# Multiprocesory se sdílenou pamětí: Koherence pamětí cache

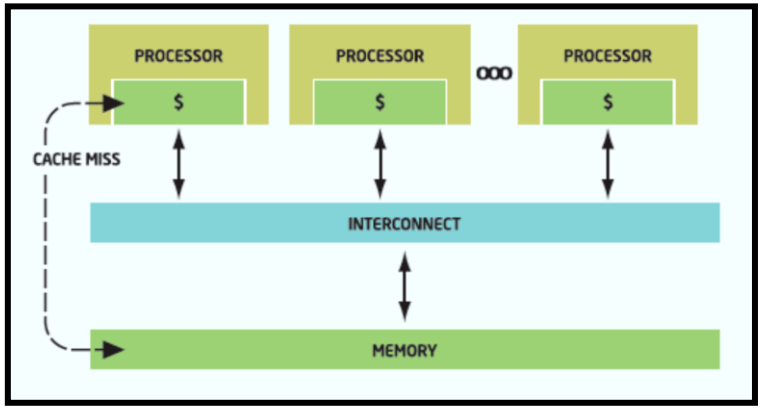
## AVS – Architektury výpočetních systémů

## Týden 10, 2023/2024

**Jirka Jaroš**

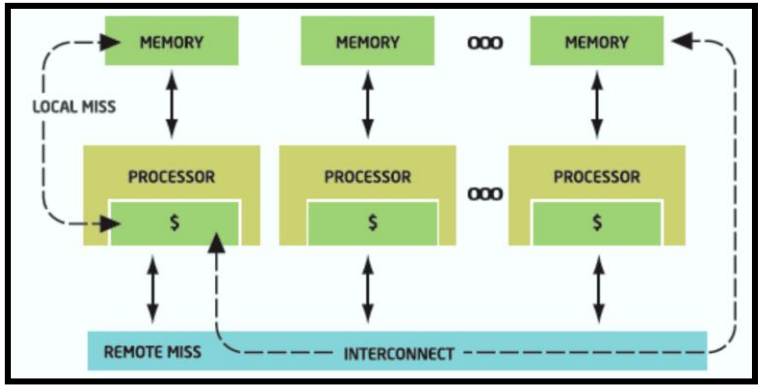
Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz

# Architektury se sdíleným adresovým prostorem




## SAS – UMA (CMP)

- uniformní doba přístupu do hlavní paměti. Implementace:
  - výkonné sběrnice: souběžné přenosy
  - X-bar: paralelní přenosy

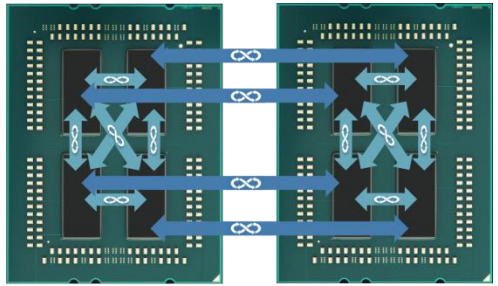



## SAS – NUMA (DSM)

- neuniformní doba přístupu do hlavní paměti (např. lokální a vzdálený výpadek v cache).
  - Vzdálený přístup: automatické zaslání zprávy tam a zpět

# Architektury se SAS: CMP (UMA) a DSM (NUMA)

## 1. Čipové multiprocesory CMP


Source: AMD

- centrální sdílená paměť (UMA)
- s plným propojením (2–8 jader nebo procesorů)
- se sběrnicí (do 16 jader)
- s křížovým přepínačem (8 a 16 jader Sparc T2, T3)
- s kruhovým propojením (lepší než sběrnice, levnější než X-bar).
  8–16 jádrový CPU Sandy Bridge, Haswell (Intel)
- s propojovací sítí – dnešní procesory Skylake X (2D mřížka), EPYC (Infinity path)

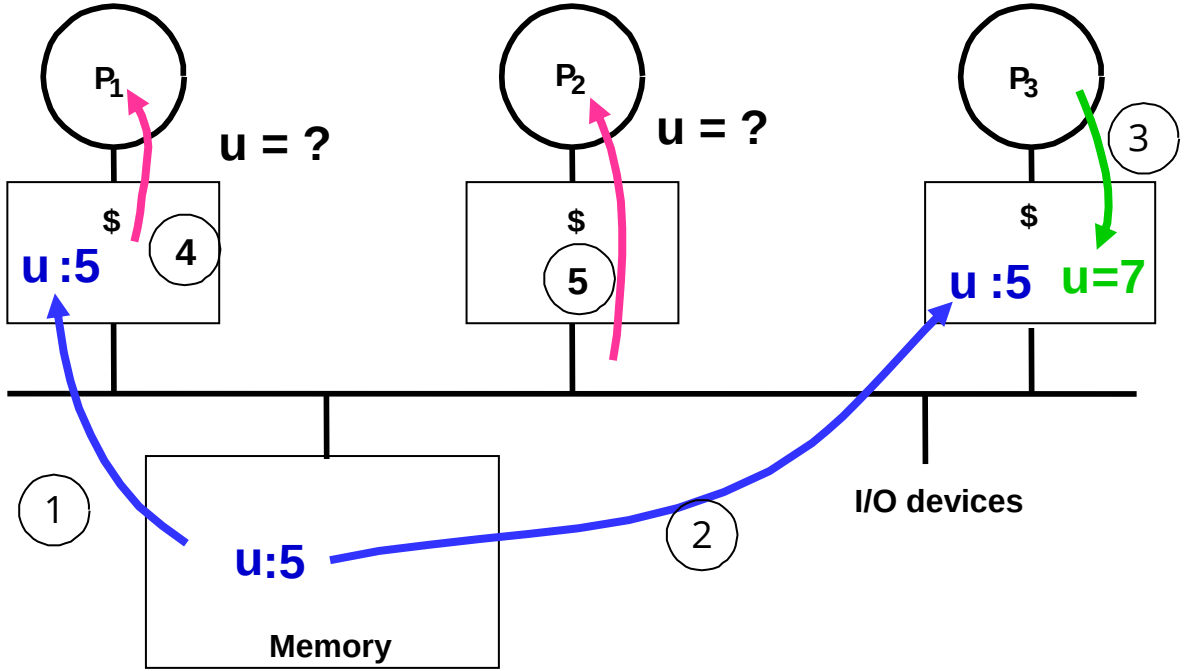
## 2. Distribuovaná sdílená paměť DSM

- sdílená paměť fyzicky distribuovaná → NUMA
- propojovací síť, např. 2D-torus, tlustý strom, hyperkostka
- někdy oddělená adresová a datová síť
- rozšiřitelnost až do 2048 jader a 64 terabajtů (TB)

