Obsah

Články

Reference

Licence článků

Souběh

Souběh (anglicky **race condition**) je chyba v systému nebo procesu, ve kterém jsou výsledky nepředvídatelné při nesprávném pořadí nebo načasování jeho jednotlivých operací. Souběh může nastat v elektronických systémech (zvláště u logických členů) a v počítačových programech (zejména ve víceúlohových a víceprocesorových systémech).

Souběh v počítačových programech

Souběh je v počítačových programech způsoben chybným současným zpracováním sdílených dat. Pokud by byla data zpracována postupně, k chybě by nedošlo. Problémem je, že ke změně dat dojde ve chvíli, kdy se se stejnými daty již pracuje jiná úloha. Souběh způsobuje změna kontextu ve víceúlohových systémech nebo současné zpracování úloh na víceprocesorových systémech.

Kritická oblast je označení dat, která jsou souběhem ohrožena. **Kritická sekce** je nejmenší část programu, která pracuje s daty v kritické oblasti.

Souběh v číslicové technice

Souběh může příležitostně nastat i u logických členů, protože různé části obvodu mohou mít různou délku odezvy a je tak možné se dostat do kolizního stavu (viz Grayův kód).

Předcházení souběhu

Atomické instrukce

Atomické instrukce procesoru jsou vždy vykonány bez přerušení. Lze je využít ve víceúlohových systémech s jedním procesorem. Nepomohou ve víceprocesorových systémech, kde dochází ke skutečnému paralelnímu zpracování dat.

Zakázání přerušení

Zakázání přerušení umožňuje vytvořit atomickou operaci z více strojových instrukcí. Na začátku operace je přerušení zakázáno, po dokončení operace je opět povoleno. Sled instrukcí tak nemůže být nijak přerušen. Protože je však zakázání přerušení velmi nebezpečné pro stabilitu systému, může tuto metodu zpravidla používat pouze operační systém. Zakázání přerušení není možné použít ve víceprocesorových systémech.

Instrukce TSL

Instrukce TSL (anglicky Test and Set Lock) umožňuje vytvořit zámek, který zajistí pouze jedinému procesu vstup do kritické sekce.

> *Podrobnější informace naleznete v článku TSL.*

Semafor

Semafor je zobecněním instrukce TSL.

> *Související informace naleznete také v článku Semafor (synchronizace).*

Příklady souběhu

V životě

Sekretářka přidává všem zaměstnancům v účetním programu sto korun. U posledního zaměstnance nestihne uložit změny a jde uvařit šéfovi kafe. Její nadřízený mezitím otevře stejný program s tím, že onomu poslednímu zaměstnanci přidá na prémiích tisíc korun a změny uloží. Změny se uloží na serveru, ale ne u sekretářky, která se mezitím vrátila a změny uložila. Z toho plyne, že zaměstnanec přišel o tisíc korun a připsala se mu jen ta stokoruna.

V programování

Současně provedený výběr a vklad peněz v bance:

```
1. proces 2. proces

Pom1 := Konto;
Pom1 := Pom1 + 1000;
----------->
Pom2 := Konto;
Pom2 := Pom2 - 1000;
Konto := Pom2;
<----------
Konto := Pom1;
```

Literatura

- KOLÁŘ, Petr. *Operační systémy* [online]. Liberec: 2005-02-01, [cit. 2008-08-30]. Dostupné online. [1]

Reference

[1] http://www.nti.tul.cz/~kolar/os/

Kritická sekce

Kritická sekce (též **kritický kód**, anglicky **critical section**) je v informatice nejmenší část zdrojového kódu, kde dochází k přístupu ke sdílenému prostředku (např. sdílená data, která označujeme jako *kritická oblast*), ke kterému nemohou současně přistupovat dva nebo více procesů či vláken. Programy, které usilují o vstup do kritické sekce, musí použít nějaké synchronizační primitivum, které má za úkol zajistit do kritické sekce exkluzivní přístup a zároveň konečnou dobu čekání na povolení ke vstupu.

Popis činnosti

Pokud je vykonáván kód kritické sekce, musí ostatní vlákna nebo procesy vyčkat. Proto je nutné použít nějaké synchronizační primitivum, které je vyvoláno při vstupu a následně též při výstupu a kritické sekce, například semafor nebo mutex.

Související informace naleznete také v článku Synchronizační primitivum.

Problém kritické sekce

Při řízení přístupu do kritické sekce musí být dodrženy tři podmínky:

1. výhradní přístup – vstup do kritické sekce je povolen nejvýše jednomu procesu
2. vývoj – rozhodování o vstupu je pouze na procesech, které o něj usilují
3. omezené čekání – rozhodnutí o vstupu nesmí být pro některého čekajícího odkládáno do nekonečna

Při usilování o vstup do kritické sekce mohou procesy použít aktivní čekání (neustále se pokoušejí vstoupit do kritické sekce). Pro odstranění aktivního čekání jsou některá synchronizační primitiva rozšířena o frontu čekajících procesů (typicky např. semafory).

