

IB109 Návrh a implementace paralelních systémů

Implementace Lock-Free datových struktur

Jiří Barnat

Klasická škola vícevláknového programování

- Přístup ke sdíleným datům musí být chráněný.
- Přístupy k datům se musí serializovat s využitím různých synchronizačních primitiv (mutexy, semaforey, monitory).
- Vlákna operují s daty tak, aby se tyto operace jevily ostatním vláknům jako atomické operace.

Problémy

- Prodlevy při přístupu ke sdíleným datům.
- Uvážnutí, živost, férovost.
- Korektnost implementace.
- Atomicita operací. (Je `++i` atomické?)

Lock-free programování

- Programování paralelních (vícevláknových) aplikací bez použití zamykání nebo jiných makro-synchronizačních mechanismů.

Vlastnosti lock-free programování

- Používá se (typicky) jedna jediná atomická konstrukce/instrukce
- Minimální prodlevy související s přístupem k datům
- Neexistuje uváznutí, je garantována živost
- Algoritmicky obtížnější uvažování
- Korektnost algoritmu náchylná na optimalizace překladače

Wait-free procedura

- Procedura, která bez ohledu na souběh dvou a více vláken dokončí svou činnost v konečném čase, tj. neexistuje souběh, který by nutil proceduru nekonečně dlouho čekat, či provádět nekonečně mnoho operací.

Lock-free procedura

- Procedura, která garantuje, že při libovolném souběhu mnoha soupeřících vláken, vždy alespoň jedno vlákno úspěšně dokončí svou činnost. Některá soupeřící vlákna mohou být libovolně dlouho nucena odkládat dokončení své činnosti.

Maurice Herlihy

- Článek: *Wait-Free Synchronization* (1991)
- Ukázal, že konstrukce jako
 - *test-and-set*
 - *swap*
 - *fetch-and-add*
 - *fronty s atomickými operacemi vložení a výběru*

nejdou vhodné pro budování lock-free datových struktur pro vícevláknové aplikace.

- Ukázal, že existují konstrukce, které vhodné jsou (např. CAS).
- Dijkstrova cena za distribuované počítání (2003)
<http://www.podc.org/dijkstra/2003.html>

Důsledek

- Současné procesory mají odpovídající HW podporu pro CAS.

Sémantika daná pseudo-kódem:

- **template** <class **T**>
bool CAS(**T*** addr, **T** exp, **T** val) {
 if (*addr == exp) {
 *addr = val;
 return true;
 }
 return false;
}

Slovní popis

- CAS porovná obsah specifikované paměťové adresy **addr** s očekávanou hodnotou **exp** a v případě rovnosti nahradí obsah paměťové adresy novou hodnotou **val**. O úspěchu či neúspěchu informuje uživatele návratovou hodnotou. Celá procedura proběhne atomicky.

Postup při přístupu ke sdíleným datům

- Přečtu stávající hodnotu sdíleného objektu
- Připravím novou hodnotu sdíleného objektu
- Aplikuji instrukci CAS

Návratová hodnota

- *True* – Objekt nebyl v mezičase modifikován, nově vypočítaná hodnota je platná a je uložena ve sdíleném objektu.
- *False* – Objekt byl v mezičase modifikován (z jiného vlákna), instrukce CAS neměla žádný efekt a je nutné celý postup opakovat.

Klíčová vlastnost

- Modifikace objektu proběhnoucí mezi načtením hodnoty objektu a aplikací instrukce CAS nesmí vyprodukovat tutéž hodnotu sdíleného objektu.

Možný chybový scénář

- Hodnota objektu, načtená vláknem A za účelem použití v následné instrukci CAS, je x .
- Před použitím instrukce CAS vláknem A, je objekt modifikována jinými vlákny, tj. nabývá hodnot různých od x .
- V okamžiku aplikace instrukce CAS vláknem A, má objekt opět hodnotu x .
- Vláknem A nepozná, že se hodnota objektu změnila.
- Následná aplikace instrukce CAS uspěje.

Provádění instrukcí mimo pořadí

- Pokud používáme CAS na zpřístupnění nějakých dat, je potřeba zajistit, aby předcházející inicializace proměnných byly již v okamžiku vykonání instrukce CAS vykonány.
- Vyžaduje použití paměťové bariéry.
- Dotčené proměnné musejí být označeny jako nestálé.