# CMP a DSM s koherentními cache

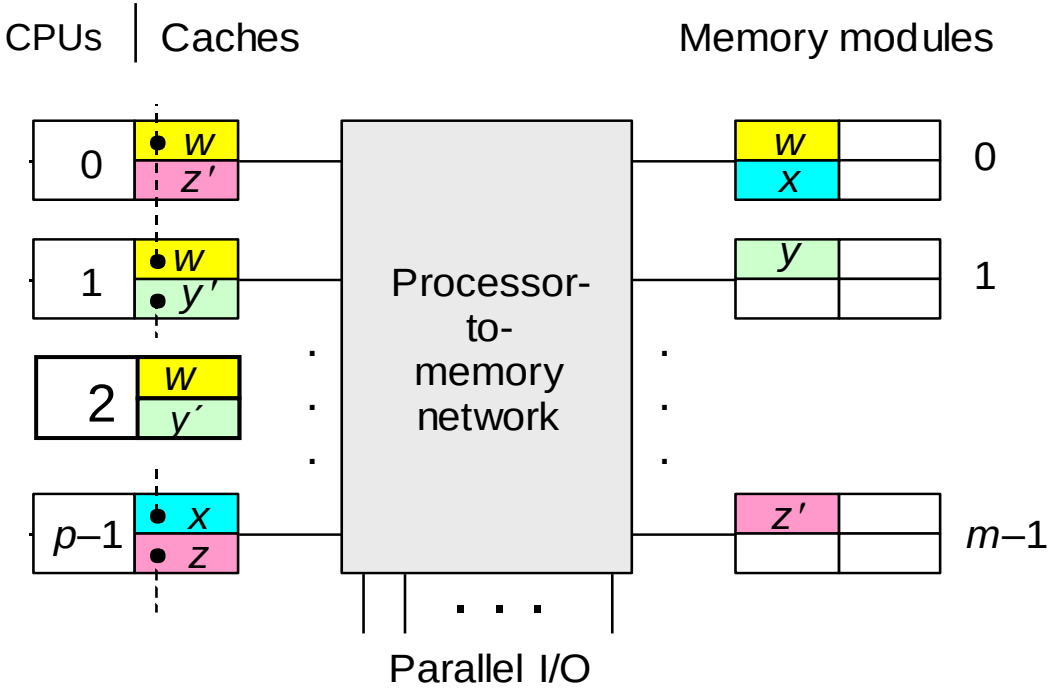
- Jsou většinou **symetrické multiprocesory** (SMP), tj.
  - jsou složeny ze shodných CPU (jader), která
  - sdílí hlavní paměť (SAS), buď UMA nebo NUMA
  - jsou řízeny rovným způsobem jednou instancí OS.
- Jakmile do **sdílené paměti** může přistupovat více vláken, je nutné zajistit, aby na jedné adrese našla při čtení vždy stejná data, ať jsou data v kterékoliv úrovni cache nebo v hlavní paměti (tj. **koherentní** kopie).
- **Proto je kritická**
  - aktualizace (zápis) dat na jednu adresu 1 vláknem (správný postup je dán protokolem koherence cache)
  - aktualizace dat na různých adresách více vlákny (pořadí je určeno modelem paměťové konzistence).

# Koherence pamětí cache (CC)




Koherence pamětí cache znamená, že pro danou adresu existuje jediná (koherentní) verze sdílených dat v jedné, několika nebo i ve všech pamětech cache. Na kopii v paměti nezáleží, ta může být zastaralá (neplatná).

# Typy bloků dat paměti cache a sdílené paměti




Možné stavy bloku v cache:

- čistý/špinavý
- jediný/sdílený
- platný/neplatný

Které kopie (s čárkou je nejnovější) se nesmí v cache vyskytovat?

x, w = čistá kopie, y´= špinavá, z = nekoherentní

# Podmínky koherence

- **Sekvenční počítač**:
  - vidí vlastní čtení a zápisy v pořadí, jak je vydává
  - při čtení vidí poslední hodnotu zapsanou na danou adresu.
  - u paralelního počítače ale různé CPU nemusí vidět paměťové akce ve stejném pořadí.
- **U paralelního počítače** musíme zajistit
  - **šíření zápisů** – každý zápis se stane viditelný všem CPU (tj. jádrům všech procesorů)
  - **serializaci zápisů** – všechny CPU vidí zápisy na adresu ***x*** v témže pořadí (u sběrnice splněno triviálně).
- **Koherence** se týká jednoznačnosti zápisů na 1 adresu
- **Konzistence** se týká pořadí přístupů na různé adresy.

# Protokoly koherence cache (Protokoly CC)

Při výpadku čtení nebo zápisu v místní cache vyšle jádro (vlákno) žádost s adresou bloku přes rozhraní do sdílené paměti (cache) vyšší úrovně.

