Distribuovaná sdílená paměť

Základní předpoklad distribuovaného stránkování je, že žádný procesor nemůže přímo přistupovat do paměti jiného (cizího) procesoru. Takovýmto systémům se říká **NORMA - No Remote Memory Access** (bez vzdáleného přístupu do paměti).

Oproti tomu systém **NUMA - No Uniform Memory Access** spočívá v tom, že procesor přímo adresuje položky v paměťovém prostoru, přičemž jednotlivé stránky mohou být libovolně distribuovány mezi paměťmi různých procesorů, aniž je tím nějak ovlivněn výsledek programu. V okamžiku přístupu na vzdálenou stránku má systém možnost volby. Buď stránku načte, nebo ji používá vzdáleně, což má vliv na výkon systému, ale ne na správnost. Vše je řešeno na hardwarové úrovni. Takto pracují multiprocesory, u nichž je přístup k paměti zajištěn bez pomoci softwaru.

Úkolem systému distribuované sdílené paměti (**DSM**) je doplnit jisté softwarové vybavení tak, aby na multipočítačích mohly běžet multiprocesorové aplikace. Protože není možnost přístupu do vzdálené paměti (**NORMA**), je jen jediná možnost při přístupu na stránku, která se nachází u jiného procesoru - načíst ji.

Prvotní snahy tedy vedly k modifikaci programového vybavení, které bylo psáno pro multiprocesory, na programové vybavení pro multipočítače, a to pokud možno s minimálními změnami tohoto softwaru. Protože jsou multiprocesorové aplikace zvyklé na paměť **sekvenčně konzistentní**, byla tedy zpočátku snaha, aby DSM také splňovala podmínku **sekvenční konzistence**. Další zjištění vedlo k tomu, že není potřeba tak silný konzistenční model, a že jeho oslabením se programy zrychlí.

Distribuovaná sdílená paměť (DSM) je abstrakce používaná pro sdílení dat mezi procesy v počítačích, kde není sdílená fyzická paměť. Uživatelé nemusí explicitně řešit problém přenosu dat mezi uzly. Základním problémem je dobrá propustnost komunikačního subsystému, která by neměla záviset na rozlehlosti systému a počtu procesorů.

DSM je prostředek pro řešení paralelních aplikací, distribuovaných aplikací nebo skupinových aplikací, ve kterých jsou individuální sdílené datové položky přístupné přímo.

Jedná se o model typu peer-to-peer, nikoliv model klient/server.

Data jsou vyměňována komunikačním systémem V každém uzlu je z důvodu rychlého přístupu lokální kopie dat, systém musí zajistit konzistentnost dat v jednotlivých uzlech.

Řešení problému souvisí s realizací **replik** i **cacheováním** sdílených souborů.

Z technického hlediska můžeme problém řešit v prostředí **těsně vázaných multiprocesorů** se společnou (sdílenou) pamětí, nebo **volně vázaných multiprocesorů** komunikujících posíláním zpráv.

Těsně vázané multiprocesory se sdílenou pamětí – propojení pomocí sběrnice dovoluje propojit max. 10 až 20 procesorů. Realizace – architektura NUMA (Non-Uniform Memory Access) s max. 64 procesory – hierarchická architektura, kde jsou procesorové desky se 4 procesory propojeny rychlou sběrnicí nebo přepínačem. Procesory vidí jeden adresní prostor obsahující všechny paměti na všech deskách. Doba přístupu k lokální paměti je však menší než doba přístupu ke vzdálené paměti – nejedná se o transparentní pohled na paměť.

Volně vázané multiprocesory s distribuovanou (sdílenou) pamětí nemají společný paměťový prostor, ale jsou propojeny velmi rychlou sítí s přepínači. Počet procesorů může být vyšší než 64. Otázkou je, zda-li mohou být algoritmy pro sdílenou paměť přímo přeneseny do distribuované paměťové architektury.

1 Systém posílání zpráv versus distribuovaná sdílená paměť

Distribuovaná sdílená paměť je porovnatelná se systémem posílání zpráv používá asynchronní komunikaci. Jiný možný model je komunikace synchronní, založená na komunikaci typu dotaz – odpověď.

1.1 Programový model

V systému výměny zpráv má každá proměnná své místo v lokální paměti, a pokud je třeba změnit její hodnotu, pak se musí vyslat zpráva. V případě sdílené paměti je proměnná sdílena – nemusí se přenášet. Sdílená paměť je použitelná pouze v homogenních systémech, kde je shodná reprezentace dat. Systém výměny zpráv dovoluje pracovat v heterogenním prostředí, protože není problém provádět konverze zobrazení dat. Dále v systému se sdílenou pamětí nejsou jednotlivé procesy vzájemně chráněny, porucha jednoho může způsobit zhroucení jiného procesu, např. příčinou může být příliš častá změna dat.

Synchronizace mezi procesy je zajištěna v modelu posílání zpráv pomocí komunikačních primitiv ve spolupráci s dalšími prvky – např. lock serverem. Synchronizace ve sdílené paměti je zajištěna zámky a semafory.

1.2 Efektivnost

Dosud nelze říci, je-li model sdílené paměti lepší než model posíláni zpráv nebo naopak. Hodně záleží na počtu procesorů a na dané aplikaci.

2 Přístupy k implementaci distribuované sdílené paměti

Distribuovaná sdílená paměť může být realizována specializovaným hardware, konvenční stránkovou sdílenou pamětí, middleware, nebo jejich kombinací.

Speciální hardware

Architektura NUMA PLUS (No Uniform Memory Access) – vychází ze specializovaného hardware, zajišťujícího konzistentní pohled procesorů na paměť. Zpracovávají operace LOAD a STORE. Pro komunikaci se vzdálenou pamětí využívají rychlého propojení, které je analogické síťovému propojení.

Stránková virtuální paměť

V mnoha systémech je sdílená paměť realizována jako oblast virtuální paměti obsazující tentýž rozsah adresního prostoru ve všech procesech. Toto uspořádání je použitelné pouze v systému s homogenními počítači se stejným formátem dat.

Middleware

Některé programové prostředky, jako **JavaSpaces** nebo **TSpaces** podporují distribuovanou sdílenou paměť bez podpory stránkování nebo speciálního hardware, cestou nezávislou na platformě. Sdílení je realizováno komunikací mezi instancemi na úrovni podpory uživatele v klientech i serverech. Procesy při přístupu k datům ve sdílené paměti volají služby této vrstvy. Instance této mezilehlé vrstvy umístěné v různých počítačích přistupují k lokálním datovým položkám a vzájemnou komunikací zajišťují jejich konzistentnost. Efektivnost komunikace může být ovlivněna vhodnou podporou hardwarem.

Přístup založený na stránkách má tu výhodu, že nepotřebuje žádnou zvláštní strukturu nad sdílenou pamětí, jejíž data chápe jako posloupnost slabik. V principu to dovoluje programům navrženým pro multiprocesory se sdílenou pamětí pracovat na počítačích bez sdílené paměti s minimální nebo žádnou úpravou. Takovou podporu nabízí např. Mach, zajišťující přirozenou podporu sdílené paměti.

Distribuovaná sdílená paměť realizovaná na bázi stránkování je nejčastěji implementovaná a její výhoda je v její přizpůsobitelnosti. Implementace je zajištěna vytvořením podpory jádra pro uživatelské zpracování výpadků stránek. Dovoluje sdílení proměnných umístěných ve sdílené paměti.

Řešení problému pomocí middleware dovoluje vytvoření abstrakcí vysoké úrovně tj. nejen sdílení paměťových míst, ale i sdílení objektů.

3 Návrh a prostředky realizace

3.1 Struktura

Jednotlivé systémy sdílené paměti se liší mimo jiné tím, jak chápou objekty se kterými pracují, a jak jsou tyto objekty adresovány.

Slabikově (bytově) orientované

Jsou přístupné jako uspořádané pole slabik ve virtuální paměti. Podporované operace jsou typu *read* (*LOAD*) nebo *write* (*STORE*).

Objektově orientované

Sdílená paměť je strukturovaná jako soubor objektů na úrovni jazyka se sémantikou vyšší úrovně. Obsah sdílené paměti je měněn pouze při přístupu k objektům a nikdy přímým přístupem k jednotlivým proměnným. Výhodou tohoto přístupu je, že sémantika objektů může být využita při vynucené konzistentnosti. Např. systém ***Orca*** vidí sdílenou paměť jako soubor sdílených objektů a automaticky serializuje operace nad danými objekty.

Neměnitelná data

Sdílená paměť je viděna jako soubor neměnitelných datových položek, které mohou procesy číst, přidávat a rušit. Příkladem je ***Linda***, ***JavaSpaces*** a ***TSpaces***. Systémy typu Linda zavádí prostor *n-tic*, který obsahuje jeden

nebo více typů datových polí. V tomtéž prostoru dvojic mohou existovat různé typy *n-tic*. Procesy sdílí data přistupováním k témuž prostoru *n-tic*. Umístění *n-tice* do prostoru *n-tic* se děje operací *write,* čtení nebo vybrání se děje pomocí operace *read* nebo *take*. Operace *write* přidá *n-tici* bez ovlivnění stávajících *n-tic*. Operace *read* vrátí hodnotu jedné *n-tice* bez modifikace ostatních. Operace *take* vrátí *n-tici*, ale navíc ji z prostoru odstraní. K přístupu do prostoru *n-tic* se používá asociativní adresování. V parametrech se neuvádí adresa *n-tice*, ale její specifikace. Systém vrátí ty *n-tice*, které odpovídají specifikaci. Aby mohl proces synchronizovat své aktivity, jsou operace *read* a *take* blokované dokud v prostoru *n-tic* existuje nějaká *n-tice*, odpovídající specifikaci. Specifikace *n-tice* obsahuje počet polí a požadované hodnoty nebo typy polí.

