bychom dokončili (vykonali) instrukci č.2 před instrukcí č.1?

Otázka č.2

- Vadí, pokud bychom dokončili instrukci č.N před instrukcí č.1?

Otázka č.3

- Vadí, pokud bychom dokončili instrukci č.4 před instrukcí č.3?

Co již víme..

- **Load / Store instrukce** jsou zodpovědné za přesuny dat mezi pamětí a vlastními registry procesoru
- Procesor disponuje značně omezeným počtem registrů
- Kompilátor generuje tzv. ***spill code***, kterým dočasně odkladá používaná data do paměti aby uvolnil místo v registrech – právě pomocí load/store instrukcí

- Datové závislosti – RAW, WAR, WAW – mezi load/store instrukcemi odkazujícími se na tu samou adresu
- **Total ordering** – dodržení programového pořadí všech load/store instrukcí. Je to nezbytné?

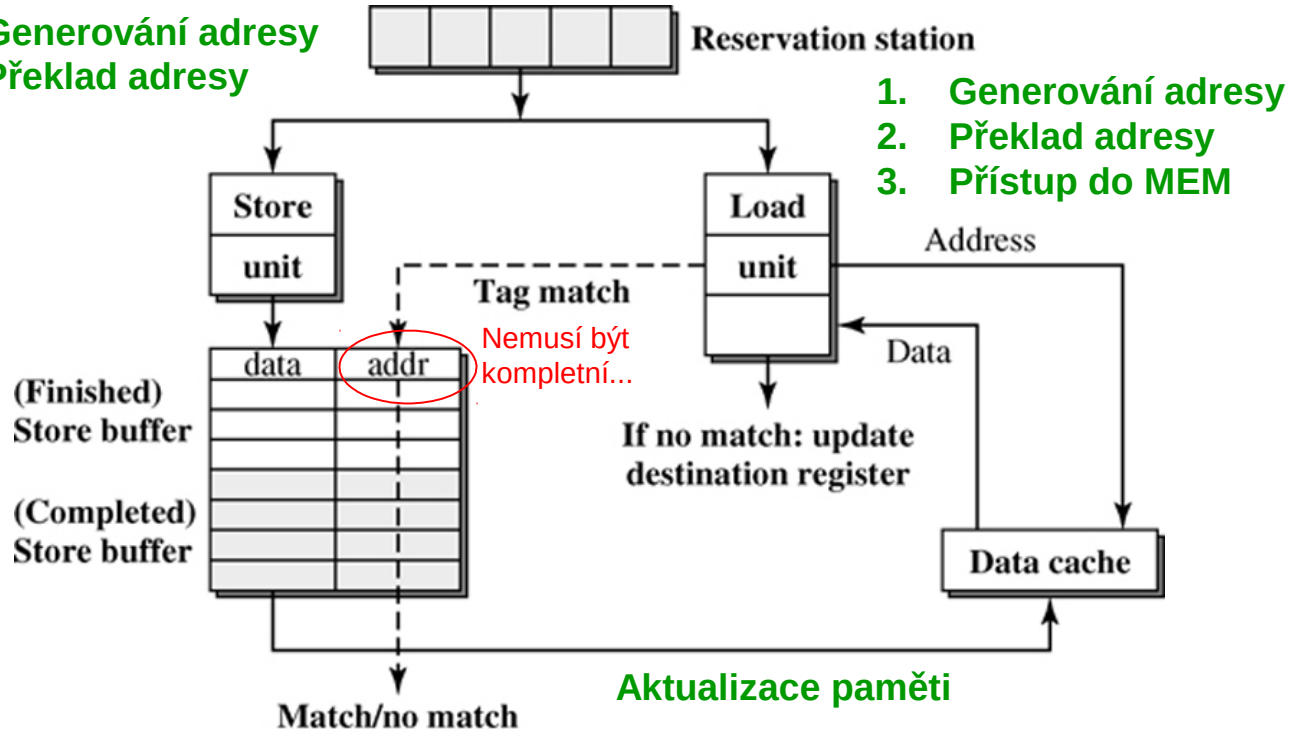
Model sekvenční konzistence

- **Podmínka sekvenční konzistence** klade jistá omezení na out-of-order vykonávání load/strore instrukcí
- Co když nastane exception?
- Stav paměti se musí odvíjet **dle sekvenčního pořadí** load/store instrukcí
- To znamená, že store instrukce musí být vykonány v programovém pořadí, nebo přesněji, že paměť musí být aktualizována tak, jakoby store instrukce byly vykonány v programovém pořadí
- Pokud budou **store** instrukce vykonány **v programovém pořadí**, máme garantováno dodržení WAW a WAR závislostí. Zůstává dodržet RAW závislosti…
- Load instrukce – out-of-order