**Protokoly CC využívají:**

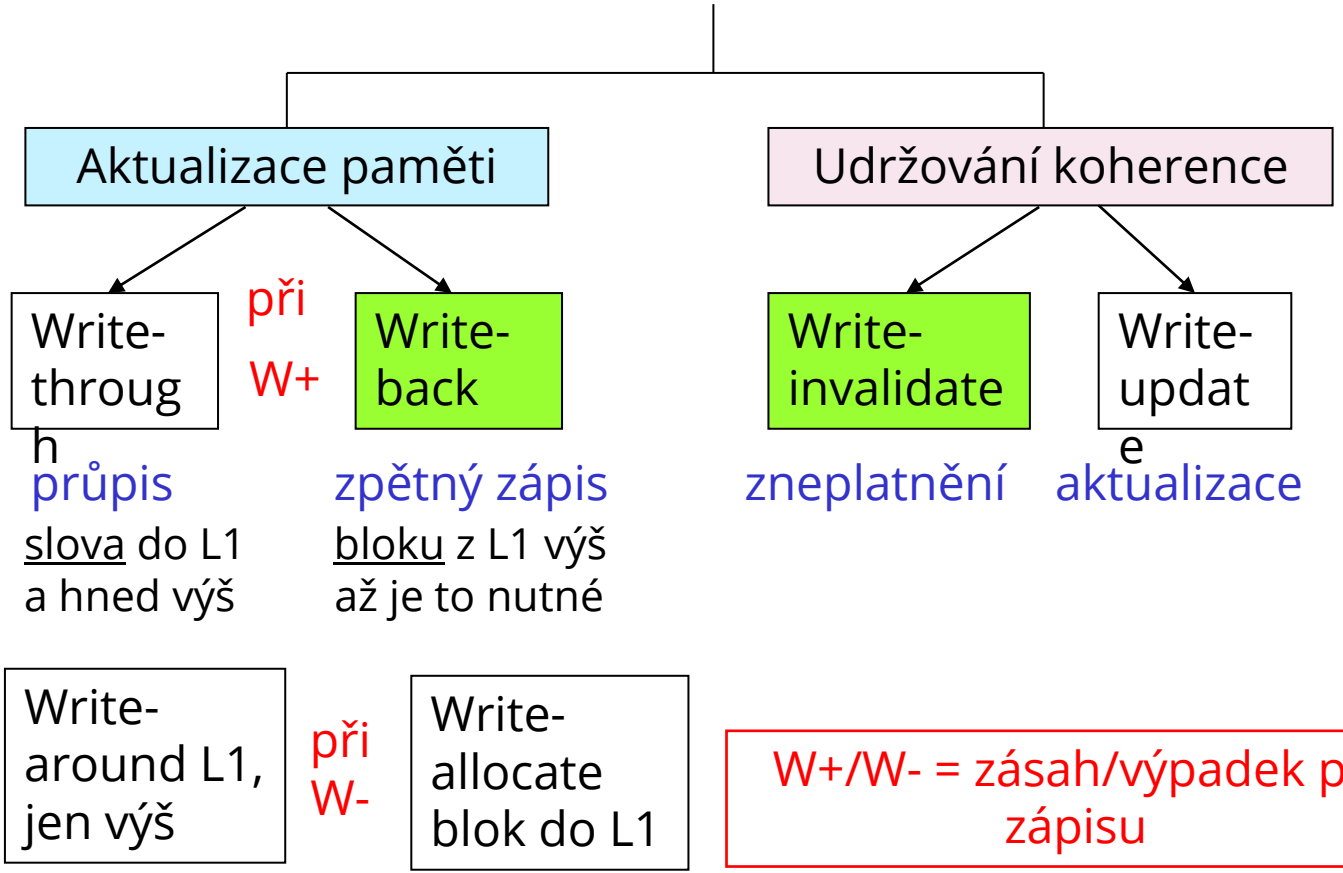
- **Monitorování komunikace** (snooping, naslouchání).
  Stav bloku je připojen ke všem jeho kopiím v místních cache. Žádost s adresou bloku jde od žadatele **všem (broadcast)**, reagují a odpovídají jen ti, kterých se to týká.
- **Distribuované adresáře**
  Stavy bloku paměti v jednotlivých cache jsou uloženy ještě odděleně v některém z adresářů. Žádost jde od žadatele přes domovský adresář bloku **jen majitelům kopií**, jen ti také odpovídají.

# Protokoly koherence

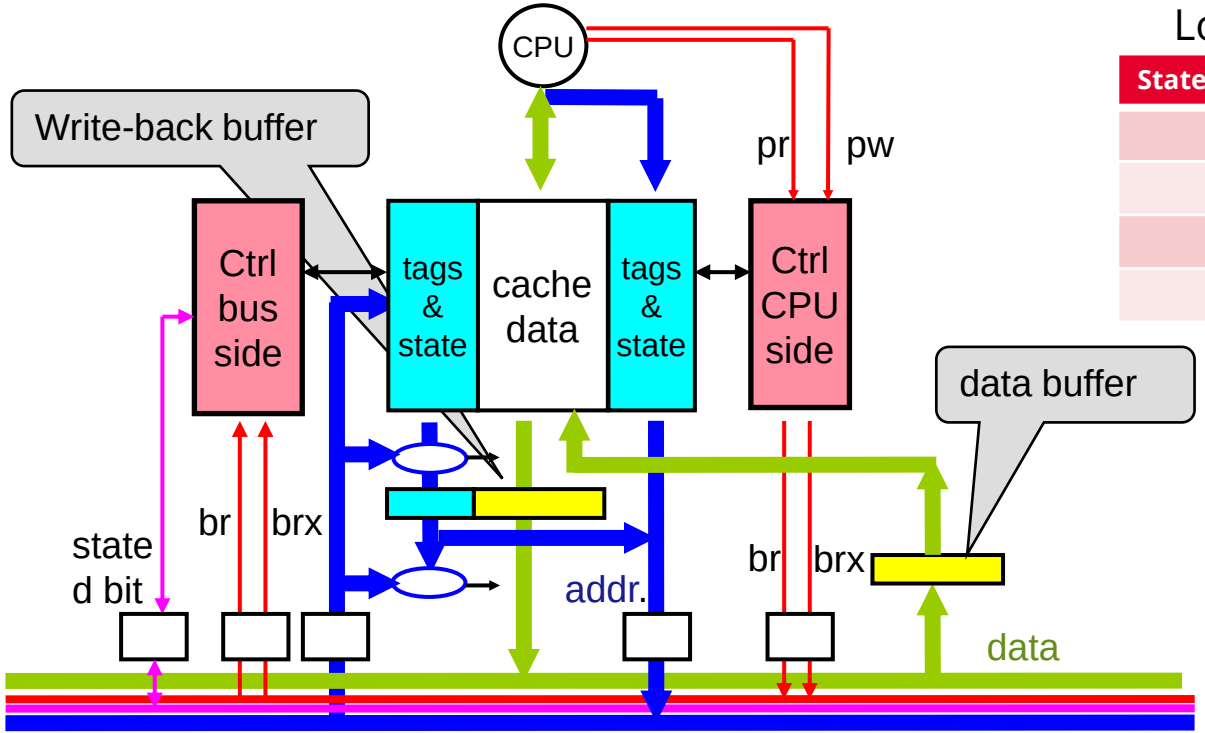
- Pro sběrnicové systémy jen monitorování komunikace.
- NUMA systémy používají většinou distribuované adresáře; pro případ naslouchání je adresová a datová síť oddělená.
- Mnoho implementačních záležitostí:
  - více úrovní cache (L1, L2, L3, …),
  - sdílená paměť (SM) na úrovni L2 nebo vyšší,
  - oddělené I-cache a D-cache,
  - velikost bloku (řádku) cache,
  - způsob aktualizace dat v hlavní paměti,
  - atd., atd.