Cena

- Použití CAS odstranilo režii související se zamykáním.
- Zůstává však režie související s koherencí cache paměti.

Win32

- `InterlockedCompareExchange(...)`

Asembler i386, (pro x86_64 nutné přejmenovat edx na rdx)

- ```
inline int32_t compareAndSwap
(volatile int32_t & v, int32_t exValue, int32_t cmpValue)
{ asm volatile ("lock; cmpxchg :%%ecx, (%%edx)" : "=a" (cmpValue)
 "d" (&v), "a" (cmpValue), "c" (exValue));
 return cmpValue;
}
```

## GCC – zabudované funkce

- `bool __sync_bool_compare_and_swap (T *ptr, T old, T new, ...)`
- `T __sync_val_compare_and_swap (T *ptr, T old, T new, ...)`

## WRRM Mapa – Příklad

## Write Rarely Read Many Mapa

- Zprostředkovává překlad jedné entity na jinou. (Klíč→Hodnota).
- Příklad – kurz Koruny vzhledem k jiným měnám
  - Mění se jednou denně.
  - Používá se při každé transakci.

## Možné implementace s využitím STL

- map, hash\_map
- assoc\_vector (uspořádané dvojice)

## Použití

- Map <Key, Value > mojeMapa;

# Implementace s použitím Mutexů

```
template <class K, class V>
class WRRMMap {
 Mutex mtx_;
 Map <K,V> map_;
public:

 V Lookup(constK& k) {
 Lock lock(mtx_);
 return map_[k];
 }

 void Update(const K& k, const V& v) {
 Lock lock(mtx_);
 map_.insert(make_pair(k,v));
 }
};
```

## Operace čtení

- Probíhá zcela bez zamykání.

## Operace zápisu

- Vytvoření kopie stávající mapy.
- Modifikace/přidání dvojice do vytvořené kopie.
- Atomická záměna nové verze mapy za předcházející.

## Reálné omezení CAS

- Obecné použití schématu CAS na WRRMMap by vyžadovalo atomickou změnu relativně rozsáhlé oblasti paměti.
- HW podpora pro CAS je omezena na několik bytů (typicky jedno, nebo dvě slova procesoru).
- Atomickou záměnu provedeme přes ukazatel.

# Implementace s použitím instrukce CAS

```
template <class K, class V>
class WRRMMap {
 Map <K,V>* pMap_;
public:
 V Lookup(constK& k) {
 return (*pMap_) [k];
 }
 void Update(const K& k, const V& v) {
 Map <K,V>* pNew=0
 do {
 Map <K,V>* pOld = pMap_;
 delete pNew; //if (pNew==0) nothing happens
 pNew = new Map<K,V>(*pOld);
 (*pNew)[k] = v;
 } while (!CAS(&pMap_, pOld, pNew));
 // DON'T delete pOld;
 }
};
```

## Proč je nutná instrukce CAS a nestačí jen $pOld = pNew$ ?

- Vlákno A udělá kopii mapy.
- Vlákno B udělá kopii mapy, vloží nový klíč a dokončí operaci.
- Vlákno A vloží nový klíč.
- Vlákno A nahradí ukazatel, vše, co vložilo B, je ztraceno.

## Update

- Je lock-free, ale není wait-free.

## Správa paměti

- Update nemůže uvolnit starou kopii datové struktury, jiné vlákno může nad datovou strukturou provádět operaci čtení.
- Možné řešení: Garbage collector (JAVA)



## Odložená dealokace paměti

- Místo delete, se spustí (asynchronně) nové vlákno.
- Nové vlákno počká 200ms a pak provede dealokaci.

## Myšlenka

- Nové operace probíhají nad novou kopií, za 200ms se všechny započaté operace nad starou kopií dokončí a bude bezpečné strukturu dealokovat.

## Problémy

- Krátkodobé intenzivní přepisování hodnot nebo vkládání nových hodnot může způsobit netriviální paměťové nároky.
- Není garantováno, že se veškeré operace čtení z jiných vláken za daný časový limit dokončí.

## Nápad

- Napodobíme metodu používanou při automatickém uvolňování paměti k tomu, abychom mohli explicitně dealokovat strukturu.
- Počítání odkazů – s každým ukazatelem je svázáno číslo, které udává počet vláken, jež tento ukazatel ještě používají.