3.2 Model synchronizace

Mnoho aplikací zavádí **omezení** týkající se hodnot ukládaných do sdílené paměti. To platí jak o aplikacích založených na DSM, tak i o aplikacích psaných pro multiprocesory se sdílenou pamětí. Např. jestliže *a* a *b* jsou dvě proměnné uložené v DSM, pak omezením může být to, že vždy platí *a=b*. Při manipulaci s těmito proměnnými může dojít ke vzniku nekonzistentnosti, omezení může být porušeno. Např. v případě, že dva procesy inkrementují obě proměnné souběžně. Řešením je synchronizovat tyto procesy tak, aby manipulaci s proměnnými mohl v daném okamžiku provádět pouze jeden z nich. Při použití DSM musí být realizovány distribuované synchronizační služby, zahrnující konstrukce známé jako **zámk**y a **semafory**. Synchronizace je realizována posíláním zpráv. Speciální instrukce používané v těsně vázaných multiprocesorech se společnou sdílenou pamětí, jako je *TestAndSet* mohou být použity i v DSM, avšak jejich použití je velmi neefektivní. Realizace DSM proto musí poskytnout jiné efektivní prostředky pro synchronizaci úlohám na aplikační úrovni.

3.3 Modely konzistentnosti

V systémech se společnou pamětí většinou předpokládáme striktní synchronizaci, podporovanou operacemi nebo funkcemi, které využívají znalosti globálního stavu systému. Tak lze využít např. operací nad semafory, kritické sekce nebo *sequencery*. V distribuovaném prostředí není globální stav znám. Proto musí být pro koordinaci výpočtu použity jiné metody, nepředpokládající znalost globálního stavu a mající přípustnou míru režie. Tyto metody mají společnou vlastnost, označovanou jako **consistency**, kterou je možno chápat jako soudržnost, čili soudržnost paměti a programů při manipulaci s daty.

Model konzistentnosti je smlouva uzavřená mezi softwarem a pamětí. Když se software zaváže pracovat s pamětí podle daných pravidel, paměť bude fungovat korektně. Pokud pravidla poruší, paměť nic negarantuje.

Při popisu modelů konzistentnosti budeme používat následující symboliku. Operace *R(x)v* představuje operaci čtení z místa *x* a vracená hodnota je *v*. Operace *W(x)v* představuje operaci zápisu hodnoty *v* do pozice *x*.

3.3.1 Striktní konzistence (strict consistency)

Jakékoliv čtení z paměti z adresy *x* vrátí hodnotu uloženou při posledním zápisu na adresu *x*. Striktní konzistence je nejsilnější, je přirozeně zajištěna na jednoprocesorových systémech.

Př.:

```
a=1; a=2; print(a);  // vytiskne vždy 2
```

Všechny zápisy jsou **okamžitě** všude viditelné, musí existovat přesný globální čas ve všech uzlech. Ideální pro programování, v distribuovaném systému však nedosažitelné.

```
P1:   W(x)1              P1:    W(x)1
P2:             R(x)1    P2:              R(x)0   R(x)1
```

Striktní konzistence Paměť, která není striktně konzistentní

Předchozí obrázek vpravo ilustruje případ, kdy aktualizovaná hodnota *x* není okamžitě viditelná v procesu P2.

3.3.2 Sekvenční konzistence (sequential consistency)

Výsledek jakéhokoliv výpočtu je stejný jako kdyby všechny operace všech procesorů byly vykonávány v nějakém sekvenčním uspořádání a operace každého jednotlivého procesoru jsou vykonávány v pořadí specifikovaném programem.

Sekvenční konzistence je o něco slabší model, avšak je snadno implementovatelná a příjemná pro programování. 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).

```
P1:      W(x)1                    P1:    W(x)1
P2:             R(x)1   R(x)1     P2:            R(x)0 R(x)1
```

Dvakrát spuštěný tentýž program

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

šest instrukcí - 6!=720 možných sekvencí, pouze 3*(5!/4)=90 nenarušuje kauzalitu

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

Výstup:
001011           101011           010111           111111

Signatura:
001011           101011           110101           111111
```

Jestliže spojíme výstupy procesů P1, P2 a P3 (v tomto pořadí), dostaneme **signaturu** - řetězec charakterizující konkrétní proložení instrukcí jednotlivých procesů, a tím i pořadí paměťových referencí.

Celkem může existovat 2^6=64 možných signatur, avšak ne všechny odpovídají sekvenční konzistenci - např. 000000, 001001 neodpovídají žádnému sekvenčnímu proložení instrukcí

P1: W(a)1 R(b)0 R(c)0

P2: W(b)1 R(a)1 R(c)0

P3: W(c)1 R(a)0 R(b)1

Signatura 001001 je nemožná v sekvenčně konzistenčním modelu

Sekvenční konzistence je příjemný model pro programování, avšak jeho výkonnost není příliš velká. Bylo dokázáno (Lipton & Sandberg, 1988) že je-li čas čtení *r*, čas zápisu *w* a čas přenosu zprávy (paketu) *t*, pak vždy platí $r+w \geq t$. Jinak řečeno - optimalizace protokolu pro čtení znamená delší čas pro zápis a naopak.

3.3.3 Kauzální konzistence (causal consistency)

Zápisy, které jsou potenciálně kauzálně vázané, musí být viděny všemi procesy ve stejném pořadí. Konkurenční zápisy mohou být viděny v různém pořadí.

Rozlišuje události, které jsou potenciálně závislé a které nikoliv. Pokud jeden proces W(x)a a druhý proces R(x) W(y)b pak W(y)b je kauzálně závislý na W(x)a. Operace, které nejsou kauzálně závislé jsou konkurenční.

P1: W(x)1 W(x)3

P2: R(x)1 W(x)2

P3: R(x)1 R(x)3 R(x)2

P4: R(x)1 R(x)2 R(x)3

Kauzální konzistence

Poslední příklad zobrazuje situaci, kdy W(x)2 a W(x)3 jsou konkurenční (tj. nejsou kauzálně závislé). Procesy P3 a P4 mohou potom změnu x číst v opačném pořadí. Tato sekvence splňuje podmínku kauzální konzistence, nesplňuje sekvenční a striktní.

P1: W(x)1 P1: W(x)1

P2: R(x)1 W(x)2 P2: W(x)2

P3: R(x)2 R(x)1 P3: R(x)2 R(x)1

P4: R(x)1 R(x)2 P4: R(x)1 R(x)2

PRAM konzistence **Kauzálně konzistentní sekvence**

Na druhém obrázku příkladu 7 není W(x)2 kauzálně závislý na W(x)1 a proto obrácené čtení R(x)1 a R(x)2 je vzhledem ke kauzální konzistenci korektní. Na prvním obrázku je W(x)2 kauzálně závislé na W(x)1 (přes R(x)1) a proto obrácené pořadí čtení porušuje kauzální konzistenci.

Implementace kauzálně konzistentní paměti vyžaduje udržování grafu závislostí zápisů na čtení.

3.3.4 PRAM konzistence (PRAM consistency)

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.

PRAM (Pipelined RAM) konzistence je snadná na implementaci - nezáleží na pořadí, v němž různé procesy vidí přístupy k paměti. Jediné, co je třeba dodržet, je pořadí zápisů z jednoho zdroje. Jinak řečeno - všechny zápisy generované různými procesory jsou konkurenční.