Load forwarding a Load bypassing

Zatím předpokládejme vydávaní load/store instrukcí z rezervační stanice in order

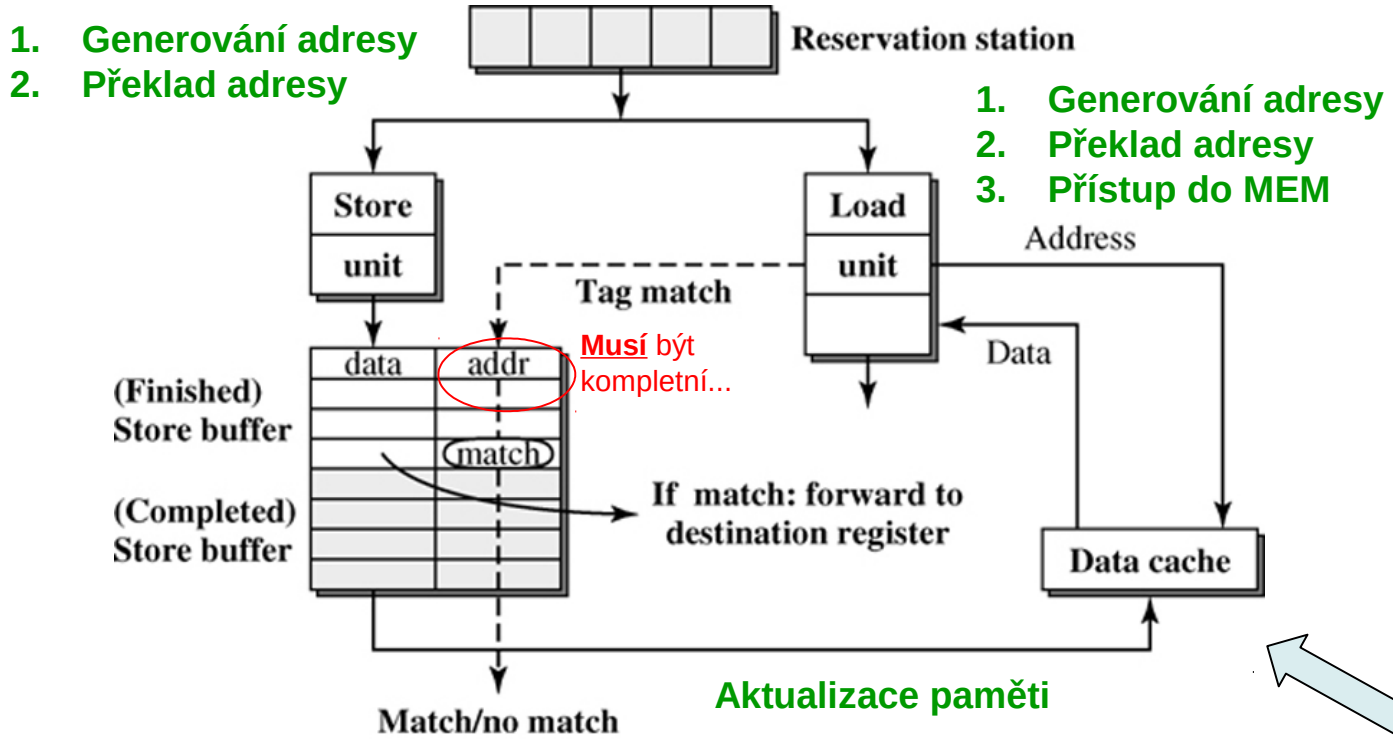
- ***Load bypassing*** umožňuje vykonat load před store, pokud jsou paměťove nezávislé. V opačném případě pak (pokud existuje): Load forwarding.



Load forwarding a Load bypassing

Zatím předpokládejme vydávaní load/store instrukcí z rezervační stanice in order

- ***Load forwarding*** přeposílá data z instrukce store do instrukce load aby bylo zaručeno dodržení RAW závislosti



Store: dispached, issued, finished, completed, retired

Load – if match: zahodí se přinesená data a berou se z Store buffru

Toto řešení (kompletní adresa) umožňuje obojí: load bypasing i load forwarding

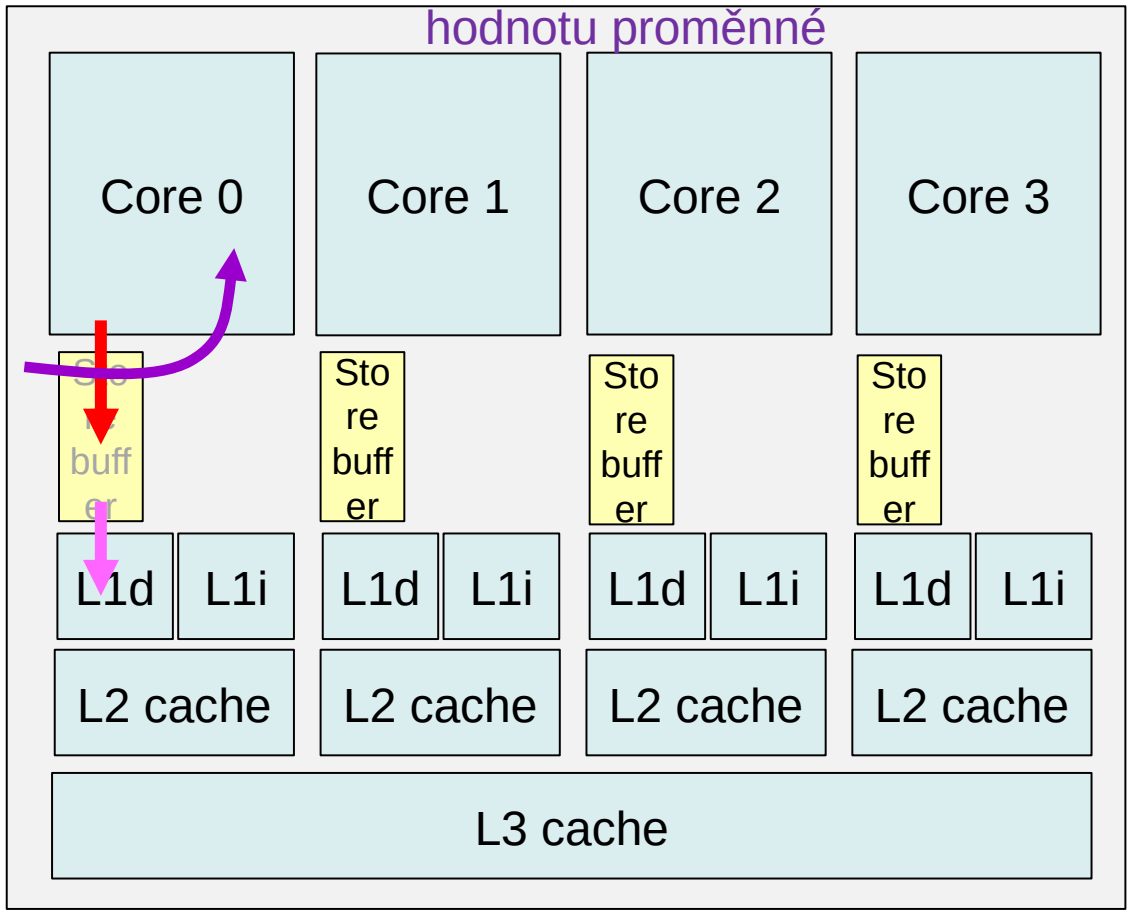
Store buffer

- Použití store buffru přináší **značné urychlení** běhu sekvenčního programu… Jenomže:

2. Load forwarding umožňuje jádru 0 načíst správnou (naposledy uloženou) hodnotu proměnné

1. Zápis do paměti se projeví ve Store buffru

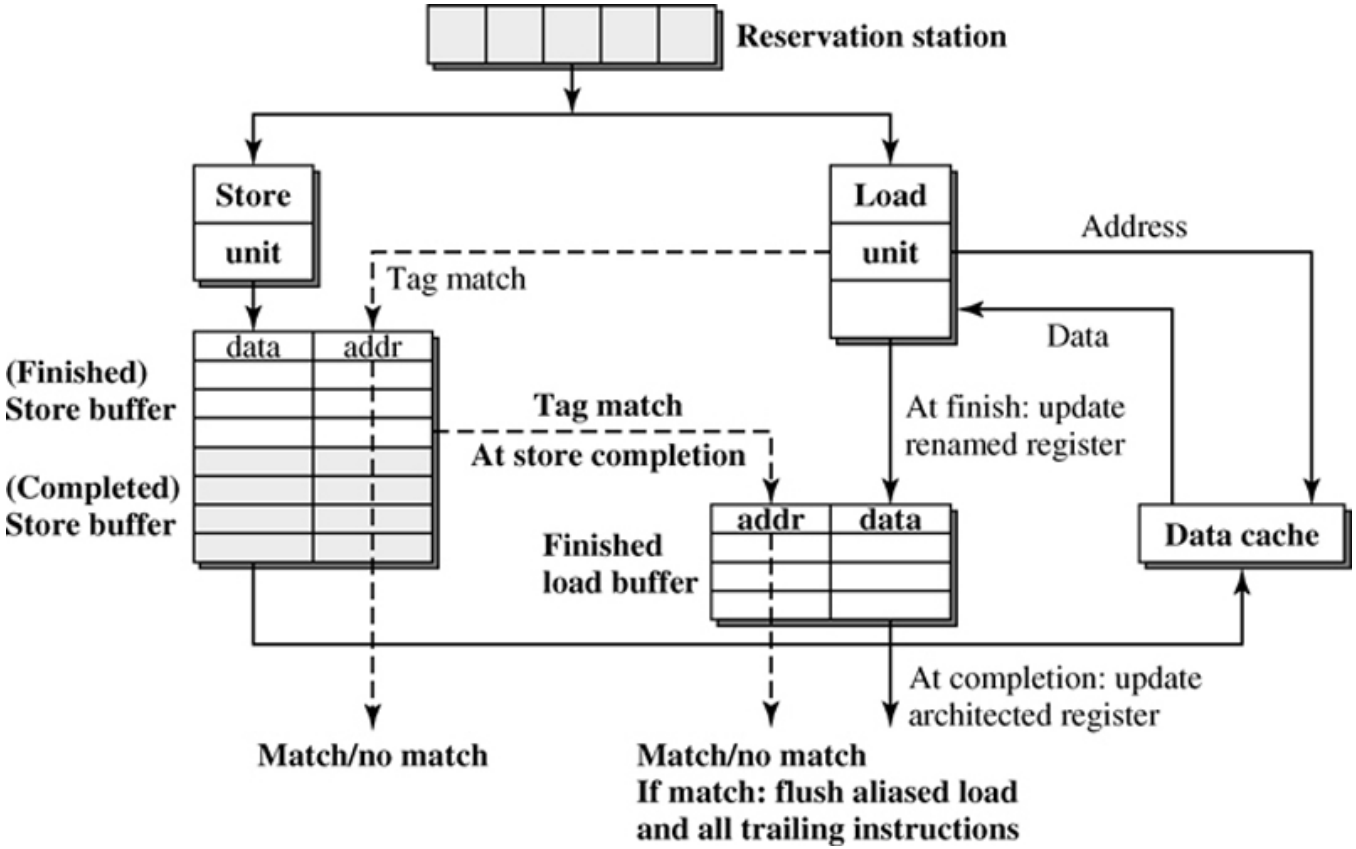
3. Po nějaké době dojde i k propagaci ze Store buffru do cache a začnou fungovat koherenční protokoly. Koherence (a tím i konzistence) systému byla tedy narušena.



Load forwarding a Load bypassing

- Pokud povolíme **vydávaní instrukcí (issuing) z rezervační stanice out-of-order**, pak se může stát, že instrukce load může být již vykonána, ale předcházející instrukce store se kterou má RAW není ještě v Store buffru (může být vykonávána, v rezervační stanici, nebo dokonce v paměti). Navíc nemáme ani informaci o její adrese (zda vůbec existuje RAW závislost).
- Řešení?
- Budeme předpokládat, že neexistuje závislost a tento předpoklad ověříme později... => **spekulativní vykonání**
- Spekulativní vykonávání podporuje ***Finished load buffer*** (Finish load queue)

Spekulativní vykonávání Load instrukcí



- Load instrukce je v **Finished load buffru** po ukončení vykonávání před dokončením
- Kdykoliv store přechází k dokončení, musí vykonat alias checking s položkami FLB. Žádný konflikt -> store je dokončena; Konflikt -> zrušení spekulace load instr.

Spekulativní vykonávání

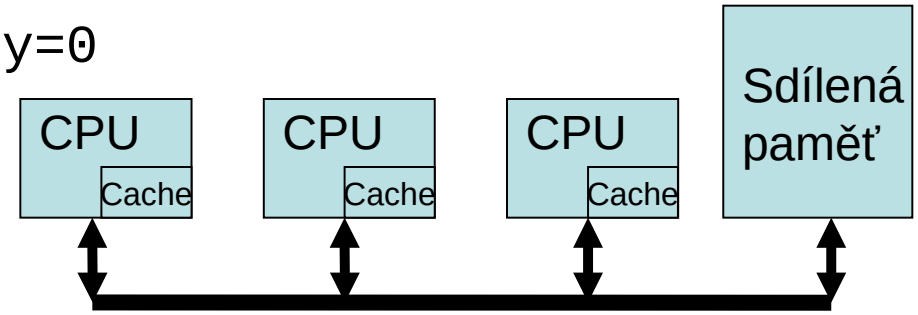
- Proč umožnit spekulace load instrukcí?
- Chceme vykonat load hned jak to jde – závisí na něm další výpočty
- Navíc, dřívější zavedení load může spustit *cache miss*
- A to může maskovat *cache miss penalty*

- Nicméně: V případě mylné spekulace – zrušení spekulativních instukcí (od load dále) – stálo nás to čas a prostředky, které mohly být využity lépe..
- Proto: ***Dependence prediction***
  V typických programech je závislost mezi store a load dobře predikovatelná
- ***Memory dependence predictor*** pak rozhodne zda začít spekulativně vykonávat load a další instrukce

Na tomto MP systému simulujte provedení programu

Počáteční hodnoty oba procesy: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Předpokládejme sekvenční konzistenci

Případný cache miss ale nesmíme propagovat ven…

Možný scénář:

1. Procesor P2 nenajde y v cache a vydá požadavek na čtení z paměti. Nejprve ale musí získat sběrnici.
2. Procesor P2 spekulativně spustí čtení proměnné x – řádek „print(x)“. Tu najde v cache s hodnotou 0. Spekulace předpokládá y==1.
3. Procesor P1 získá sběrnici a provede zápis do proměnné x „x=1“. Nyní je v jeho cache označena jako M (MESI protokol) a zneplatněna v cache procesoru P2. To koliduje s adresou, kterou jsme si poznamenali během spekulace. Spekulace je zrušena.
4. Procesor P1 získá sběrnici a zapíše y=1 do paměti.
5. Procesor P2 získá sběrnici a načte hodnotu y.
6. Procesor P2 získá sběrnici, zažádá o x, v P1 přejde z M do S, zároveň do paměti a k P2, kde bude taky S. P2 vytiskne 1.