## Modifikace WRRM mapy

- *Procedura Update* provádí podmíněnou dealokaci, tj. dealokuje objekt odkazovaný ukazatelem, pouze pokud žádné jiné vlákno ukazatel nepoužívá.
- *Procedura Lookup* postupuje tak, že zvýší čítač spojený s ukazatelem, přistoupí ke struktuře skrze tento ukazatel, sníží čítač po ukončení práce se strukturou a podmíněně dealokuje strukturu.

## Čítač asociovaný s ukazatelem MAP<K,V>\*

- `template <class K, class V>`  
`class WRRMMap {`  
    `typedef std::pair<Map<K,V>*,unsigned> Data;`  
    `Data* pData_;`  
    `...`  
}
- CAS instrukce nad ukazatelem `pData_`
- Podmíněná dealokace:  
    `if (pData_>second==0) delete (pData_>first);`

## Problém v proceduře Lookup

- Vlákno A **načte strukturu** Data (přes `*pData_`) a je přerušeno.
- Vlákno B vloží klíč, sníží čítač a dealokuje `*pOld->first`.
- Vlákno A **zvýší čítač**, ale přistoupí k neplatnému ukazateli.

## Problém předchozí verze

- Akce uchopení ukazatele a zvýšení odpovídajícího čítače nebyly atomické.

## Řešení

- Pomocí jedné instrukce CAS je třeba přepnout ukazatel a korektně manipulovat s čítačem.
- Teoreticky je možné implementovat CAS pracující s více strukturami zároveň, ovšem ztrácí se efektivita, pokud neexistuje odpovídající HW podpora.
- Moderní procesory mají podporu pro instrukci CAS pracující se dvěma po sobě uloženými slovy procesoru (CAS2).

## Myšlenka

- **template** <class K, class V>  
**class** WRRMMap {  
    **typedef** std::pair<Map<K,V>\*,**unsigned**> Data;  
    Data data\_;  
    ...  
}
- Struktura Data je tvořena dvěma slovy: **ukazatel** a **čítač**
- Ukazatel a čítač jsou uloženy v paměti vedle sebe.
- Strukturu je možné modifikovat pomocí instrukce CAS2.

```
V Lookup (const K& k) {
 Data old;
 Data fresh;
 do {
 old = data__;
 fresh = old;
 ++fresh.second;
 } while (!CAS2(&data__, old, fresh));
 V temp = ((*fresh.first)[k]
 do {
 old = data__;
 fresh = old;
 --fresh.second;
 } while (!CAS2(&data__, old, fresh));
 if (fresh.second == 0) { delete fresh.first; }
 return temp;
}
```

## Otázka

- Umíme atomicky realizovat počítání odkazů, je tedy navrhované řešení korektní?

## Problém

- Zvýšení a snížení čítače procedurou Lookup je ve zcela nezávislých blocích. Pokud se mezi provedením těchto bloků realizuje nějaká procedura Update, tak přičtení a odečtení jedničky k čítači proběhne nad jinými ukazateli.
- Riziko předčasné dealokace.
- **Ztráta ukazatelů na staré kopie – memory leak.**

## Řešení

- Čítač spojený s ukazatelem použijeme jako stráž.
- Procedura Update bude provádět změny struktury jen tehdy, pokud žádné jiné vlákno ke struktuře nepřistupuje.

## Odkládání provedení modifikace v proceduře Update

- Atomické nahrazení ukazatele se děje v okamžiku, kdy jsou všechna ostatní vlákna mimo proceduru Lookup.
- Časové intervaly, po které se jednotlivá vlákna nacházejí v proceduře Lookup čtenářům se však mohou překrývat.
- Čítač po celou dobu existence jiného vlákna v proceduře Lookup neklesá na minimální hodnotu a procedura Update tzv. hladoví (starve).

## Optimalizace procedury Update

- Při opakovaných neúspěšných instrukce CAS dochází k opakovanému kopírování původní struktury a následnému mazání vytvořené kopie.
- Neefektivní opakované kopírování lze odstranit pomocí pomocného ukazatele `last`.