CC protokol je *distribuovaný algoritmus implementovaný souborem kooperujících řadičů cache (CacheCtrl) – konečných automatů.*
CC Je popsán diagramem stavových přechodů bloků cache.

# Protokoly CC podporované v HW



# Řadič (Ctrl) paměti cache s nasloucháním




Local Cache

| State | Tag | Data |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

Remote Cache

| State | Tag | Data |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

comparator

buffer

- pr, pw = proc. read, write
- br, bw/brx = bus read, write/read exclusive

# Protokoly CC podporované v praxi

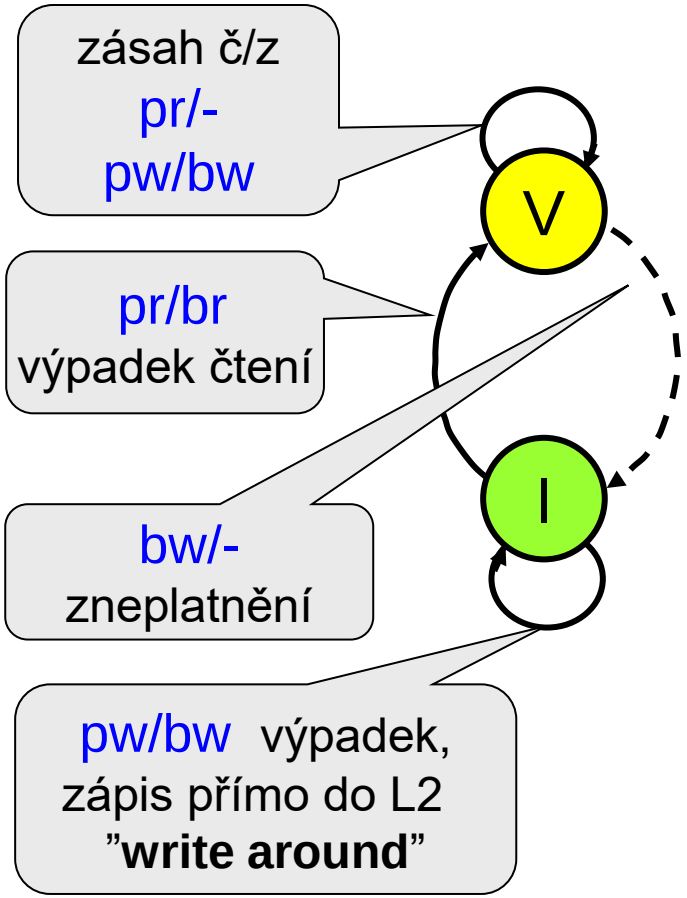
Aktualizace paměti

Udržování koherence

$$\left\{\begin{array}{l}\textbf{write-through}\\ \textbf{write-back}\end{array}\right\} \times \left\{\begin{array}{l}\textbf{write-update}\\ \textbf{write-invalidate}\end{array}\right\}$$

- **write-through, write-invalidate:** 2 stavy
- **write-back, write-invalidate:**
  - MSI (3 stavy),
  - MESI (4 stavy) – Intel Pentium D
  - MOESI (5 stavů) – AMD Opteron
  - MESIF (5 stavů) – Intel Nehalem a vyšší
- **write-back, write update**: Dragon (4 stavy)

# Dvoustavový CC protokol (Valid – Invalid) na sběrnici




| Aktualizace | Koherence |
|---|---|
| „write-through, | write-invalidate“ |
| zapisuje slovo, | zneplatní blok |

Legenda:
(vstup/výstup) řadiče cache
- pr, pw = proc. read, write
- br, bw = bus read, write
- blok v cache označen jedním bitem “in/valid”

**použití**: CC mezi privátními cache L1 a sdílenou L2 ve vícejádrových procesorech.
Nevýhoda: velké množství výpadků

# Třístavový protokol MSI

Aby se nemusely opakované zápisy do bloku cache neustále propisovat výš do SM, je možné špinavou kopii v L1 cache označit dalším stavem M a nechat ji v L1 co nejdéle (dokud nebude blok vybrán k výměně za jiný blok).

| **Modifikovaný M:** | **Sdílený S:** | **Neplatný I:** |
|---|---|---|
| jen 1 cache má platnou špinavou kopii, kopie ve sdílené paměti SM je zastaralá<br>dirty bit = 1 | 1 nebo více pamětí cache má kopii shodnou se sdílenou pamětí SM<br>(čistá kopie) | v této cache je pouze neplatná kopie (zastaralá) |

**Blok S lze jen číst**. Před zápisem musí CPU změnit jeho stav na M a ostatní kopie S musí být zneplatněny;
CPU je pak jediný **vlastník bloku M**, má k němu exkluzivní přístup a **může** do něj **zapisovat**.