Shrnutí

- Bylo věnováno značné úsilí jak urychlit běh aplikací na single-core procesorech – out-of-order, spekulace, store buffer před cache,…
- Tyto techniky mnohdy nejsou kompatibilní s modelem sekvenční konzistence

- Čeho se tedy vzdáme?

  Odpověď: Modelu sekvenční konzistence

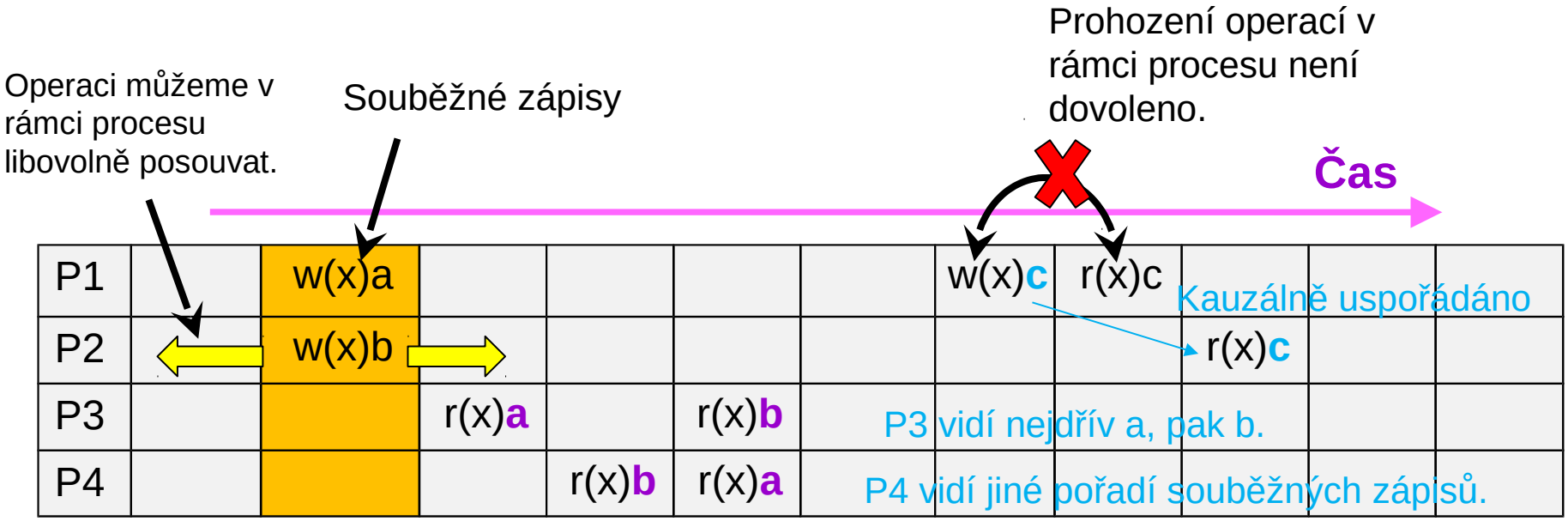
- Jak ale zabezpečíme, že programátor nedostane neočekávávané výsledky?

  Odpověď: Nabídneme mu jiný model konzistence – ten poskytne sekvenčně konzistentní pohled pouze v jistých okamžicích

  K tomu potřebujeme další instrukce.. => podpora HW

Kauzální konzistence *– causal consistency (Hutto, Ahamad, 1990)*

- Zápisy, které jsou potenciálně kauzálně vázané, musí být viděny všemi procesy ve stejném pořadí. Souběžné zápisy mohou být viděny v různém pořadí.
- Rozlišuje události, které jsou potenciálně závislé a které nikoliv
  - Čtení na daném P je kauzálně uspořádáno před zápisem (i na jinou adresu) – zapisovaná hodnota může záviset na přečtené hodnotě
  - Čtení je kauzálně uspořádáno za dřívějším zápisem na tu samou adresu, pokud čtení obdrželo data zapsaná daným zápisem
  - Zápisy na tu samou adresu daným P jsou kauzálně uspořádány tak jak byly provedeny
- **Slabší než sekvenční konzistence**



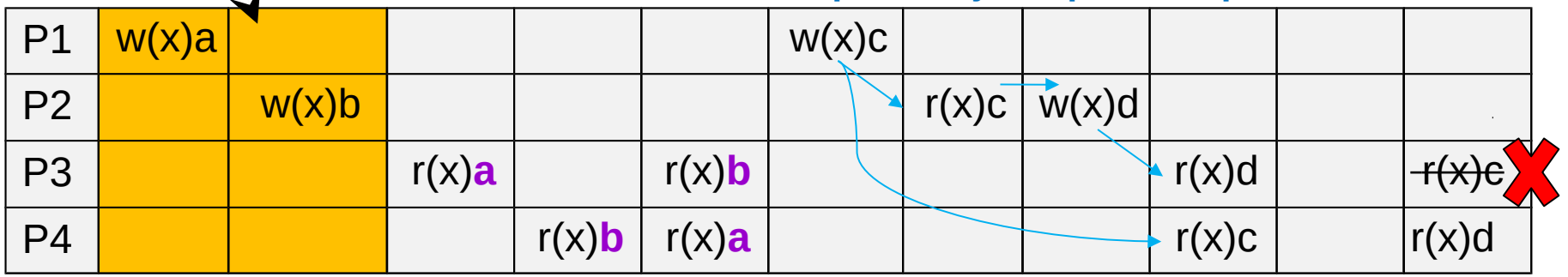
Další modely konzistence

Kauzální konzistence *– causal consistency (Hutto, Ahamad, 1990)*

- Zápisy, které jsou potenciálně kauzálně vázané, musí být viděny všemi procesy ve stejném pořadí. Souběžné zápisy mohou být viděny v různém pořadí.
- Rozlišuje události, které jsou potenciálně závislé a které nikoliv
  - Čtení na daném P je kauzálně uspořádáno před zápisem (i na jinou adresu) – zapisovaná hodnota může záviset na přečtené hodnotě
  - Čtení je kauzálně uspořádáno za dřívějším zápisem na tu samou adresu, pokud čtení obdrželo data zapsaná daným zápisem
  - Zápisy na tu samou adresu daným P jsou kauzálně uspořádány tak jak byly provedeny
- **Slabší než sekvenční konzistence**

Zápisy nejsou kauzálně vázány – souběžné zápisy

Zápis w(x)d u P2 je kauzálně vázán na dřívější r(x)c, který je kauzálně vázán na zápis w(x)c u P1. Proto tyto zápisy jsou rovněž kauzálně vázány a systém musí zajistit jejich pořadí: w(x)c < w(x)d. Takže poslední čtení u P3 r(x) nemůže vrátit c protože jsme před tím přečetli d.