# WRRM Map s využitím CAS2 – Update

```
void Update (const K& k, const V& v) {
 Data old;
 Data fresh;
 old.second = 1;
 fresh.first = 0;
 fresh.second = 1;
 Map<K,V>* last = 0;
 do {
 old.first = data_.first;
 if (last != old.first) {
 delete fresh.first;
 fresh.first = new Map<K,V>(old.first);
 fresh.first->insert(make_pair(k,v));
 last = old.first;
 }
 } while (!CAS2(&data_, old, fresh));
 delete old.first;
}
```

## Lookup

- Není wait-free, inkrementace a dekrementace čítače interferuje s procedurou Update.
- Volání procedur Update je málo – nevadí.

## Update

- Není wait-free, interferuje s procedurou Lookup.
- Volání procedur Lookup je mnoho – problém.

## Čeho jsme dosáhli

- WRRM BNTM Mapa
- **Write Rarely Read Many, But Not Too Many**

# Hazardní ukazatele

aneb tak trochu

“Lock-Free Garbage Collector”

## Motivace

- Dealokace datových struktur v kontextu lock-free programování je obtížná.
- Ukazatel na datový objekt nerozliší, zda je možné, objekt uvolnit z paměti, či nikoliv.
- Čítače použití ukazatelů nejsou dobré řešení.

## Princip řešení pomocí hazardních ukazatelů

- Vlákna vystavují ostatním vláknům seznam ukazatelů, které momentálně používají – tzv. **hazardní ukazatele**.
- Bezpečně lze dealokovat pouze objekty, které nejsou odkazovány hazardními ukazateli.

## Původní problém lock-free implementace WRRM Mapy

- Procedura *update* vytvoří kopii mapy, modifikuje ji, nahradí touto kopií aktuální mapu a starou mapu dealokuje.
- Dealokace staré mapy může interferovat s probíhající procedurou *Lookup* jiného vlákna.

## Řešení

- WRRM Mapa udržuje seznam ukazatelů, které jsou momentálně používány nějakým vláknem v proceduře *Lookup*.
- *Lookup* – vkládá a odebírá ukazatel do seznamu.
- *Update* – uchovává (per-thread) již neplatné ukazatele a příležitostně je prochází a dealokuje ty, které nejsou hazardní.
- Hazardní ukazatele jsou uchovávány ve sdílené datové struktuře  $\implies$  je třeba ošetřit paralelní přístupy.

## Spojový seznam

- Záznam seznamu obsahuje ukazatel a příznak validity:

```
int active_
void* pHazard_
```

## Metoda Acquire()

- Vytvoří nebo znovu použije neplatný objekt seznamu a vrátí volajícímu ukazatel na tento objekt.
- Použije se pro zveřejnění používaného ukazatele.

## Metoda Release()

- Použije se pro zneplatnění objektu, tj. oznámení, že ukazatel uložený v tomto objektu již není dále používán.

```
class HPRecType {
 HPRecType * pNext_;
 int active_;
 static HPRecType* pHead_;
 static int listLen_;
public:
 void * pHazard_;
 static HPRecType* Head() { return pHead_; }
 static HPRecType* Acquire() {
 ...
 }
 static void Release(HPRecType* p) {
 p->pHazard_ = 0; // Order matters, pHazard_=0 first
 p->active_ = 0;
 }
}
```

# Objekt pro používané ukazatele

```
static HPRecType* Acquire() {
 HPRecType *p = pHead_;
 for(; p; p=p->pNext_) { // Try to reuse
 if (p->active_ or !CAS(&p->active_,0,1)) continue;
 return p;
 }
 int oldLen; // Increment the length
 do {
 oldLen = listLen_;
 } while (!CAS(&listLen_,oldLen, oldLen+1));
 HPRecType *p = new HPRecType; // Allocate new slot
 p->active_ = 1;
 p->pHazard_ = 0;
 do { // Push it to the front
 old = pHead_;
 p->pNext_ = old;
 } while (!CAS(&pHead_, old , p));
 return p;
}
```



## Princip

- Ukazatele na instance určené k dealokaci jsou schromažďovány do seznamu odložených ukazatelů.
- Každé vlákno má svůj vlastní seznam.

## Retire()

- Nahrazuje funkci **delete**, odkládá ukazatel do seznamu.
- Je-li seznam příliš dlouhý, volá proceduru Scan, která ze seznamu odstraní nadále nepoužívané ukazatele.
- Příliš dlouhý – dáno parametrem  $R$ .