```
P1:                          P2:
a=1;                         b=1;
if(b==0) kill(P2);           if(a==0) kill(P1);
```

V případě PRAM konzistence je možný výsledek, že oba procesy budou zabity, a to v případě, že P1 přečte *b* dříve, než uvidí jeho zápis a P2 přečte *a* dříve, než uvidí jeho zápis. Tento výsledek není možný při libovolném uspořádání instrukcí, neodpovídá tedy sekvenční konzistenci, avšak vzhledem k PRAM konzistenci je korektní.

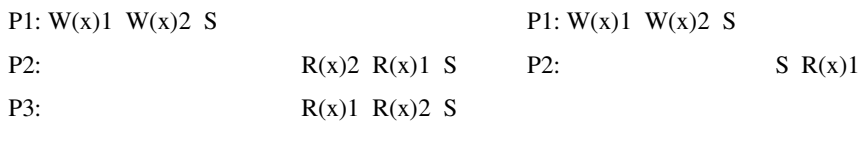
3.3.5 Slabá konzistence (weak consistency)

Výše uvedené konzistenční modely jsou stále pro mnoho aplikací velmi restriktivní (a tedy málo efektivní), neboť vyžadují propagaci všech zápisů všem procesům. Ne všechny aplikace vyžadují sledování všech zápisů, natož pak nějaké jejich pořadí. Např. proces v kritické sekci může v rychlé smyčce číst a zapisovat hodnoty nějakých proměnných. Ostatní procesy nemají důvod (nebo ani možnost) jednotlivé zápisy vidět, proto není nutné, aby byly všechny propagovány. Paměť však nemá možnost vědět, že nějaký proces je v kritické sekci a tudíž musí propagovat všechny zápisy.

Řešením je nechat proces ukončit kritickou sekci a poté zajistit rozeslání změn všem ostatním procesům. Toho může být dosaženo použitím speciálního druhu proměnné - **synchronizační proměnné**, která bude použita pro synchronizační účely.

1. Přístup k synchronizačním proměnným je sekvenčně konzistentní.
2. Přístup k synchronizační proměnné není povolen dokud neskončí všechny předchozí zápisy.
3. Přístup k datům není povolen dokud nebyly dokončeny všechny předchozí přístupy k synchronizačním proměnným

Bod 1 říká, že všechny procesy vidí všechny přístupy k synchronizační proměnné ve stejném pořadí. Bod 2 zaručuje, že před přístupem k synchronizační proměnné budou dokončeny všechny předchozí zápisy. Bod 3 říká, že při přístupu k obyčejným proměnným jsou dokončeny všechny předchozí přístupy k synchronizačním proměnným. Provedením synchronizace před čtením dat proto zajistí jejich aktuální verzi.



Slabá konzistence **Porušení slabé konzistence**

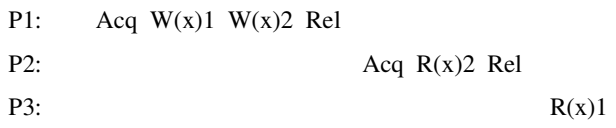
V levé části obrázku jsou události odpovídající slabé konzistenci - procesy P2 a P3 mohou před synchronizací vidět paměť v libovolném pořadí. Pravý obrázek ukazuje porušení slabé konzistence - proces P2 po synchronizaci vidí neaktuální hodnotu proměnné *x*.

3.3.6 Uvolňovací konzistence (release consistency)

Slabá konzistence má tu nevýhodu, že paměť při přístupu k synchronizační proměnné nepozná, zda se jedná o vstup nebo o výstup z kritické sekce. Pokaždé proto musí vykonat akce potřebné pro oba případy. K rozlišení těchto případů používá uvolňovací konzistence dvě operace - acquire (požadavek) a release (uvolnění).

1. Před běžným přístupem ke sdílené proměnné musí být úspěšně ukončeny předchozí požadavky procesu.
2. Před provedením uvolnění musí být ukončeny všechny předchozí zápisy i čtení prováděné procesem.
3. Požadavky a uvolnění musí být PRAM konzistentní.

Jestliže proces vydá požadavek, pak je zaručeno, že poté jsou všechny lokální kopie aktuální. Po uvolnění jsou propagovány všechny změny ostatním procesům. Jestliže jsou všechny podmínky splněny a požadavky a uvolnění se používají správě spárované, výsledek jakéhokoliv výpočtu je ekvivalentní sekvenčně konzistentní paměti.



Uvolňovací konzistence

Eager release consistency - po *release* se propagují změny všem procesům.

Lazy release consistency - *po release* se nic nepropaguje, až při *acquire* jiného procesu - časové značky.

3.3.7 Přístupová konzistence (entry consistency)

1. Požadovaný přístup procesu k synchronizační proměnné není povolen dokud nebyly provedeny všechny aktualizace chráněných sdílených dat procesu.
2. Exklusivní přístup procesu k synchronizační proměnné je povolen pouze v případě, že žádný jiný proces nepřistupuje k synchronizační proměnné, a to ani neexklusivně.
3. Po exkluzivním přístupu k synchronizační proměnné si příští neexkluzivní přístup libovolného procesu k synchronizační proměnné musí vyžádat aktuální kopii dat od vlastníka synchronizační proměnné.

Všechna sdílená data jsou vázána na synchronizační proměnnou. Při přístupu k synchronizační proměnné jsou synchronizována pouze ta data, která jsou vázána na synchronizační proměnnou.

Každá synchronizační proměnná má v každém okamžiku vlastníka - proces, který k ní naposledy přistupoval. Vlastník může opakovaně vystupovat do a vystupovat z kritické sekce bez nutnosti komunikace. Proces, který není vlastníkem, musí požádat o vlastnictví. Při přenosu vlastnictví se aktualizují hodnoty dat vázaných na synchronizační proměnnou. Přístup k datům a synchronizačním proměnným může být exkluzivní (read & write) nebo neexkluzivní (read only).

3.3.8 Shrnutí konzistenčních modelů

	Konzistence	Vlastnosti
a	Striktní	Absolutní časové uspořádání
	Sekvenční	Všechny události jsou vidět ve stejném pořadí
	Kauzální	Kauzálně vázané události jsou vidět ve stejném pořadí
	PRAM	Události jednoho procesu vidět ve stejném pořadí
b	Slabá	Sdílená data jsou konzistentní po synchronizaci
	Uvolňovací	Sdílená data jsou konzistentní po opuštění kritické sekce
	Přístupová	Sdílená data vázaná na kritickou sekci jsou konzistentní při vstupu do kritické sekce

Přehled základních konzistenčních modelů (a) bez synchronizačních operací (b) se synchronizačními operacemi

3.4 Možnosti aktualizace

Existují dvě základní možnosti jak propagovat aktualizaci dat z jednoho procesu ostatním. Jsou to **write-update** a **write-invalidate**. Je možné je aplikovat na různé modely konzistence.

Write-update

Aktualizace dat se provádí lokálně a pomocí skupinového adresování se rozesílá všem managerům replik, udržujícím kopie datových položek. Ti okamžitě modifikují data čtená lokálními procesy. Procesy čtou kopie lokálních dat bez potřeby komunikace. Kromě toho že může existovat více čtenářů, může také několik procesů zapisovat tatáž data ve stejnou dobu.

Model konzistentnosti paměti, který je realizován s metodou **write-update** závisí na mnoha faktorech, zejména na vlastnostech skupinového protokolu. Sekvenční konzistentnosti může být dosaženo použitím skupinového protokolu s úplným uspořádáním, kdy se všechny procesy dohodnou na pořadí aktualizací. Výhodou jsou laciné operace *čtení*, které probíhá lokálně. Nevýhodou je relativně vysoká cena implementace uspořádaného multicastu programově. Proto v konkrétních realizacích se používají skupinové protokoly s technickou podporou na přístupové úrovni.

Write-invalidate

Je obecně realizovaná v systémech typu multiple-reader/single-writer sharing. Data mohou být buď zpřístupněna více procesům najednou pro čtení, nebo jednomu procesu pro čtení i zápis. Položka, která je zpřístupněná pouze pro čtení může být libovolně kopírována do ostatních procesů. Pokud se proces pokusí zapsat do položky, pošle se nejprve do všech procesů zpráva, která ji zneplatní a teprve po potvrzení zprávy může proces položku aktualizovat. Tím jsou procesy ochráněny před čtením starých dat. Všechny procesy, které se ve stejnou dobu pokusí aktualizovat položku jsou blokovány do doby, než je aktualizace vybraným procesem dokončena. Výsledkem je, že procesy přistupují k položce při zápisu uspořádaně – podle pravidla první přijde, první je obsloužen. Toto přístupové schéma zajišťuje sekvenční konzistentnost.

Pokud se použije zneplatnění zpožděné, dosáhneme release-consistency.

3.5 Granularita (zrnitost, krystaličnost)

Otázka, která je spojena se strukturou DSM je zrnitost sdílení. Obecně sdílí všechny procesy celý obsah sdílené paměti. Během činnosti programů je však třeba sdílet pouze některé části sdílené paměti během nějakého času. Proto by bylo pro sdílenou paměť zahubující vždy přenášet celý obsah sdílené paměti jakmile proces k ní přistupuje a aktualizuje ji.

Nyní se zaměříme na implementace založené na stránkách. V **page-based DSM** podporuje hardware změny v adresním prostoru na úrovni stránek – v podstatě změnou odkazu na stránku v tabulce stránek. Velikost stránek je typicky do 8 kb, takže to je to množství dat, které je třeba přenést sítí k zajištění konzistentnosti. Nezáleží přitom na tom, byla-li aktualizována celá stránka nebo jen jedna slabika.

Použití menších velikostí stránek nevede ke zlepšení. V mnoha případech aktualizuje proces velké množství dat, což by vedlo k přenosu několika stránek po sobě. S více přenosy je spojena větší administrativa. Použití menších stránek vede k k velkému počtu jednotek, o které se musí systém starat.

Záležitost komplikuje i to, že procesy zápasí s tím, aby byla aktualizovaná data umístěna v jedné stránce, k čemuž přispívá větší rozsah stránky.