Další modely konzistence

- ***PRAM konzistence*** *(PRAM consistency* = pipelined random access memory consistency*) = FIFO konzistence,* (*Lipton, Sandberg (1988)*
  - Zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, ve kterém byly prováděny, avšak zápisy různých procesů mohou být viděny různými procesy různě.
  - **Slabší než kauzální konzistence**

Zápisy různých procesorů mohou být viděny v různém pořadí

Princip kauzality neplatí. Opět se jedná o zápisy různých procesorů, proto P3 může vidět jiné pořadí než P4.

P1	w(x)a					w(x)c				
P2		w(x)b					r(x)c	w(x)d		
P3			r(x)**a**		r(x)**b**				r(x)d		r(x)c
P4				r(x)**b**	r(x)**a**				r(x)c		r(x)d

Připomenutí

- V sekvenčním programu bude napsáno:

```
Instr.1:   load R1, A    // čtení proměnné A z paměti do R1
Instr.2:   load R2, B
Instr.3:   store R3, C   //hodnota R3 do C
Instr.4:   load R4, D
…
Instr.N:   store R5, A
```

Otázka č.1

- Vadí, pokud bychom dokončili (vykonali) instrukci č.2 před instrukcí č.1?

Otázka č.2

- Vadí, pokud bychom dokončili instrukci č.N před instrukcí č.1?

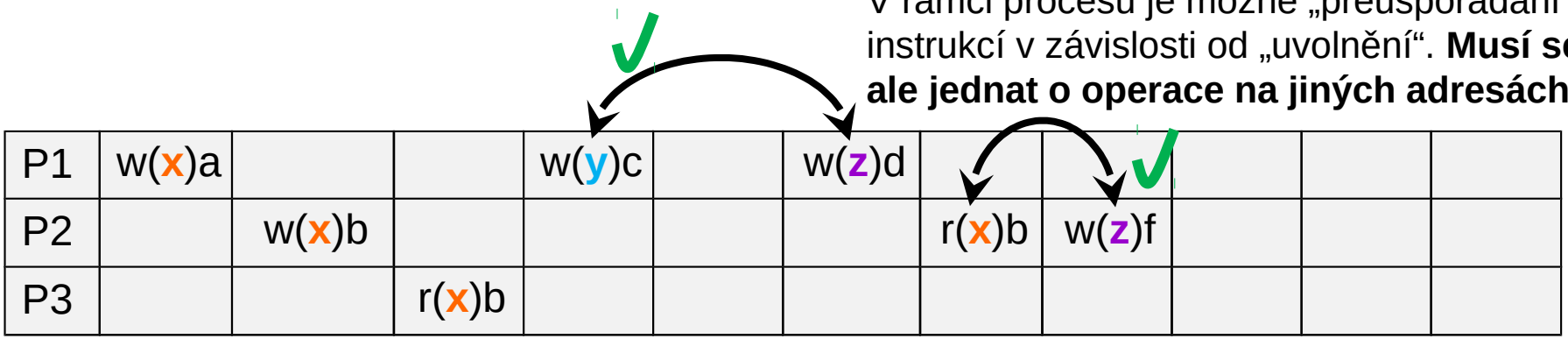
Otázka č.3

- Vadí, pokud bychom dokončili instrukci č.4 před instrukcí č.3?

Relaxed konzistence

- Sekvenční konzistence zachovává pořadí čtení a zápisů:
  1. W→R: zápis se musí dokončit před následujícím čtením
  2. R→R: čtení se musí dokončit před následujícím čtením
  3. R→W: čtení se musí dokončit před následujícím zápisem
  4. W→W: zápis se musí dokončit před následujícím zápisem
- Relaxed konzistence ulevuje na některém(ých) z těchto požadavků
- Navíc, můžeme ulevit i na požadavku jednotného sekvenčního proložení instrukcí viděného všemi procesory stejně, kdy:
  5. Procesor může vidět výsledek svého zápisu dřív než jej vidí ostatní
  6. Procesor může vidět výsledek cizího zápisu dřív než jej vidí ostatní

V rámci procesu je možné „přeuspořádání“ instrukcí v závislosti od „uvolnění“. **Musí se ale jednat o operace na jiných adresách.**



Další modely konzistence

Relaxed konzistence – Co přináší uvolnění?

- **W→R**: odstraňuje Write pryč z kritické cesty – překryv Write a následujícího Read „redukuje“ latenci paměti pro Write. (Write v koherentním NUMA systému není jenom zápis, je to i nalezení platného bloku – dotazování do home node, poslání invalidace všem ostatním, načtení bloku, atd.)
- **R→R a R→W:** neblokující cache – při read miss můžeme pokračovat v práci a nemusíme čekat až se miss obslouží – spekulativní vykonávání
- **W→W**: paralelismus na úrovni paměti
- **Čtení vlastního zápisu před ostatními**: Load forwarding – store buffer před cache, tzn. urychlení běhu programu
- **Čtení zápisu od cizího před ostatními**: čtení z vlastní paměti ještě před tím, než je změna propagována všem ostatním