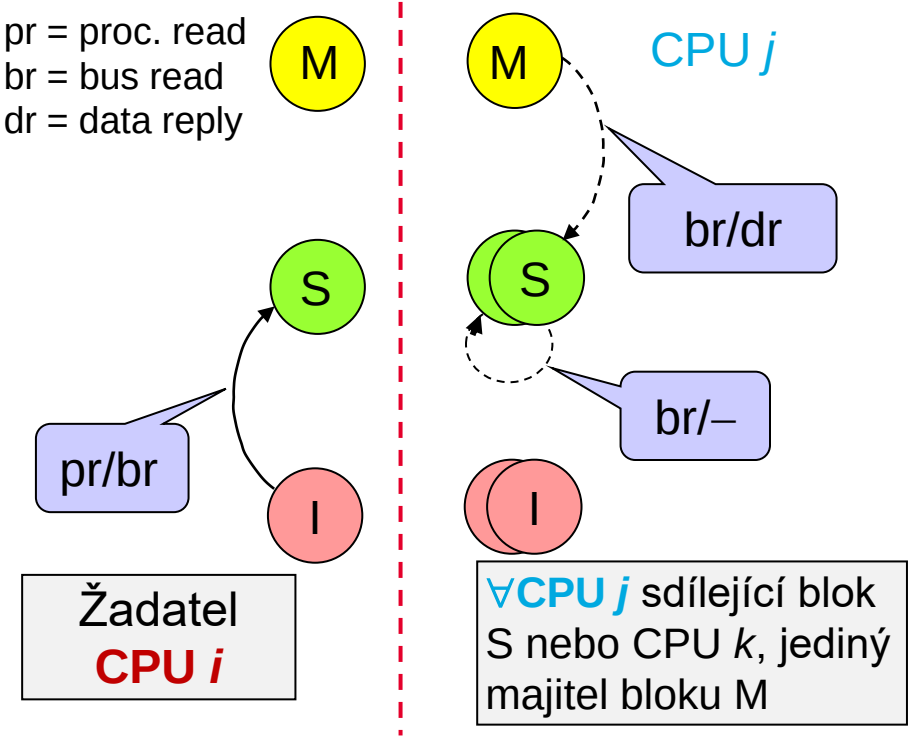
# Protokol MSI – naslouchání sběrnice

- Stav bloku dat se může lišit v různých cache. Změny stavů bloků cache (2 bity valid a dirty) provádějí řadiče cache (CacheCtrl), které reagují na lokální nebo vzdálené výpadky/zásahy v cache.
- CC protokol je *distribuovaný algoritmus implementovaný souborem kooperujících řadičů cache (CacheCtrl) – konečných automatů.*
  Je popsán diagramem stavových přechodů bloků cache.
  Značení: (vstup/výstup) řadiče cache.
- **Dále předpokládáme**:
  1. pouze 1 úroveň cache L1 a sdílená paměť RAM;
  2. CC protokol: write-back, write-invalidate;
  3. atomickou sběrnici, tj. pouze 1 rozpracovanou transakci.

# Fáze paměťové transakce na sběrnici

1. **Žádost o sběrnici, arbitráž, přidělení sběrnice.** Arbitr sběrnice vybere jednu ze žádostí a potvrdí ji.
2. **Žádost s adresou** bloku dá vybraný žadatel na sběrnici. Ostatní CPU naslouchají sběrnici, řadiče cache vyhodnocují situaci a reagují na ni změnou nebo hlášením stavu bloku, případně odesláním dat (majitel bloku **M**).
3. **CPU, která má dostat data**, má žádaný blok buď ve stavu **I** nebo blok není v cache přítomen vůbec. Pak musí vybrat blok pro výměnu (**victim** = oběť) a pokud je to blok **M**, musí provést jeho zpětný zápis (**write back, wb**) do sdílené paměti. Dokud CPU není připravena, nastaví wired-OR wait signál.
4. **Odesílatel hlásí signálem** dr (data reply / response) přítomnost dat na sběrnici. Příjemce dat jejich dodání nepotvrzuje, zapíše blok do cache s aktualizovaným stavem.

# Výpadek při čtení (R-) sdílené kopie S nebo modifikované kopie M




pr → CacheCtrl *i* → **R-** :

1. žádost o sběrnici;
2. když přidělena: br;
3. když *victim* dirty: wb;
4. když data přijata: I → S;

CacheCtrl *j*:

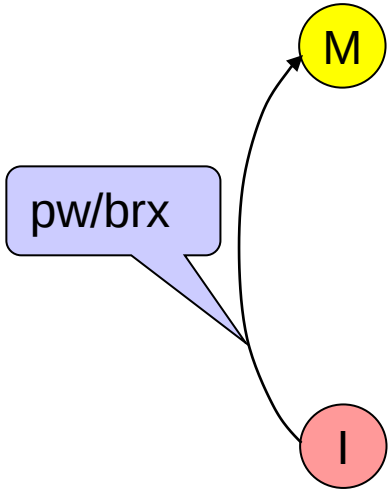
1. br: když stav = M: dr M;
2. M → S;
3. dirty bit → sběrnice

MemCtrl:

br: když dirty bit = 0: dr S;

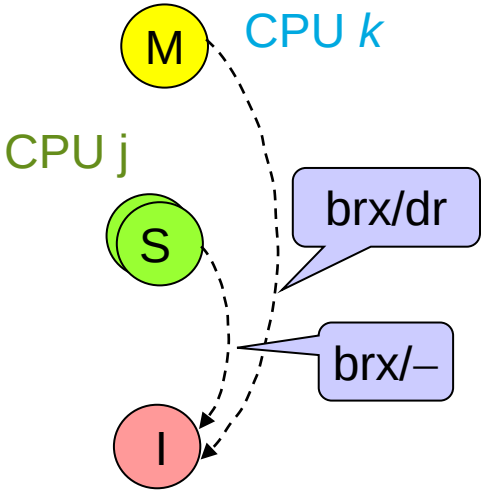
**Závěr:** data dá na sběrnici paměť (S) nebo majitel (M). Pokud byl blok ve stavu (M), putují data i do paměti.