Předpokládejme dva procesy, které přistupují do jedné stránky, kdy první proces pouze čte data A, zatímco druhý proces pouze aktualizuje data B. Na aplikační úrovni v tomto případě neexistuje žádný konflikt. Avšak přesto musí být celá stránka přenášena, protože DSM implicitně neví, která místa ve stránce byla změněna. Tento jev je znám jako **falešné sdílení** – dva nebo více procesů sdílí části stránky, ale ke každé části ve skutečnosti přistupuje pouze jeden. V případě *write-invalidate* protokolu může vézt falešné sdílení k zbytečným zneplatněním. V případě *write-update* protokolu, kdy několik pisatelů falešně sdílí datové položky, může způsobit jejich přepsání staršími verzemi.

V praxi je volba jednotky sdílení volena na základě fyzikální velikosti stránek. Pokud je velikost stránky malá, může být základní jednotka brána jako souvislá posloupnost stránek. Návrh vhodné délky stránky respektující hranice stránek je určující pro počet stránkových přenosů za běhu programu.

3.6 Thrashing (porážka, mlácení, bytí)

Potenciálním problémem *write-invalidate* protokolů je **trashing**. Říkáme, že se objeví trashing, jestliže DSM tráví nadměrný množství času rušením platnosti a přenášením sdílených dat, v porovnání s časem stráveným aplikačním procesem vykonáváním užitečné práce. To se stane tehdy, jestliže několik procesů dokončuje aktualizaci týchž dat nebo při falešně sdílených datových jednotkách. Jestliže např. jeden proces opakovaně čte datovou položku a druhý proces ji pravidelně aktualizuje, pak je tato položka neustále přenášena od zapisujícího a zneplatňována čtenářem. Toto je případ sdílení dat, kdy write-invalidate je nevhodný a write-update je lepší.

4 Sekvenční konzistentnost a Ivy

Popis metod pro implementaci sekvenčně konzistentní, stránkové DSM.

4.1 Model systému

Základní model je uvažován tak, že soubor procesů sdílí segment DSM. Segment je mapován do téhož adresního prostoru v každém procesu. Hodnoty významných pointerů mohou být ukládány v segmentu. Procesy jsou spouštěny na počítačích s podporou stránkování. Budeme předpokládat, že existuje pouze jeden proces na počítači, který má přístup do DSM segmentu. Pokud existuje v uzlu více procesů přistupujících do sdílené paměti, pak využívají sdílenou paměť přímo pomocí tabulky stránek, kdy jeden segment sdílené paměti může být zapsán do více tabulek stránek. Jedinou komplikací je koordinace vybírání a rozšiřování aktualizací do stránky, při přístupu dvou nebo více procesů Tento popis tyto detaily ignoruje.

Pro aplikace je stránkování transparentní, mohou číst i zapisovat data do DSM. Avšak aby DSM udržela sekvenční konzistentnost pro zapisující a čtoucí procesy, omezuje práva přístupu do stránek. Jednotka pro řízení přístupu do stránkové paměti dovoluje nastavit přístupová práva na none, R/O a R/W. Pokud se proces pokusí nastavená práva porušit, vrátí se mu chybové hlášení podle typu přístupu. Jádro přesměruje chybu do ovladače specifikovaného v DSM pro každý proces. Ovladač pro zpracování chyby stránky zpracuje chybu speciálním způsobem.

Problém aktualizace zápisu

Pokud je DSM založeno na stránkování, pak pro sdílení se používá technika write-update pouze tehdy, mohou-li být zápisy bufferovány. Je to proto, že standardní systém zpracování chyb stránek není použitelný pro zpracování každého zápisu do paměti.

Předpokládejme, že je stránka chráněna proti zápisu. Pokud se proces pokusí do stránky zapsat, vznkne výpadek stránky a je volán příslušný ovladač. Tento ovladač v principu může zjistit chybující instrukci i adresu a hodnotu, která měla být zapsána. Může poslat multicast zprávu s aktualizací ještě předtím, než obnoví přístup pro zápis a ukončí přerušenou instrukci.

Ale nyní když byl zápisový přístup obnoven, nezpůsobí další aktualizace ve stránce její výpadek. Aby každý zápis způsobil výpadek stránky, je nutné aby ovladač nastavil proces do tracovacího režimu, ve kterém procesor generuje přerušení při každé instrukci. Ovladač trasování vypne práva zápisu do stránky a zase vypne

trasovací režim. Celé se opakuje při další chybě zápisu. Metoda je náchylná být velmi drahá. Během provádění procesu vzniká mnoho výjimek.

V praxi se write-update používá ve spojení se stránkovou pamětí pokud stránka není chráněna proti zápisu po počáteční chybě stránky a navíc je povoleno provézt několik zápisů předtím, než je aktualizovaná stránka rozšířena. Používá se technika bufferování zápisu. Navíc je možné proces propagace stránky zefektivnit tím, že si zapamatujeme kopii stránky před prvním zápisem a pak porovnáním původního stavu s aktualizovaným rozešleme pouze změny. Ostatní procesy generují aktualizovanou stránku také z kopie před aktualizací a ze seznamu odlišností.

4.2 Write invalidation

Algoritmus založený na neplatných datech využívá ochranu stránky k vynucení sdílení konzistentních dat. Pokud proces aktualizuje stránku, má pouze lokální práva číst a zapisovat. Ostatní procesy nemají k této stránce přístup. Pokud jeden nebo více procesů stránku čte, mají práva čtení. Ostatní procesy nemají ke stránce přístup. Proces s nejaktuálnější verzí stránky p je označen jako její vlastník – owner(p). Je to buď jeden zapisující, nebo jeden ze čtoucích. Soubor procesů, které vlastní kopii stránky p je nazýván copy-set – a označován copyset(p).

Pokud proces P_w se pokusí zapsat do stránky p, pro kterou nemá přístupová práva nebo pouze práva R/O, objeví se výpadek stránky, který je zpracován následovně:

- Stránka je přenesena do P_w, pokud dosud nemá aktualizovanou R/O kopii
- U všech ostatních kopií je zrušena platnost, práva přístupu jsou nastavena na no-access u všech členů copyset(p).
- Do všech členů copyset(p) se dosadí P_w
- Vlastník p owner(p) se nastaví na P_w
- DSM nastaví P_w pro proces p na práva R/W a spustí instrukci, která způsobila výjimku.
- Poznamenejme, že může nastat případ, kdy dva nebo více procesů s R/O kopiemi mohou vyvolat chybu zápisu ve stejném čase. R/O kopie stránky může být zastaralá pokud je vlastnictví je nakonec připouštět. K detekci kdy je současná R/O kopie stránky zastaralá můžeme použít sekvenční čísla spojená s každou stránkou. Tato čísla se inkrementují kdykoliv je vlastnictví přesunuto. Proces, který požaduje přístup pro zápis vloží sekvenční číslo své kopie R/O, pokud ji vlastní. Současný vlastník může pak říct zda byla stránka modifikována a proto je třeba, aby bylo poslána. Toto schéma je nazýváno „shrewd algorithm“ – mazaný algoritmus.

Pokud proces PR se pokouší číst stránku p ke které nemá přístupová práva nastane výpadek stránky při čtení. Procedura pro zpracování výpadku pracuje následovně

- Stránka je kopírována od vlastníka owner(p) do PR.
- Jestliže současný vlastník je jediný zapisující, pak zůstane jako vlastník p a jeho přístupová práva pro p jsou nastavena na R/O. Ponechaná práva přístupu pro čtení jsou potřebná v případě, že se proces pokusí číst stránku později. – bude mu ponechána současná verze stránky. Avšak, jak vlastník bude muset zpracovat následující požadavky pro stránku i když nebude přistupovat ke strance znovu. Takže může být vhodnější redukovat práva na no access a přenést vlastnictví na PR.
- Přidá PR do copyset(p)
- DSM v PR umístí stránku s právy R/O na vhodné místo v svém adresním prostoru a restartuje přerušenou instrukci.

Je možné pro, že se během algoritmu přenesení objeví druhý výpadek stránky. Aby byly přesuny zpracovány konzistentně, je nový požadavek na stránku zpracován až je probíhající přenos ukončen.

Daný popis dosud vysvětloval co musí být provedeno. Dále se budeme zabývat problémem jak realizovat zpracování výpadku stránky.

4.3 Invalidation protocols

K realizaci protokolu pro zrušení platnosti dat je třeba vyřešit následující problémy

1. jak vybrat owner(p) pro danou stránku p
2. kde ukládat copyset(p)

V literatuře je popsáno několik architektur a protokolů které dávají různé přístupy k těmto problémům. Nejednodušší je centralizovaný algoritmus. K uložení umístění vlastníků owner(p) pro každou stránku p se použije jeden server nazývaný manager. Může to být jeden z procesů běžících s aplikací, nebo to může být jakýkoliv jiný proces. V tomto algoritmu se množina copyset(p) ukládá v owner(p). Jsou tedy uloženy identifikátory a transportní adresy členů copyset(p). Pokud se objeví výpadek stránky, lokální proces pošle zprávu manageru, který obsahuje

číslo stránky a požadovaný typ přístupu. Pak proces čeká na odpověď. Manager zpracuje požadavek vyhledáním adresy vlastníka owner(p) a posláním požadavku vlastníku. Pokud dojde k výpadku stránky pro zápis, nastaví manager nového vlastníka – klienta, který vyslal požadavek. Následující požadavky jsou řazeny do fronty na klientu dokud není dokončen přenos vlastnictví do něj. Předchozí vlastník pošle stránku do klienta. V případě výpadku stránky pro zápis pošle také copyset(p) pro danou stránku. Klient provede zneplatnění dat jakmile přijme copyset. Posílá multicast požadavek členům copyset a očekává potvrzení od všech procesů že zneplatnění bylo provedeno. Multicast nemusí být uspořádaný. Původní vlastník nemusí být zahrnut v seznamu cílů, protože zneplatní sám sebe. Manager je úzké místo pro výkonnost systému a kritické místo pro chyby. Byly navrženy tři alternativy které dovolují aby zátěž spojená s managementem stránek byla rozdělena mezi počítač.