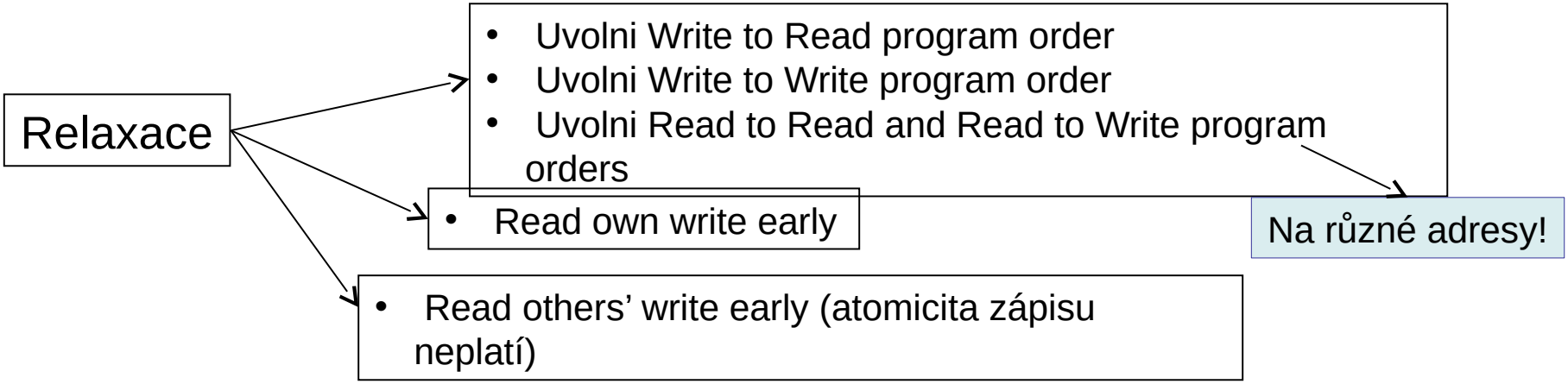
- Uvolnění tedy přináší možnost paralelního běhu. Vynucená serializace požadovaná sekvenční konzistencí je potlačena.

Relaxed konzistence

Mezi tyto modely řadíme:

- **Total Store Ordering** (TSO) – IBM 370: operace čtení může být dokončena před dřívějším zápisem na jinou adresu, ale čtení nemůže vrátit zapisovanou hodnotu do té doby než všechny procesory vidí zápis
- **Total Store Ordering** (TSO) – SPARC: operace čtení může být dokončena před dřívějším zápisem na jinou adresu. Čtení nemůže vrátit hodnotu zapsanou jiným procesorem do té doby než všechny procesory vidí zápis, ale procesor může vrátit hodnotu vlastního zápisu před tím než ji ostatní vidí
- **Processor Consistency** (PC): čtení může být dokončeno před tím než dřívější zápis (libovolným procesorem na libovolné místo) je viditelný všem, tzn. čtení některým procesorem může vrátit novou hodnotu, zatímco u jiných procesorů čtení ze stejné adresy vrací ještě starou hodnotu.
- **Partial Store Ordering** (PSO) – podobná k TSO. Rozdíl: PSO zachovává pouze pořadí zápisů na stejnou adresu, zápisy do různých míst můžou být mimo původní pořadí.
- A další…

Relaxed konzistence



	W->R	W->W	R->R,W	Čtení vlastního zápisu před ostatními	Čtení zápisu od cizího před ostatními
TSO – IBM 370	x				
TSO – SPARC: SPARC, IA-32, Intel64, AMD64	x			x	
PC	x			x	x
PSO	x	x		x	
Weak consistency: PowerPC, ARMv7, IA-64	x	x	x	x	x

Konzistence u IA-32 a Intel64

Intel Core i5, Core i7, Intel Xeon, Intel Core2 Extreme

- Čtení vzhledem ke čtení a zápis vzhledem k zápisu konkrétního procesoru nejsou přeuspořádány (vyjma speciálních long string store a string move zápisových operací) --tzn. R->R a W->W není uvolněno
- Zápis nepředchází dřívější čtení --tzn. R->W není uvolněno
- Čtení může předběhnout dřívější zápis na jinou adresu -- uvolnění W->R, Dekkerův algoritmus selže

P1:	**P2**:
X=1;	Y=1;
R1=Y;	R2=X;

Když počáteční hodnoty X=Y=0 může vrátit R1=0 a současně R2=0.

- Čtení nemůže předběhnout dřívější zápis na stejnou adresu
- Load-forwarding uvnitř procesoru je povolen – tzn. čtení vlastního zápisu před ostatními

P1:	**P2**:
X=1;	Y=1;
R1=X;	R3=Y;
R2=Y;	R4=X;

Když počáteční hodnoty X=Y=0 může vrátit R2=0 a současně R4=0.

Konzistence u IA-32 a Intel64

Intel Core i5, Core i7, Intel Xeon, Intel Core2 Extreme

- Zápisy jsou tranzitivně viditelné – zápisy, které jsou kauzálně vázány se jeví všem procesorům ve stejném pořadí

P1:	**P2**:	**P3**:
X=1;	R1=X;	R2=Y;
	Y=1;	R3=X;

Když počáteční hodnoty X=Y=0 nemůže vrátit R1=1, R2=1 a současně R3=0.

- Zápisy jsou viděny ve stejném pořadí všemi **ostatními** procesory – procesor, který zapisuje může vidět jiné pořadí

P1:	**P2**:	**P3**:	**P4**:
X=1;	Y=1;	R1=X;	R3=Y;
		R2=Y;	R4=X;

Když počáteční hodnoty X=Y=0 nemůže vrátit R1=1, R2=0, R3=1 a současně R4=0.

- Architektura IA-32 a Intel64 tedy splňuje konzistenci TSO - SPARC.

Jaké chování paralelních programů lze očekávat?

Příklad A:

P1:
```
A=1;
flag=1;
```

P2:
```
while(flag==0);
print(A);
```

Příklad B:

P1:
```
A=1;
B=1;
```

P2:
```
print(B);
print(A);
```

Příklad C:

P1:
```
A=1;
```

P2:
```
while(A==0);
B=1;
```

P3:
```
while(B==0);
print(A);
```

Příklad D:

P1:
```
A=1;
print(B);
```

P2:
```
B=1;
print(A);
```

Bude vykonání programu ve shodě se sekvenční konzistencí?

	Příklad A	Příklad B	Příklad C	Příklad D
TSO – SPARC	Ano	Ano	Ano	Ne
PC	Ano	Ano	Ne	Ne
PSO	Ne	Ne	Ne	Ne
Weak consistency	Ne	Ne	Ne	Ne

Za předpokladu, že kompilátor dodrží pořadí zapsaných příkazů… Počáteční hodnoty: A=flag=0.

Jak dosáhnout rozumného chování programu?

Použijeme **paměťovou bariéru** (memory barrier)

- Všechny datové operace PŘED bariérou se musejí dokončit
- Všechny datové operace ZA bariérou musejí čekat na dokončení bariéry
- Bariéry jsou viděny v programovém pořadí

Programátor tedy uvažuje oblasti, kde **paměťové operace nad sdílenými proměnnými můžou být libovolně přeuspořádány**. Ty oddělí bariérami.