# Výpadek při zápisu (W-) do sdílené S nebo modifikované kopie M




**CPU *i*,** žadatel

pw = procesor write




∀ **CPU *j*** sdílející blok S nebo **CPU *k***, jediný majitel bloku M

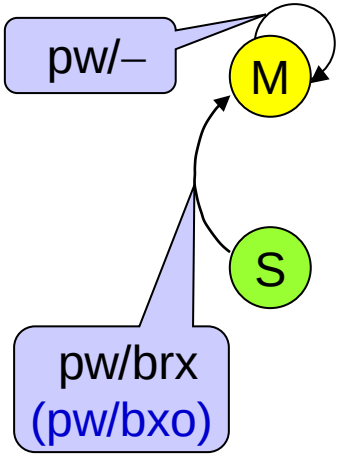
pw → CacheCtrl *i* → **W-** :
1. žádost o sběrnici;
2. když přidělena: brx (bus read exclusive) na sběrnici;
3. když victim dirty: wb;
4. když data přijata: I → M;

CacheCtrl *j*, *k*:
1. brx: když stav = M: dr M;
2. změna stavu S, nebo M → I;

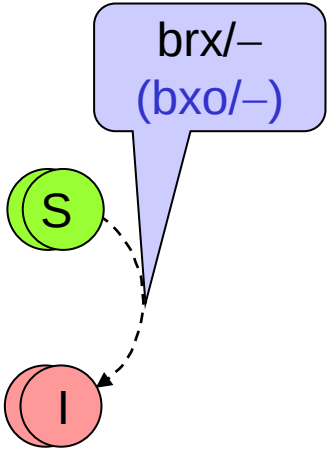
MemCtrl:
brx: když dirty bit = 0:
dr S, S → I; kopie neplatná

**Závěr:** před zápisem se musí data zneplatnit u předchozího vlastníka (M) nebo všech vlastníků (S). Data dá na sběrnici paměť (S) nebo vlastník (M).

# Zásah při zápisu (W+) do sdílené S nebo modifikované kopie M




Žadatel **CPU *i***
chce zapsat
do bloku S/M




∀ **CPU *j***
sdílející blok S

pw → CacheCtrl *i* → W+:
1. když stav = M: **exit**;
2. žádost o sběrnici;
3. když přidělena: brx („bus read exclusive“);
4. stav S → M;

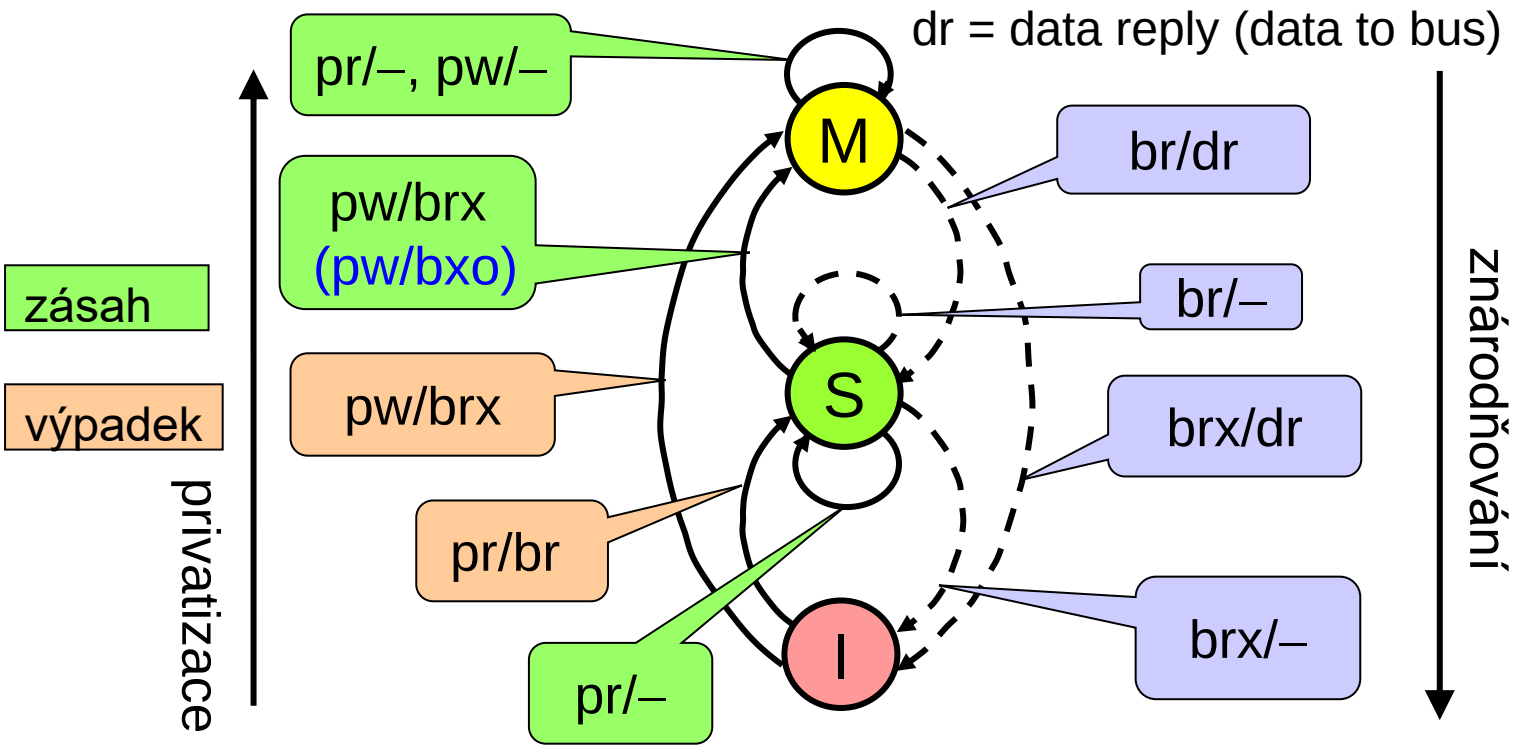
CacheCtrl *j*:
brx: stav S → I

MemCtrl:
brx: když dirty bit = 0: dr S;
(zbytečné, žadatel má kopii S.)
*Alternativa:*
bxo: „bus exclusive owner“ žádný přenos!