1. Fixní distribuovaný management stránek, kde existuje několik managerů, z nichž každý je funkčně ekvivalentní centrálním manageru, ale stránky jsou mezi managery statisticky rozděleny. Klienti pomocí hashovací funkce vypočtou index, určující manager, který se o stránku stará. Dle indexu pak z tabulky vyhledají adresu odpovídajícího managera. Toto schéma zlepší problém zatížení obecně, ale má nevýhodu v tom, že mapuje stránky na managery fixně, což nemusí být vhodné. Pokud není přístup ke stránkám rozložen rovnoměrně, jsou někteří manageři zatíženi více než jiní.
2. Distribuovaný algoritmus založený na multicastech. Multicast může být použit k eliminaci managerů kompletně. Pokud proces vyvolá výjimku, posílá pomocí multicastu požadavek na stránku všem ostatním procesům. Odpoví pouze proces, který stránku vlastní. Starost musí být věnována případu, kdy dva nebo více klientů požaduje tutéž stránku v tutéž dobu. Každý klient musí nakonec obdržet stránku, i když jeho požadavek je multicast během přenosu vlastnictví. Uvažujme dva klienty C1 a C2, kteří použijí multicast pro nalezení stránky vlastněné O. předpokládejme, že O přijme nejdříve požadavek od C1 a přenáší do něj vlastnictví. Dříve než stránka dorazí, obdrží O i C1 požadavek od C2. O zruší požadavek C2 protože již nevlastní stránku. C1 pozdrží zpracování požadavku od C2 dokud C1 neobdrží stránku – jinak odhodí požadavek, protože není vlastníkem a požadavek C2 se ztratí úplně. Avšak problém zůstává. Požadavek C1 bude zařazen do fronty na C2 prozatím. Po té, co C1 má nakonec stránku C2, přijme C2 požadavek od C1, ale ten je nyní zastaralý. Řešením je použít úplně uspořádaný multicast, tak že klienti mohou bezpečně zahazovat požadavky které přijdou před jejich vlastními (požadavky jsou doručovány jim samým tak jako ostatním procesům. Jiné řešení, které používá lacinější neuspořádaný multicast, ale které spotřebuje širší pásmo, je spojit každou stránku s vektorem časových razítek, jedno pro každý proces. Pokud je přenášeno vlastnictví stránky, tak je to časové razítko. Pokud proces obdrží vlastnictví, inkrementuje položku v časovém razítku. Pokud proces požaduje vlastnictví, přibalí poslední časové razítko pro danou stránku. V našem případě C2 může zrušit požadavek C1, protože položka C1 v časovém razítku požadavku je menší než ta, která přijde se stránkou. Ať se použije uspořádaný nebo neuspořádaný multicasty, má toto schéma nevýhodu multicast schémat – procesy které nejsou vlastníky stránek jsou přerušovány bezvýznamnými zprávami, které ničí čas procesoru.
3. Dynamický distribuovaný management

4.4 Dynamický distribuovaný algoritmus řízení

Algoritmus dynamického distribuovaného řízení dovoluje aby bylo vlastnictví stránky možno přenášet mezi procesy ale s využitím možnosti muticastu jako metody pro lokalizaci vlastníka stránky. Myšlenka je založena na rozdělení zátěže lokalizovaných stránek mezi ty počítače, které k nim přistupují. Každý proces udržuje pro každou stránku p příznak - tip jako by byl vlastníkem stránky – předpokládaný vlastník p, nebo probOwner(p). na začátku je každý proces seznámen s přesným umístěním stránky. Obecně jsou však tyto hodnoty pouze tipy, protože stránky mohou být přenášeny kdykoliv v libovolný okamžik. Tak jako v předchozím algoritmu, je vlastnictví přenášeno pouze tehdy, jestliže se objeví výpadek stránky z důvodu zápisu.

Vlastník stránky je umístěný v řetězcích tipů, které jsou nastaveny když je vlastnictví stránky přenášeno z počítače do počítače. Délka řetězce, která je dána počtem forwardovaných zpráv nutných k lokalizaci vlastníka se může prodlužovat neomezeně. Algoritmus tomu brání aktualizací tipů. Jakmile je dostupných více aktualizovaných hodnot. Tipy jsou aktualizovány a požadavky dále posílány následovně:

- Jakmile proces posílá vlastnictví ke stránce p jinému procesu, aktualizuje probOwner(p) na příjemce

Když proces zpracuje individuální požadavek na stránku p, aktualizuje probOwner(p) na požadujícího

- Jestliže proces, který požadoval přístup pro čtení ke stránce p ji přijme, aktualizuje probOwner(p) na vykonavatel
- Jestliže proces přijme požadavek na stránku p, kterou nevlastní, pošle požadavek na probOwner(p) a nuluje probOwner(p) jako požadujícího.

První tři aktualizace vychází z protokolu pro přenos vlastnictví stránky a za předpokladu R/O kopírování. Odůvodnění pro aktualizaci pokud posílá dále požadavek je v tom, že pro požadavky zápisu bude požadující brzy vlastník, pokud jím není již nyní.Proto probOwner aktualizace je prováděna pro požadavek přístupu čtení i zápisu.

Průměrná délka řetězce ukazatelů může být redukována periodickým vysíláním broadcastů s umístěním aktuálních vlastníků do všech procesů. Výsledkem je zhroucení všech řetězů až na délku jedna.

4.5 Trashing (porážka)

Omezení ztrát vzniklých zbytečným přesunováním vlastnictví stránek může řešit programátor vhodným uspořádáním příkazů. Řešení, které je vzhledem k programům transparentní je následující. Každá stránka je přidělována s malým časovým úsekem. Chce-li proces přistupovat ke stránce, je dovoleno zadržet přístup pro daný interval, který slouží jako typ časového okna. Ostatní požadavky na stránku jsou zadržovány. Velkou nevýhodou je stanovení délky časového úseku. Řešením je volba délky časového úseku dynamicky, pozorováním doby přístupu nebo sledováním délky front, ve kterých procesy čekají na stránku.

5 Release consistency a Munin

Předchozí algoritmus byl navržen pro dosažení sekvenční konzistentnosti. Výhodou sekvenční konzistentnosti je, že DSM zachovává představu sdílené paměti. Nevýhodou je, že je drahé ji realizovat. DSM systémy často vyžadují využití multicastu v realizaci ať jsou založeny na write-upddate nebo write-invalidate. Pro zneplatnění stačí neuspořádaný multicast.Lokalizace vlastníka zprávy má tendenci být drahá. Centrální manager, který zná umístění vlastníka každé stránky se jeví jako úzké místo. Pospojované ukazatele vyvolávají v průměru mnoho zpráv. Navíc algoritmus založený na zneplatnění může vézt k thrashing (těžká porážka, výprask).

Release consistency byla poprvé představena v datech multiprocesoru, který realizoval DSM hardwareově, původně využívaje write-invalidation protokol. Release consistency je volnější než sekvenční konzistentnost a levnější z hlediska realizace, ale má rozumnou sémantiku, která je tvárná pro programátory.

Myšlenka release consistency je redukovat režii DSM využitím faktu, že programátoři používají synchronizaci objektů jako je semaforů, zámků a barier. Realizace DSM může využít znalosti o přístupech k těmto objektům k povolení, aby byla paměť nekonzistentní v několika bodech, přičemž použití synchronizačních objektů nicméně udržuje konzistentnost na aplikační úrovni.

5.1 Přístupy do paměti

Abychom pochopili release consistency – nebo jakýkoliv jiný model paměti, který bere v úvahu synchronizaci – začneme s kategorizací přístupů do paměti podle jejich pravidel, pokud existují, v synchronizaci. Kromě toho budeme diskutovat o tom, jak má být prováděn asynchronní přístup do paměti aby získal na výkonnosti a dával jednoduchý operační model jaký efekt má přístup do paměti.

Jak jsme předtím říkali, DSM implementace v distribuovaných systémech pro obecné použití používá pro synchronizaci spíše přenos zpráv než sdílené proměnné, a to z důvodu efektivnosti. Ale může pomoci v následujících diskusích k pochopení synchronizace založené na sdílených proměnných. Následující pseudokód realizuje zámky s využitím funkce testAndSet . funkce nastaví zámek na 1 a vrací 0 nalezla-li nulu, jinak vrací 1. To dělá automaticky.

Typy přístupů do paměti

Hlavní rozdíl je mezi protikladným přístupem a neprotikladným, normálním přístupem. Dva přístupy jsou protikladné jestliže:

- Mohou se objevit souběžně
- Alespoň jeden je zápis

Takže dvě operace čtení nemohou být nikdy protikladné, čtení a zápis do téhož místa vytvářený dvěma procesy které se mezi operacemi synchronizují je neprotikladný.

Dále dělíme protikladný přístup na synchronizovaný a nesynchronizovaný:

- Synchronizovaný přístup jsou operace čtení nebo zápisu které přispějí k synchronizaci
- Asynchronní přístup jsou operace čtení a zápisu, které jsou konkurenční ale které nepřispívají k synchronizaci.

Synchronizovaný přístup je protikladný, protože potenciálně synchronizované procesy musí být schopné přistupovat k synchronizačním proměnným souběžně a musí je aktualizovat. Naopak operace čtení nemusí