## Scan()

- Vytvoří kopii seznamu používaných ukazatelů a seznam setřídí.
- Pro každý odložený ukazatel hledá binárním půlením v seznamu používaných ukazatelů, zda je ještě používán.
- Nadále nepoužívané objekty dealokuje.

# Seznam odložených ukazatelů určených k dealokaci

```
class HPRecType {
 ...
};

__per_thread__ vector<Map<K,V>*> rlist;

template <class K, class V>
class WRRMMap {
 ...
private:
 static void Retire(Map<K,V>* pOld) {
 rlist.push_back(pOld);
 if (rlist.size() >= R)
 Scan(HPRecType::Head());
 }
 void Scan(HPRecType* head) {
 ...
 }
};
```

# Dealokace odložených ukazatelů

```
void Scan(HPRecType* head) {
 vector<void*> hp; // collect non-null hazard pointers
 while (head) {
 void* p = head->pHazard_;
 if (p) hp.push_back(p);
 head = head->pNext_;
 }
 sort(hp.begin(),hp.end(), less<void*>());
 vector<Map<K,V>*>::iterator i = rlist.begin();
 while (i!=rlist.end()) { // for every retired pointer
 if (!binary_search(hp.begin(),hp.end(),*i) { // if not used anymore
 delete *i; // delete it
 if (&*i != &rlist.back()) //and dequeue it
 *i = rlist.back(); // replace it with the last one
 rlist.pop_back(); // dequeue the last one
 } else {
 ++i;
 }
 }
};
```

```
void Update(const K& k, const V& v) {
 Map <K,V>* pNew=0
 do {
 Map <K,V>* pOld = pMap_;
 delete pNew; //if (pNew==0) nothing happens
 pNew = new Map<K,V>(*pOld);
 (*pNew)[k] = v;
 } while (!CAS(&pMap_, pOld, pNew));
 Retire(pOld);
}
```

```
V Lookup(constK& k) {
 HPRecType *pRec = HPRecType::Acquire();
 Map<K,V> *ptr;
 do {
 ptr = pMap_;
 pRec -> pHazard_ = ptr;
 } while (pMap_ != ptr); // is ptr still valid? if so, go on
 V result (*ptr) [k];
 HPRecType::Release(pRec);
 return result;
}
```

## WRRM Mapa a Hazardní ukazatele

- Volání procedury Update interferuje s procedurou Lookup.
- Procedura Lookup není wait-free.
- Předpokládáme přístup v režimu **Write Rarely**, takže to nevadí.

## Hazardní ukazatele

- Možné řešení problému deterministické dealokace v případě, že systém nepodporuje garbage collection.
- Obecně je možné udržovat vícero hazardních ukazatelů na jedno vlákno.
- Amortizovaná složitost je konstantní.

## Návrh Lock-Free datových struktur

- Je možné navrhnout lock-free datové struktury.
- Zajímavá algoritmika.
- Obtížné, pokud chceme deterministické uvolňování paměti.
- Vhodné pro prostředí s Garbage Collectorem (JAVA).

## Další programátorská rozhraní



## MCAS

- Rozšíření standardní instrukce CAS pro použití s libovolně velkou datovou strukturou.

## Transakční paměť

- Paměť je modifikována v jednotlivých transakcích.
- Transakce seskupuje mnoho čtení a zápisů do paměti – je schopna obsáhnout komplexní modifikaci datových struktur.
- Základním manipulovatelným objektem je slovo procesoru, tj. obsah jedné paměťové buňky.
- Příklad: přesun prvku v dynamicky zřetěženém seznamu.

## Load-Link/Store-Conditional

- Dvojice instrukcí, která dohromady realizuje CAS.
- LL načte hodnotu a SC ji přepíše, pokud nebyla modifikována. Za modifikaci se považuje i přepsání na tutéž hodnotu.
- LL/SC stejná síla jako CAS, avšak nemá ABA problém.
- HW podpora: Alpha, PowerPC, MIPS, ARM

## Problémy

- Změna kontextu se v praxi považuje za modifikaci místa.
- Teoreticky není možné realizovat wait-free proceduru.
- Obtížné ladění.