**Závěr:** před zápisem se musí data zneplatnit u všech vlastníků (S). Příkaz bxo místo brx vyžádá jen invalidaci bloku (S) u ostatních cache a blokuje načítání z paměti.

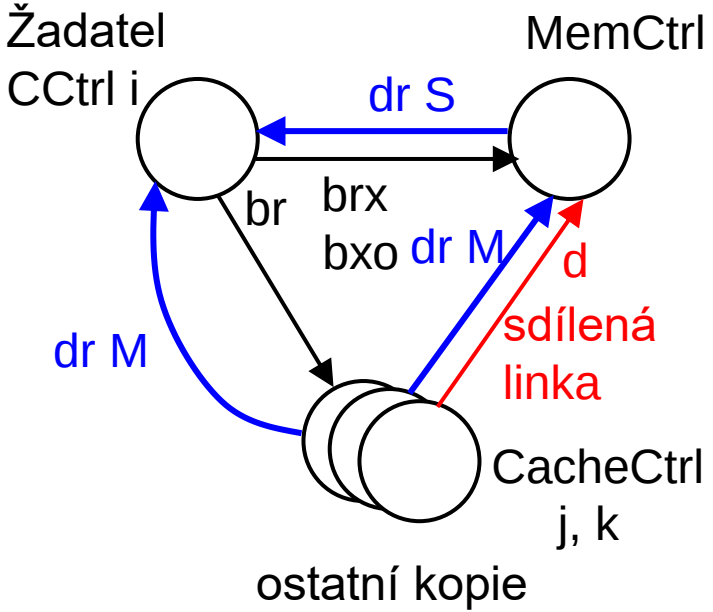
# Celkový stavový diagram protokolu MSI

Tlusté přechody: žadatel, čárkované přechody: ostatní cache




bxo = exclusive owner, variantní transakce bez přenosu dat

# Komunikace na sběrnici u CC s nasloucháním




- Na sběrnici existují sdílené linky, na kterých může CPU nastavit signál nebo podmínku viditelný/nou všem CPU a též řadiči MemCtrl.
- Sdílený wired-or signál dirty bit d na sběrnici je interpretován paměťovým řadičem takto:

br/brx a d = 1: probíhá přenos bloku dr M z cache do cache a při br i do paměti, kde bude označen S

br/brx a d = 0: přenos bloku dr S z paměti do cache žadatele

bxo a d = 0: neproběhne žádný přenos dat.

# Příklad 1. CC protokol MSI

3 procesorový systém se společnou sběrnicí, sdílenou pamětí a koherentními pamětmi cache L1 (protokol MSI) vykonává operace se sdílenou proměnnou u dle tabulky. Vyplňte tabulku pomocí symbolů M, S, I, br, brx, SM, C1, C2, C3, - :

| | Akce procesoru | Signály na sběrnici | Stav bloku v C1 | Stav bloku v C2 | Stav bloku v C3 | Data dodá | Data přijme |
|---|---|---|---|---|---|---|---|
| 0. | - | - | M | I | I | - | - |
| 1. | P3 čte u | | | | | | |
| 2. | P3 píše do u | | | | | | |
| 3. | P1 píše do u | | | | | | |
| 4. | P2 čte u | | | | | | |
| 5. | P3 píše do u | | | | | | |
| 6a. | P3 čte u | | | | | | |
| 6b | P3 čte u | | | | | | |

# POKROČILÉ PROTOKOLY CC S NASLOUCHÁNÍM

# 4 stavový protokol MESI (Illinois)

- Další stav **E** bloku cache značí, že existuje čistá kopie pouze v jediné cache. Stačí 2 bitový kód jako pro MSI.
- Výhody:
  - Ve stavu **E** (i **M**) je možný zápis (E → M, M → M) **bez oznamování** brx nebo bxo na **sběrnici**
  - Blok **E** dodá cache dalšímu žadateli (E → S) rychleji místo Mem!

- Na sběrnici jsou ale teď místo signálu **dirty bit d** nově dva wired-OR signály:
  - owner bit e (exkluzivní vlastník, signalizuje **M** i **E**) a
  - shared bit f (kopie jen v paměti a v žádné cache: f = false;
    existuje platná kopie alespoň v jedné cache: f = true).

# 5 stavový protokol MESIF (Intel)

- Kolektivní cíl všech cache je minimalizovat přístupy do sdílené hlavní paměti mimo čip, využít data na čipu.
- Nový stav F, read-only forwarding, umožňuje přenos čisté kopie z jedné cache do druhé. **Je to rychlejší (např. u vícejádrových procesorů) než přenos z paměti.**
- Ve sdílených kopiích je vždy jen jedna ve stavu F a ta se kopíruje. Ostatní kopie jsou ve stavu S.
- Když je blok ve stavu F kopírován, stav F migruje do nové kopie zatímco zdrojová kopie přejde do stavu S.
- Blok M se musí před jeho sdílením nebo výměnou (kvůli místu) nahrát zpět do paměti.

# 5 stavový protokol MOESI (AMD)

- MOESI má jinou optimalizaci:
  1. **eliminuje kopírování bloku M do paměti před jeho sdílením**; původní blok M přejde do stavu O (owned),
  2. **další sdílené špinavé kopie S** vznikají kopírováním dat z bloku O. Kopie v paměti je zastaralá.
- Stavy E a O a M jsou unikátní (blok jen v jedné cache)
- Sdílené kopie jsou buď špinavé M → [O, S, S, S,... ] nebo čisté E → [ S, S, S, ...]
- Stavy M, O, E dovolují zápis (O a E přejdou do M) a nahrazují paměť jako zdroj dat (přenos cache-to-cache).