dosáhnout synchronizace. Ale ne všechny protichůdné přístupy jsou synchronizační přístupy – existují třídy paralelních algoritmů ve kterých procesy vytváří protichůdný přístupy ke sdíleným proměnným aby je aktualizovaly a četly jiné výsledky a ne synchronizovaly.

Synchronizační přístup je dále dělen na přístup získat a přístup uvolnit, související s jejich rolí v potenciálním blokování procesu vytvářejícího přístup, nebo odblokování nějakých jiných procesů.

Provádění asynchronních operací

- Operace write mohou být realizovány asynchronně. Zapisovaná hodnota je bufferována před jejím propagováním a efekty zápisu jsou ostatními procesy pozorovány až později.
- DSM realizace mohou předem vybírat hodnoty v očekávání jejich čtení, aby předešly pozastavení procesu v době kdy potřebuje data.
- Procesory mohou provádět instrukce mimo pořadí. Čekají-li na ukončení prováděného přístupu do paměti, mohou vyvolat další instrukci pokud nezávisí na prováděné instrukci

Z pohledu asynchronních operací musíme rozlišovat mezi okamžikem kdy je operace read nebo write vyvolána a okamžikem, kdy je provedena nebo dokončena.

Budeme předpokládat že DSM je alespoň koherentní. To znamená, že každý proces souhlasí s pořadím operací zápis do téhož místa.Za tohoto předpokladu můžeme mluvit jednoznačně o uspořádání operací zápisu do daného místa.

Říkáme, že operace zápisu W(x)v byla provedena co se týče procesu P jestliže od tohoto okamžiku operace čtení procesu P vrátí hodnotu v zapsanou operací zápisu, nebo hodnotu nějakého následujícího zápisu do x.

Podobně říkáme, že operace čtení R(x)v byla provedena co se týče procesu P, pokud žádná následující operace zápisu vložená do téhož místa může případně nahradit hodnotu v kterou P čte. Např. P může mít předem vybranou hodnotu kterou bude číst. Nakonec operace o byla provedena jestliže je provedena s ohledem na všechny procesy.

5.2 Release consistency

Požadavky, se kterými se přejeme seznámit jsou:

- Udržovat synchronizační sémantiku objektů jako jsou zámky a bariery
- K získání výkonnosti povolujeme přiměřené množství asynchronnosti pro operace s pamětí
- K omezení překrývání mezi přístupy k paměti abychom garantovali provádění, které je ekvivalentní sekvenční konzistentnosti

Release consistency je definována následovně:

1. před běžnou operací čtení nebo zápisu je povoleno provézt s respektováním ostatních procesů, všechny předchozí získané přístupy musí být provedeny
2. předtím než je povoleno provedení operace *release* s respektováním ostatních procesů, všechny předchozí operace read a write musí být provedeny
3. operace *acquire* a *release* jsou sekvenčně konzistentní při vzájemném respektování.

Pravidla 1 a 2 garantují, že pokud se provádí operace *release*, žádný jiný proces požadující *lock* nemůže starou verzi dat modifikovaných procesem, který provádí *release*.To je konzistentní s očekáváním programátora, který uvolňuje zámek, potvrzuje, že proces skončil modifikací dat uvnitř kritické sekce.

5.3 Munin

Munin je DSM systém založený na programových "objektech", ale který může umístit každý objekt na samostatné stránce tak, aby mohl být použit MMU hardware pro detekování přístupu ke sdíleným objektům. Munin používá model s několika procesory, z nichž každý má stránkovaný lineární adresový prostor, ve kterém jedno nebo více vláken provádějí nepatrně modifikovaný multiprocesorový program. Cílem projektu Munin je, aby multiprocesorové programy po minimálních změnách efektivně běžely na multicomputerových systémech s použitím DSM.

Změny spočívají v označení deklarací sdílených proměnných klíčovým slovem **shared**, aby je kompilátor rozpoznal. Při tom může být také dodána informace, jak bude sdílená proměnná používána, což dovoluje provádění určitých optimalizací. Implicitně kompilátor ukládá každou sdílenou proměnnou na samostatnou stránku, ačkoliv rozsáhlé sdílené proměnné (jako jsou pole) mohou obsadit několik stránek. Programátor však může určit, že několik

sdílených proměnných se stejným typem sdílení bude uloženo v jedné stránce. Pro proměnné různých typů to není dovoleno, protože protokol používaný pro konzistenci stránek závisí na typu na nich uložených proměnných.

Spuštěním zkompilovaného programu je spuštěn kořenový proces (root process) na jednom z procesorů. Tento proces může vytvářet další procesy na jiných procesorech, které s ním pak poběží paralelně a budou s ním a mezi sebou komunikovat prostřednictvím sdílených proměnných tak, jako to dělají normální multiprocesorové programy. Je-li proces spuštěn na jednom procesoru, běží na něm po celou dobu (procesy nemigrují).

Přístup ke sdíleným proměnným je realizován normálními instrukcemi CPU pro čtení a zápis. Nejsou použity žádné zvláštní metody ochrany. Při pokusu o použití sdílené proměnné, která není přítomná, dojde k výpadku stránky a řízení převezme systém Munin.

Způsob synchronizace pro vzájemné vyloučení je úzce svázán s modelem paměťové konzistence. Lze deklarovat proměnné typu zámek, jejichž zamykání a odemykání se provádí pomocí knihovních funkcí. Dále jsou podporovány bariéry, podmíněné proměnné (condition variables) a další synchronizační proměnné.

5.4 Release Consistency

Munin je založen na softwarové implementaci release consistency. Munin dává uživatelům prostředky k tomu, aby mohli strukturovat své programy s použitím kritických oblastí (critical regions), definovaných dynamicky pomocí volání žádostí (acquire) a uvolnění (release). Zápisy do sdílených proměnných se musí provádět uvnitř kritických oblastí, čtení mohou být uvnitř i venku. Pokud je proces aktivní uvnitř kritické oblasti, systém negarantuje jakoukoliv konzistenci sdílených proměnných, ale v okamžiku, kdy je kritická oblast opuštěna, jsou sdílené proměnné modifikované od posledního uvolnění (release) zaktualizovány na všech počítačích. Programům, které dodržují tento model, se distribuovaná sdílená paměť jeví jako sekvenčně konzistentní.

Munin rozlišuje tři třídy proměnných:

1. Obyčejné proměnné
2. Sdílené datové proměnné
3. Synchronizační proměnné.

Obyčejné proměnné

nejsou sdíleny a mohou být čteny a zapisovány pouze procesem, který je vytvořil.

Sdílené datové proměnné

jsou viditelné více procesům a jeví sekvenčně konzistentní, pokud je všechny procesy používají pouze uvnitř kritických oblastí. Musí být deklarovány jako sdílené, ale přistupuje se k nim pomocí obvyklých instrukcí pro čtení a zápis.

Synchronizační proměnné

jako jsou zámky a bariéry, jsou zvláštní a přístupné pouze prostřednictvím systémových volání (lock, unlock pro zámky, increment a wait pro bariéry). Tyto procedury zajišťují, že distribuovaná sdílená paměť vůbec pracuje.

Př.: *Release consistence v Muninu* pro tři kooperující procesy, každý běžící na vlastním počítači. V určitém okamžiku chce proces 1 vstoupit do kritické oblasti chráněné zámkem L (všechny kritické oblasti musí být chráněny stejnou synchronizační proměnnou). Příkaz *Lock* zajistí, že žádný správně se chovající proces není právě ve stejné kritické oblasti. Potom je proveden přístup ke třem sdíleným proměnným (a, b, c) použitím obvyklých instrukcí. Nakonec je zavolán příkaz *Unlock*, který způsobí propagaci výsledků všem počítačům, které vlastní kopie proměnných a, b, c. Tyto změny jsou zabaleny do minimálního počtu zpráv. Přístupy k těmto proměnným na jiných počítačích, zatímco je proces 1 uvnitř kritické oblasti, dají nedefinované výsledky.

5.5 Protokoly Muninu

Kromě používání release consistency, používá Munin další techniky ke zvýšení výkonu. Mezi nimi je nejdůležitější to, že programátor může zařadit každou sdílenou proměnnou do jedné ze 4 kategorií:

1. **Read-only** (pouze ke čtení)
2. **Migratory (**migrační)
3. **Write-shared** (sdílená pro zápis)
4. **Conventional** (konvenční).

Původně Munin podporoval i další kategorie, ale zkušenost ukázala, že mají pouze okrajový význam, a proto se od nich upustilo. Každý počítač spravuje adresář obsahující odkazy na všechny proměnné, kde se mimo jiné říká o každé proměnné, do které kategorie patří. Pro každou kategorii je použit jiný protokol.

Read-only

proměnné jsou nejsnadnější. Když odkaz na read-only proměnnou způsobí výpadek stránky, Munin vyhledá proměnnou v adresáři, najde kdo ji vlastní a požádá vlastníka o kopii potřebné stránky. Protože se stránky obsahující read-only proměnné nemění (poté co byly inicializovány), nedochází k problémům s konzistencí. Read-only proměnné jsou chráněny MMU hardwarem. Pokus o zápis do takové proměnné způsobí závažnou chybu (fatal error).

Migratory

sdílené proměnné používají protokol acquire/release. Ten je vidět například u zámků na př1. Jsou používány uvnitř kritických oblastí a musí být chráněny synchronizačními proměnnými. Myšlenka spočívá v tom, že tyto proměnné migrují z počítače na počítač, když procesy vstupují a vystupují z kritických oblastí. Nejsou replikovány.