- IA-32, Intel64 definuje tři typy bariéry: **sfence, lfence, mfence**
  - Sfence – všechny store před bariérou se musí dokončit před store za
  - Lfence – všechny load před bariérou se musí dokončit před load za
  - Mfence – všechny paměťové operace před bariérou se musí dokončit (být globálně viditelné) před vykonáním paměťové operace za bariérou
- PowerPC používá **sync**
- OpenMP používá direktivu **flush**

Jak dosáhnout rozumného chování programu?

Použijeme **paměťovou bariéru** (memory barrier)

Garantuje uspořádání

Urychluje propagaci flag-u

Příklad A:

P1:

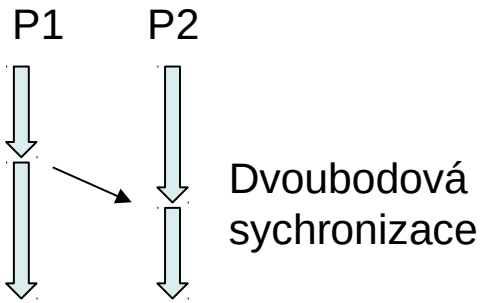
```
A=1;
#pragma omp flush
flag=1;
#pragma omp flush
```

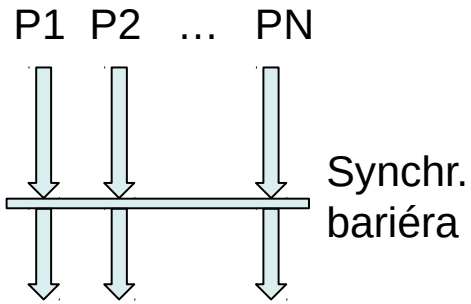
P2:

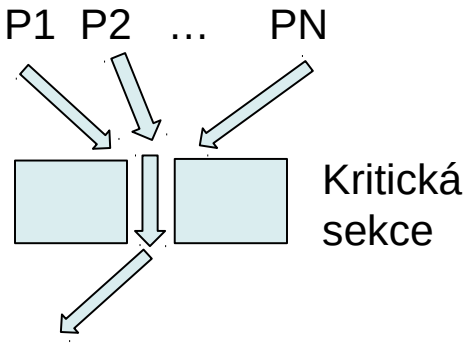
```
while(flag==0);
#pragma omp flush
print(A);
```

- Je garantováno, že P2 bude číst A s hodnotou 1, dokonce i když paměťové operace P1 před flush-em a za flush-em jsou přeuspořádány hardwérem nebo kompilátorem.
- Implementace paměťové bariéry musí zajistit, že sdílené proměnné (thread-visible) jsou viditelné všem vláknům/procesorům za tímto bodem, tzn. kompilátor musí zajistit, že hodnoty z registrů procesoru jsou zapsány do paměti (je generována operace Write), procesor vyprázdní write-buffry, atd.
- **Paměťová bariéra tedy zajišťuje sekvenčně konzistentní pohled na paměť pouze v definovaných okamžicích – musí se ale jednat o vzájemně sladěnou akci všech zúčastněných vláken/procesorů.**

Typy synchronizačních událostí







Z pohledu paralelního programování rozlišujeme:

- **Dvoubodová synchronizace**: zajištění předání dat mezi dvěma procesy (vlákny). Jeden z procesů může eventuálně pokračovat v činnosti bez nutnosti čekání – viz předchozí slajd
- **Synchronizační bariéra:** všechny procesy dané skupiny procesů musí v tomto bodě čekat až dorazí poslední, teprve pak mohou zase pokračovat. (Pozor: Nezaměňovat s pojmem paměťová bariéra.)
- **Vzájemné vyloučení** – kritická sekce: Pouze jednomu procesu v čase je umožněno projít označeným kusem kódu.

Typy synchronizačních událostí - OpenMP

Dvoubodová synchronizace:

Již bylo ukázáno

Synchronizační bariéra:

```
…
#pragma omp barrier
…
```

Kritická sekce:

```
#pragma omp critical
{
    … // A = A+1;
}
```

Poznámka: Operace **flush** (paměťová bariéra) je použita při dvoubodové synchronizaci a generována při použití synchronizační bariéry, při vstupu do kritické sekce i výstupu z kritické sekce – důležité pro zajištění sekvenčně konzistentního pohledu na paměť v daném místě.

Dříve probírané instrukce pro zajištění atomicity (test-and-set, instrukční pár ll-sc) spolu s instrukcí paměťové bariéry (zajištění sekvenčně konzistentního pohledu v daném okamžiku) jsou základem pro implementaci výše uvedených synchronizačních událostí.

Omezení synchronizace jen na ty přístupy, kdy je potřeba

- Pro efektivní využití procesorových jader jsou modely využívající paměťové bariéry týkající se všech přístupů do paměti příliš omezující
- Snaha definovat paměťový model, ve kterém se přesně na úrovni zdrojového kódu označí, jestli musí být daná operace atomická jen k dané proměnné (relaxed), slouží jako potvrzení platnosti dat (release), kontrole platnosti dat (consume), převzetí kontroly/zámku (acquire), kombinovaný (acq-rel) a verze operací se zaručením kompletní synchronizace (seq_cst). Pouze poslední, nejdražší (Sequentially-consistent ordering) odpovídá dříve probíraným paměťovým bariérám.
- Nejpropracovanější je tento model pravděpodobně v standardu jazyka C++11

  http://en.cppreference.com/w/cpp/atomic/memory_order

Ticket-lock s využitím C++ paměťového modelu

- Ticket-lock je implementace spinlocku, kritické sekce s aktivním čekáním
- Peter Cordes – analýza dotazu na StackOverflow na optimalizaci pro GCC

  https://stackoverflow.com/questions/33284236/implementing-a-ticket-lock-with-atomics-generates-extra-mov

```
#include <atomic>

struct atom_ticket { std::atomic<uint32_t> next_ticket,now_serving;};

void lock_acquire(atom_ticket* tkt) {
    const auto my_ticket =
        tkt->next_ticket.fetch_add(1,std::memory_order_acquire);
    while (tkt->now_serving.load(std::memory_order_acquire) !=
           my_ticket) {
        _mm_pause(); /* specifické pro x86, #include <immintrin.h> */
    }
}

void lock_release(atom_ticket* tkt) {
    tkt->now_serving++; // typem je dané, že bude inkrement atomický
}
```