# Shrnutí vlastností bloků cache

| | čistý/ špinavý | unikátní | ostatní cache | povolení | zdroj dat |
|---|---|---|---|---|---|
| **S** | čistý | ne | S nebo I | R | ne |
| **M** | špinavý | ano | I | R/W | ano |
| **E** | čistý | ano | I | R/W | ano |
| **F** | čistý | ano | S nebo I | R | ano |
| **O** | špinavý | ano | S nebo I | R/W | ano |
| **I** | – | – | – | žádné | ne |

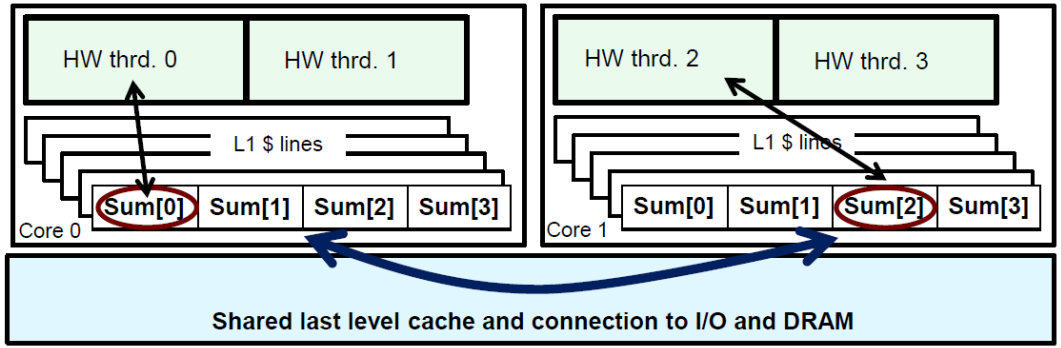
# FALEŠNÉ SDÍLENÍ

# Falešné sdílení

- **Různá data (prvky vektoru) modifikovaná opakovaně dvěma vlákny by neměla být v jednom bloku cache!**

```
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < size(); i++)
  a[i]++;
```

- Při zápisu jedním jádrem bude totiž pokaždé kopie bloku druhému jádru zneplatněna.
- Náprava: **zvýšením dimenze pole** (1 prvek na celý blok, *padding*) nebo **privatizace**.



# Příklad: m = 4 vlákna, řádkové součty matice a(m, n)

```
float a[m][n], s[m];   // s = vektor řádkových součtů
void rowsum1(float* a[], float* s, int m, int n)
{
  int i, j;
  #pragma omp parallel for private(i, j) shared(s, a)
  for (i = 0; i < m; i++)
  {
    s[i] = 0.0f;
    for (j = 0; j < n; j++)
       s[i]+= a[i][j];
  }
}
```

**P = 4: 15.06 s**
**P = 2: 6.08 s**
**P = 1: 3.77 s**

Dvě vlákna mohou aktualizovat prvky s[i] nacházející se v 1 bloku cache.

V bloku 64 B bude 16 prvků s[i] s offsetem 0, 4, 8, …, 60.

# Odstranění falešného sdílení prázdnou výplní (padding)

```
float a[m][n], s[m][C]; // C * sizeof(float) = velikost bloku cache [byte]
void rowsum2(float* a[], float* s[], int m, int n)
{
  #pragma omp parallel for shared(s, a)
  for (int i = 0; i < m; i++)
  {
     s[i][0] = 0.0f;
     for (int j = 0; j < n; j++)
        s[i][0] = s[i][0] + a[i][j];
   }
}
```

**P = 4: 1.03 s**
**P = 2: 2.04 s**
**P = 1: 3.76 s**

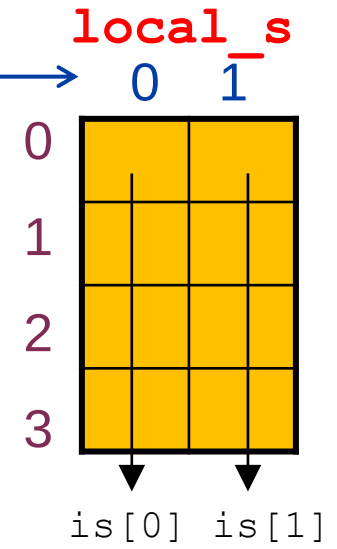
Např. pro blok cache 64 B je C = 16 → float s[i][0] bude vždy na začátku bloku cache, zbytek bloku (15 float) nebude využit.

# Příklad: Počet sudých a lichých prvků integer pole is

```
constexpr int MAX_NUM_THREADS = 4;
int           local_s[MAX_NUM_THREADS][2], is[2];

#pragma omp parallel shared(local_s, is)
{
  int my_id = omp_get_thread_num();
  #pragma omp for private(i, index) schedule(static)
  for (i = 0; i < N; i++) {
    index = ia[i] % 2; // zbytek
    local_s[my_id][index]++;
  }

  #pragma omp atomic
    is[0] += local_s[my_id][0]; /*sude*/
  #pragma omp atomic
    is[1] += local_s[my_id][1]; /*liche*/
}
```



# Odstranění falešného sdílení privatizací local_s

```
constexpr int MAX_NUM_THREADS = 4;
int           local_s[2], is[2];
#pragma omp parallel private (local_s) shared (is)
{
  local_s[0] = 0;
  local_s[1] = 0;

  #pragma omp for private(index) schedule(static)
  for (i = 0; i < N; i++) {
    index = ia[i] % 2;
    local_s[index]++;
  }

  #pragma omp atomic
    is[0] += local_s[0];
  #pragma omp atomic
    is[1] += local_s[1];
}
```

modifikace lokálních kopií dat v privátním zásobníku

redukce lokálních dat na sdílené pole is se 2 prvky

# Pokračování příště