Před použitím migrační proměnné je nutné nejprve pro sebe získat zámek. Jestliže je proměnná čtena, vytvoří se kopie její stránky na počítači, kde se čtení provádí a původní kopie se smaže. Optimalizací tohoto způsobu komunikace může být, že zámek svázaný s migrační proměnnou se zašle spolu s ní v jedné zprávě, čímž se eliminují zprávy navíc.

Jestliže programátor usoudí, že je bezpečné, aby více procesů najednou zapisovalo do téže proměnné, použijí se proměnné **sdílené pro zápis**. Například více procesů může paralelně zapisovat do pole tak, že každý proces přistupuje pouze do určité oblasti pole. Na začátku jsou stránky obsahující write-shared proměnné označeny jako read-only. Mohou už v této chvíli být replikované. Dojde-li k zápisu, obsluha výjimky (zápis do read-only stránky) vytvoří kopii stránky nazývanou "dvojče" (twin), označí stránku jako "dirty" a umožní, aby MMU mohlo provádět zápisy do stránky.

Když dojde k uvolnění (pozn.: z textu není jasné, kdy to je, protože tady není kritická sekce!), Munin provede srovnání dirty stránky s jejím dvojčetem slovo po slovu a pošle změny (spolu se všemi migračními stránkami) všem procesům, které je potřebují. Pak nastaví ochranu stránky zpět na read-only.

Když příjemce obdrží seznam změn, zkontroluje každou stránku, zda ji neměnil také. Jestliže ji neměnil, akceptuje změny, které přišly. Jestliže byla stránka změněna také lokálně, provede se porovnání lokální stránky, jejího dvojčete a příchozího seznamu slovo po slovu. Jestliže bylo slovo změněno lokálně a příchozí měněn nebyl, pak příchozí slovo přepíše lokální. Jestliže byly změněny obě slova (lokální i příchozí), je signalizována běhová chyba. Jestliže neexistují žádné konflikty, výsledek slévání stránek přepíše lokální stránku a běh procesu pokračuje.

Se sdílenými proměnnými, u kterých není uvedena žádná ze čtyř kategorií, se zachází stejným způsobem jako v **konvenčních** DSM systémech založených na stránkování. Existuje pouze jedna kopie každé zapisované stránky, která je přesouvána z procesu do procesu na požádání. Read-only stránky jsou replikovány dle potřeby.

Př.: *Chování systému s ryze sekvenční konzistencí*. Nejprve se oba procesy zastaví na bariéře. Bariéra může být implementována pomocí správce bariéry (barrier manager), kterému každý proces zašle zprávu a je blokován, dokud nedostane odpověď. Správce bariéry nepošle žádnou odpověď, dokud všechny procesy nedorazí k bariéře. Po minutí bariéry, může proces 1 začít zápisem do prvku a[0]. Pak se proces 2 pokusí zapsat do a[1], což způsobí výpadek stránky následovaný načtením stránky se sdíleným polem. Potom se proces 1 pokusí o zápis do prvku a[2], čímž způsobí další výpadek a tak dál. Při troše smůly může každý zápis znamenat přenos celé stránky, což způsobí velký provoz na síti.

Př.: *Použití release consistency*. Opět oba procesy nejprve minou bariéru. První zápis do a[0] způsobí vytvoření dvojčete (twin page) pro proces 1. Podobně první zápis do a[1] způsobí vytvoření dvojčete pro proces 2. V této chvíli nejsou potřebné žádné přenosy stránek mezi počítači, každý proces ukládá do své kopie pole a bez jakýchkoliv výpadků stránek. V momentě, kdy každý z procesů dorazí na druhou bariéru, jsou nalezeny rozdíly mezi jeho aktuálními a původními (uloženy ve dvojčeti) hodnotami pole **a**. Rozdíly jsou poslány všem procesům, o kterých se ví, že mají o tyto stránky zájem. Tyto procesy pak mohou poslat rozdíly všem dalším procesům, které se o ně zajímají a o kterých nevěděl zdroj změn. Každý příjemce slije změny se svou vlastní verzí stránky. Konflikty končí běhovou chybou.

Poté co proces zveřejnil tímto způsobem změny, pošle zprávu správci bariéry a čeká na odpověď. Ve chvíli, kdy všechny procesy rozeslaly své změny a dorazily k bariéře, správce bariéry rozešle odpovědi a všichni můžou pokračovat. Při této metodě je přeprava stránek potřebná pouze při dosažení bariéry.

5.6 Adresáře

Munin používá adresáře k nalezení stránek obsahujících sdílené proměnné. Když je vyvolána výjimka způsobená odkazem na sdílenou proměnnou, Munin zakóduje virtuální adresu, která způsobila výjimku, tak aby našel záznam o proměnné v adresáři sdílených proměnných. Ze záznamu zjistí, do které kategorie proměnná patří, zda je přítomná její lokální kopie a kdo ji pravděpodobně vlastní. Write-shared stránky nemusí mít nutně jen jednoho vlastníka. Pro konvenční sdílené proměnné je to poslední proces, který požadoval přístup pro zápis. Pro migrační proměnné, je vlastníkem proces, který v této chvíli proměnnou drží.

Pro nalezení pravděpodobného vlastníka je použit následující algoritmus. Po startu procesu v Mininu vlastní všechny sdílené proměnné kořenový proces (root process). Když pak proces P1 přistoupí ke sdílené proměnné, způsobí výpadek, který zašle rootu zprávu se žádostí o tuto proměnnou. Root mu dá stránku, kterou požaduje, a poznamená si, že P1 je nyní vlastník. Jestliže potom P2 žádá tutéž stránku, root mu odpoví, že P1 je pravděpodobný vlastník. Když P2 požádá proces P1 o proměnnou, dostane ji. Jestliže P2 chce do proměnné psát nebo stránka je migratory, stane se P2 novým vlastníkem a P1 si to poznamená. Nyní můžeme předpokládat, že P3 a P4 úspěšně žádají o stránku. Řetěz procesů se tím prodlouží. Nyní P1 vyžaduje proměnnou znova. Protože si myslí, že vlastník je stále proces P2, pošle zprávu procesu P2 a dozví se od něj, že proměnnou vlastní proces P3. Dále P1 pokračuje v dotazech podle řetízku procesů, až požadovanou stránku dostane a řetěz se změní jako na obrázku Takto nakonec každý proces získá nějakou představu o tom, kdo by pravděpodobně mohl proměnnou vlastnit, a prohledáním řetízku najde skutečného vlastníka.

Adresáře se také používají k udržování množin kopií (copysets). Množina kopií však nemusí být zcela konzistentní. Pro příklad předpokládejme, že každý z procesů P1 a P2 drží nějakou write-shared proměnnou a každý ví o tom druhém. Pak si P3 vyžádá od vlastníka P1 svou vlastní kopii a dostane ji. P1 si poznamená, že P3 má kopii, ale neřekne to P2. Později P4, který si myslí, že P2 je vlastník, získá kopii, což způsobí, že do copyset u procesu P2 je přidán P4. V tomto okamžiku žádný z procesů nemá kompletní seznam vlastníků dané stránky.

Nicméně je možné udržovat konzistenci. Představme si, že P4 nyní uvolní zámek, tím pošle změny procesu P2. Potvrzovací zpráva od P2 procesu P4 obsahuje upozornění, že P1 má také kopii. Když P4 kontaktuje P1, dozví se také o procesu P3. Tímto způsobem si nakonec P4 zkonstruuje celý copyset tak, aby mohl zaktualizovat všechny kopie.

Ke snížení režie, aby se nemusely posílat aktualizace procesům, které se již dále nezajímají o konkrétní write-shared stránky, se používá algoritmus založený na časování. Když proces drží stránku, nepřistupuje k ní po daný časový interval a dostane aktualizační stránku, stránku zahodí. Příště, když dostane zprávu o aktualizaci pro zahozenou stránku, řekne procesu, který mu aktualizaci poslal, že už kopii nevlastní, a ten ho vyřadí ze své copyset. řetěz pravděpodobných vlastníků se používá k označení poslední kopie, která nemůže být odhozena bez nalezení nového vlastníka nebo zápisu na disk. Tento mechanismus zajistí, že stránka nemůže být zahozena všemi procesy a tím ztracena.

5.7 Synchronizace

Munin si udržuje druhý adresář pro synchronizační proměnné. Tyto proměnné jsou rozmístěny analogickým způsobem jako obyčejné sdílené proměnné. Zámky fungují jako by byly centralizované, ale v praxi je použita distribuovaná implementace, aby se vyhnulo příliš velké komunikaci s jedním počítačem.

Když chce proces získat zámek, nejprve zjistí, zda jej nevlastní sám. Pokud ano a zámek je volný, požadavek je splněn. Jestliže zámek není lokální, je nalezen prostřednictvím adresáře pro synchronizační proměnné, kde je uložen odkaz na pravděpodobného vlastníka. Je-li zámek volný, požadavek je uspokojen. Jestliže není volný, žádající proces je zařazen na konec fronty. Takto každý proces zná identitu procesu, který jej ve frontě následuje. Když je zámek uvolněn, vlastník ho předá dalšímu procesu v seznamu.

Bariéry jsou implementovány centrálním serverem. Při vytvoření bariéry je nutné zadat, na kolik procesů se bude na bariéře čekat. Proces, který ukončil určitou část svého výpočtu, zašle bariérovému serveru zprávu s požadavkem o čekání (wait). Když čeká požadovaný počet procesů, všem z nich jsou zaslány zprávy, které je uvolní.

5.8 Midway

Midway je systém pro distribuované sdílení paměti založený na sdílení jednotlivých datových struktur. V některých rysech je podobný Muninu, ale má i některé vlastní zajímavé rysy. Jeho cílem je umožnit, aby stávající multiprocesorové programy efektivně běžely na multicomputeru po provedení pouze malých změn v kódu. Programy v Midway jsou obyčejné konvenční programy psané v jazyce C, C++ nebo ML vybavené určitými přídavnými informacemi, které poskytuje programátor. Paralelismus je v programech pro Midway vyjádřen pomocí