Ticket-lock – kompilace pro x86_86

- Kompilace na https://gcc.godbolt.org/
- x86_64 gcc 5.2 -std=gnu++1y -Wall -O3 -ffast-math -fverbose-asm -march=native -mtune=native

```
lock_acquire(atom_ticket*):
        mov         edx, 1
        lock xadd   DWORD PTR [rdi], edx
        add         rdi, 4
.L2:
        mov         eax, DWORD PTR [rdi]
        cmp         edx, eax
        jne         .L2
        rep ret

lock_release(atom_ticket*):
        lock add    DWORD PTR [rdi+4], 1
        ret
```

Ticket-lock – kompilace pro MIPS

- Kompilace na https://gcc.godbolt.org/
- MIPS gcc 5.4 -std=gnu++1y -Wall -O3 -ffast-math

```
lock_acquire(atom_ticket*):
1:  ll      $3,0($4)
    addiu   $1,$3,1
    sc      $1,0($4)
    beq     $1,$0,1b
    nop
    sync
    addiu   $4,$4,4
2:  lw      $2,0($4)
    sync
    bne     $3,$2,1b
    nop
    jr      $31
    nop
```

```
lock_release(atom_ticket*):
    sync
1:  ll      $1,4($4)
    addiu   $1,$1,1
    sc      $1,4($4)
    beq     $1,$0,1b
    nop
    sync
    jr      $31
    nop
```

Ticket-lock – kompilace pro ARM Aarch64

- Kompilace na https://gcc.godbolt.org/
- ARM64 gcc 6.3 -std=gnu++1y -Wall -O4

```
lock_acquire(atom_ticket*):
.L4:ldaxr   w2, [x0]
    add     w1, w2, 1
    stxr    w3, w1, [x0]
    cbnz    w3, .L4
    add     x0, x0, 4
.L2:ldar    w1, [x0]
    cmp     w2, w1
    bne     .L2
    ret
```

```
lock_release(atom_ticket*):
    add     x0, x0, 4
.L7:ldaxr   w1, [x0]
    add     w1, w1, 1
    stlxr   w2, w1, [x0]
    cbnz    w2, .L7
    ret
```

Ticket-lock – kompilace pro ARM 32-bit

- Kompilace na https://gcc.godbolt.org/
- ARM gcc 6.3.0 -std=gnu++1y -Wall -O4

```
lock_acquire(atom_ticket*):
    push {r4, r5, r6, lr}
    mov  r1, #1
    mov  r5, r0
    bl   __sync_fetch_and_add_4
    mov  r6, r0
    add  r5, r5, #4
.L2:ldr  r4, [r5]
    bl   __sync_synchronize
    cmp  r6, r4
    bne  .L2
    pop  {r4, r5, r6, lr}
    bx   lr
```

```
lock_release(atom_ticket*):
    push {r4, lr}
    add  r0, r0, #4
    mov  r1, #1
    bl __sync_fetch_and_add_4
    pop  {r4, lr}
    bx   lr
```

Ticket-lock – kompilace pro ARM Cortex-A7

- Kompilace na https://gcc.godbolt.org/
- ARM gcc 6.3 -std=gnu++1y -Wall -O4 -march=armv7-a

```
lock_acquire(atom_ticket*):
.L4:ldrex   r2, [r0]
    add     r3, r2, #1
    strex   r1, r3, [r0]
    cmp     r1, #0
    bne     .L4
    add     r0, r0, #4
    dmb     ish
.L2:ldr     r3, [r0]
    dmb     ish
    cmp     r2, r3
    bne     .L2
    bx      lr
```

```
lock_release(atom_ticket*):
    add     r0, r0, #4
    dmb     ish
.L7:ldrex   r3, [r0]
    add     r3, r3, #1
    strex   r2, r3, [r0]
    cmp     r2, #0
    bne     .L7
    dmb     ish
    bx      lr
```

Paměťový model pro paralelní programování a Linux

- Paul E. McKenney, IBM
- Memory Ordering in Modern Microprocessors,

  http://www2.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
- Is Parallel Programming Hard, And, If So, What Can You Do About It?

  https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
- SMP Scalability Papers

  http://www2.rdrop.com/users/paulmck/scalability/
- Read-Copy-Update (RCU) papers

  http://www2.rdrop.com/users/paulmck/RCU/

Závěr

- Pro synchronizaci v paralelních počítačích se sdílenou pamětí je vhodné definovat sekvenčně konzistentní paměťový systém.
- Dnešní paralelní systémy podporují některý ze slabších modelů paměťové konzistence, kdy sekvenčně konzistentní pohled je možné zajistit pouze v definovaných místech programu a to nejlépe pomocí některé ze synchronizačních operací.
- Synchronizační operace jsou vzájemné vyloučení, dvoubodová synchronizace a synchronizační bariéra.
- Základem implementace synchronizačních operací jsou atomické RMW primitivy a paměťová bariéra.
  - V ISA procesorů se vyskytují RMW instrukce T&S, SWAP, F&I, C&S
  - Novější procesory podporují tvorbu RMW primitiv pomocí instrukcí LL a SC, které umožňují efektivní implementaci synchronizačních operací v systémech se skrytými paměťmi
- Paměťová bariéra se stará o oddělení paměťových operací před a za bariérou. Jedná se o instrukci, kterou rovněž najdeme v ISA současných procesorů.

Použité zdroje:

1. Bečvář M: Přednášky Architektury paralelních počítačů II: Sekvenční konzistence paměti, Implementace synchronizačních událostí, s použitím slajdů Prof. Ing. Pavla Tvrdíka, CSc.
2. Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005
3. https://www.cs.utexas.edu/~pingali/CS395T/2009fa/lectures/mesi.pdf
4. D.E.Culler, J.P. Singh,A.Gupta: Parallel Computer Architecture: A HW/SW Approach,Morgan Kaufmann Publishers, 1998.
5. Einar Rustad: Numascale. Coherent HyperTransport Enables the Return of the SMP
6. Intel Itanium Processor 9300 Series and 9500 Series - Datasheet http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/itanium-9300-9500-datasheet.pdf
7. Daniel Molka et al.: Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2015_ICPP_authors_version.pdf?lang=de
8. Michael R. Marty: Cache Coherence Techniques for Multicore Processors, 2008.
9. Brian Railing: Synchronization: Basics. 15-213: Introduction to Computer Systems https://www.cs.cmu.edu/afs/cs/academic/class/15213-m16/www/lectures/24-sync-basic.pdf
10. http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/14_relaxedReview.pdf
11. Adve and Gharachorloo: Shared Memory Consistency Models, WRL Research Report.
12. Synchronizing Instructions for PowerPC™ Instruction Set Architecture http://cache.freescale.com/files/32bit/doc/app_note/AN2540.pdf