balíku „céčkových“ funkcí pro implementaci multithreadingu v Machu (Mach C-threads package). Vlákno může vytvořit jedno nebo více dalších vláken. Vlákna potomků běží paralelně s jejich rodičem a mezi sebou. Potenciálně může každé vlákno běžet na jiném stroji, tedy každé vlákno jako samostatný proces. Všechna vlákna sdílejí stejný lineární adresový prostor, který obsahuje jak privátní, tak i sdílená data. Úkolem Midway je efektivním způsobem udržovat konzistenci sdílených dat.

5.9 Entry Consistency

Konzistence je udržována na základě požadavku, aby všechny přístupy ke sdíleným proměnným a datovým strukturám byly prováděny uvnitř zvláštního druhu kritických sekcí, o kterých ví runtime systém Midway. Každá z těchto kritických sekcí je strážena zvláštní synchronizační proměnnou, obecně zámkem, ale možná také bariérou. Každá sdílená proměnná, ke které se přistupuje uvnitř kritické sekce, musí být explicitně svázána se zámkem (bariérou) stráží cí tuto kritickou sekci prostřednictvím volání procedury. Potom vstupuje-li nebo vystupuje-li proces do (z) kritické sekce, Midway přesně ví, které sdílené proměnné potenciálně mohly být použity.

Midway podporuje entry consistency, která pracuje následujícím způsobem. Aby proces mohl přistupovat ke sdíleným datům, normálně vstoupí do kritické sekce voláním knihovní funkce "lock", jejímž parametrem je proměnná typu zámek. Při volání se také určí, zda proces požaduje exkluzivní nebo neexkluzivní přístup. Exkluzivní uzamknutí je třeba, bude-li jedna nebo více sdílených proměnných aktualizováno . Pokud bude proces sdílené proměnné pouze číst, stačí neexkluzivní uzamknutí, které umožní, aby více procesů vstoupilo najednou do stejné kritické oblasti. Protože hodnota žádné sdílené proměnné se nebude měnit, nemůže takto dojít k žádné škodě.

Při volání "lock" Midway runtime systém zajistí zámek a navíc zaktualizuje hodnoty všech sdílených proměnných spojených s tímto zámkem. Aby to zajistil, bude pravděpodobně potřebovat poslat zprávy ostatním procesům, aby získaly novější hodnoty. Když dostane všechny odpovědi, poskytne zámek volajícímu procesu (za předpokladu, že nejsou další konflikty) a ten začne provádět kritickou oblast. Když proces dokončí kritickou sekci, uvolní zámek. Narozdíl od release consistency se při uvolnění neuskutečňuje žádná komunikace, modifikované sdílené proměnné nejsou poslány na ostatní počítače. Pouze až další proces následně získá zámek, jsou nová data přenesena.

Aby entry consistency mohla fungovat, požaduje Midway, aby programy splňovaly tři charakteristiky, které multiprocesorové programy nemají:

1. Sdílené proměnné musí být deklarovány pomocí klíčového slova "shared".
2. Každá sdílená proměnná musí být svázána se zámkem nebo bariérou.
3. Přístup ke sdíleným proměnným musí být pouze uvnitř kritických sekcí.

Tyto věci vyžadují zvláštní úsilí od programátora. Jestliže tato pravidla nejsou plně dodržována, není generováno žádné chybové hlášení, ale program může dojít k chybným výsledkům. Protože programování tímto způsobem je poněkud náchylné k chybám, zvláště v případě, že upravujeme staré multiprocesorové programy, kterým už vlastně nikdo nerozumí, podporuje Midway také sekvenční konzistenci a release konzistenci. Tyto modely požadují méně detailní informace pro korektní provádění operací.

Na dodatečné informace požadované Midway by mělo být pohlíženo jako na součást smlouvy mezi softwarem a pamětí.

5.10 Implementace

Při vstupu do kritické sekce musí runtime systém Midway nejprve získat patřičný zámek. K získání exkluzivního přístupu je nutné nalézt vlastníka zámku, což je poslední proces, který jej uzamknul exkluzivně. Každý proces si udržuje pravděpodobného vlastníka, stejně jako to dělá IVY a Munin, čímž vzniká řetěz následných vlastníků končící aktuálním vlastníkem. Jestliže tento proces právě nepoužívá zámek, je vlastnictví zámku přeneseno na proces, který vznesl požadavek. Je-li zámek právě používán, žádající proces je blokován, dokud zámek nebude uvolněn. K neexkluzivnímu uzamknutí stačí kontaktovat libovolný proces, který zámek zrovna vlastní. Bariéry jsou spravovány centralizovaným bariérovým správcem.

Současně s uzamknutím zámku jsou žádajícímu procesu aktualizovány hodnoty všech sdílených proměnných. V nejjednodušším protokolu pouze starý vlastník všem rozešle kopie. Midway však používá optimalizace ke snížení objemu dat, která musí být přenesena. Předpokládejme, že požadavek o zámek byl uskutečněn v čase T1 a předchozí požadavek stejného procesu byl v čase T0. Pak jsou přeneseny pouze ty proměnné, které byly modifikovány od okamžiku T0, protože ostatní proces už má.

Tato strategie s sebou přináší otázku, jak si má systém zjistit, která data byla modifikována a kdy. K udržování informace, které sdílené proměnné byly měněny, je třeba použít zvláštní kompilátor, který generuje kód, jež za běhu programu udržuje tabulku s položkami pro všechny sdílené proměnné v programu. Kdykoli je aktualizována

sdílená proměnná, změna je zaznamenána v tabulce. Jestliže není k dispozici tento specializovaný kompilátor, je k detekci zápisů do sdílených dat použit MMU hardware jako v Muninu.

Čas každé změny je udržován pomocí protokolu časových známek (timestamp protocol) založeném na *Lamportově algoritmu* (1978). Každý počítač si udržuje logické hodiny, které jsou zvyšovány, kdykoliv je poslána zpráva, a jejich hodnota je vložena do zprávy. Když přijde zpráva, příjemce nastaví své logické hodiny na větší z dvojice hodnot: čas v přijaté zprávě, aktuální čas na lokálních hodinách. Použitím těchto hodin je čas efektivně rozdělen do intervalů definovaných pomocí zasílání zpráv. Při požadavku na získání zámku, zadá žádající procesor čas předchozího uzamčení a zeptá se na všechny relevantní sdílené proměnné, zda od této doby byly modifikovány či nikoliv.

Použitím takto implementované entry consistency dává potencionálně výborný výkon, protože komunikace je třeba pouze, když proces provádí zamykání. Dále pouze ty proměnné, jejichž hodnoty jsou neaktuální, musí být přenášeny. Obzvláště vstoupí-li proces do kritické sekce, opustí ji a opět do ní vstoupí, není třeba žádná komunikace. Tento vzor chování je obvyklý v paralelním programování, proto potenciální zisk je zde podstatný. Cena zaplacená za tento výkon je programátorský interface, který je komplexnější a více náchylný k chybám než při používání jiných modelů konzistence.

6 Ostatní modely konzistentnosti (důslednost)

Modely paměťové konzistentnosti mohou být rozděleny na uniform models, které nerozlišují mezi typy přístupů do paměti a hybridní modely, které rozlišují mezi normálním a synchronizovaným přístupem.

Kromě volné a sekvenční konzistentnosti existují následující modely.

Causal consistency (příčinná konzistence)

Čtení a zápisy mohou být spojeny předchozími vztahy. To je definováno k udržení mezi paměťovými operacemi kdy buď jsou prováděny jedním procesem nebo proces čte hodnotu zapsanou jiným procesem, nebo existuje posloupnost takových operací spojujících dvě operace. Omezení modelu je to, že hodnota vrácená čtením musí být konzistentní s se vztahem který se stal předtím.

Processor consistency

Nejjednodušší cesta k myšlené procesorové konzistentnosti je ta, že paměť je souvislá a že všechny procesy souhlasí s uspořádáním libovolných dvou zápisových přístupů vytvořených jedním procesem – tj. dohoda s jejich programovými příkazy.

Pipelined RAM

Všechny procesory souhlasí s požadavky zápisů vložených libovolným z procesorů.

Entry consistency

V tomto modelu je každá sdílená proměnná spojena se synchronizačním objektem jako je zámek, který řídí přístup k proměnné. Každý proces, který nejdříve požaduje zámek má zajištěno, že čte poslední hodnotu proměnné. Proces, který chce zapisovat, musí nejdříve získat zámek v exkluzivním režimu tak, aby k proměnné měl přístup pouze tento proces. Souběžně může číst několik procesů tak, že si požádají o přidělení zámku v neexkluzivním režimu. Falešnému sdílení v release konzistency je předcházeno za cenu zvýšení složitosti programu.

Scope consistency

Pokouší se zjednodušit programový model entry consistency. Proměnné jsou spojeny se synchronizačními objekty většinou automaticky namísto spoléháním se na programátora že spojí s proměnnými explicitně zámky. Např. systém může monitorovat které proměnné jsou aktualizovány v kritických sekcích.

Weak consistency

Nerozlišuje mezi synchronizačními přístupy acquire – osvojit a release – uvonit. Zaručuje, že všechny obvyklé předchozí přístupy se ukončí před ukončením jakéhokoliv typu synchronizovaného přístupu.