Reference

Související články

- Synchronizační primitivum
- Semafor (synchronizace)

Vzájemné vyloučení

Vzájemné vyloučení (anglicky **mutual exclusion**, nebo zkráceně **mutex**) je algoritmus používaný v programování jako synchronizační prostředek. Zabraňuje tomu, aby byly současně vykonávány dva (nebo více) kritické kódy nad stejným sdíleným prostředkem, jako například globální proměnné. Kritický kód je část kódu, ve které proces nebo vlákno přístupuje k veřejným prostředkům. Kritický kód není mechanismus ani algoritmus pro vzájemné vyloučení. Pokud jeden proces vstoupil do kritického kódu a nedokončil poslední instrukci (nevystoupil z kritického kódu), nemůže nad tímto prostředkem žádný jiný proces vstoupit do kritického kódu.

Stavy mutexu

Mutex může nabývat dvou stavů: volný a vlastněný. K tomu udržuje dva druhy informace. Prvním je identifikátor procesu, který mutex drží. Druhým je počet uzamknutí. Počet uzamknutí je vyjádřen číslem, které aktualizuje operace lock (získání mutexu) nebo unlock (uvolnění mutexu). Cílem je zabránění více procesům držet daný mutex. Při vícenásobném uzamknutí musí být i stejný počet odemknutí.

Získání mutexu

Případy, jak proces získá mutex:

- mutex je volný - proces se stává držitelem mutexu a počet uzamknutí je 1
- mutex je vlastněn aktuálním procesem - počet uzamknutí se zvýší o 1
- mutex je vlastněn neaktuálním procesem - proces se zablokuje a čeká na uvolnění mutexu

Uvolnění mutexu

Případy uvolnění mutexu procesem:

- mutex je volný - nedefinovaný stav, nemůže dojít k vyššímu počtu odemčení než bylo zamčení
- mutex je vlastněn aktuálním procesem - počet uzamknutí se sníží o 1, pokud je potom nulový, je uvolněn čekajícím procesům
- mutex je vlastněn neaktuálním procesem - nedefinovaný stav, neaktuální proces nemůže uvolňovat mutex, ten tak nebude uvolněn

Nevýhody

Nesprávné použití mutexu může vést ke zpomalení procesů, kdy většinu času jsou zablokovány ze vzájemného čekání, nebo v horším případě k uváznutí dvou a více procesů. Jako řešení potom musí být alespoň jeden proces ukončen.

Související články

- Synchronizační primitivum
- Kritická sekce
- Monitor (synchronizace)

Spinlock

Spinlock je v operačních systémech druh zámku, na nějž je třeba aktivně čekat – čekající proces tedy při čekání na spinlock spotřebovává systémové prostředky.

Spinlocky se zpravidla používají pouze v operačním systému, aplikacím jsou poskytována složitější synchronizační primitiva, které čekající aplikace uspí a zařadí do fronty, takže v době, kdy jsou zablokovány, může běžet něco jiného. Na druhou stranu, tyto složitější struktury vyžadují ochranu svých dat proti vícenásobnému přístupu, a k tomu lze použít právě jednodušší a rychlejší spinlocky.

Příklad implementace spinlocku v assembleru architektury x86:

```
zacatek:
mov eax, 1                ; přesuneme jedničku do registru
xchg eax, [ $zamek ]      ; jednou instrukcí atomicky prohodíme obsah registru
                          ; s proměnnou držící zámek. nyní je v proměnné určitě jednička
test eax, eax             ; pokud je v registru nula, zámek byl před prohozením
                          ; odemčený, tudíž jsme jej získali a můžeme pokračovat
jnz zacatek               ; ... jinak to zkusíme znovu od začátku

; ( kritická sekce )

mov eax, 0                ; konec, vrátíme do proměnné nulu a tím zámek odemkneme
xchg eax, [ $zamek ]      ; na toto by měla fungovat i prostá instrukce mov,
                          ; ale na některých procesorech se pokazí
```

Související články

- Atomicita
- Semafor (synchronizace)
- Monitor (synchronizace)
- Kritická sekce

TSL

TSL (anglicky **Test-and-Set Lock**) je v informatice jednoduchá atomická operace, která slouží k vytváření synchronizačních zámků. Operace TSL funguje tak, že nastaví hodnotu typu boolean na pravdu (*true*) a vrátí její předchozí hodnotu. Zajistí tak, že ze soutěžících procesů je pouze jednomu dovoleno vstoupit do kritické sekce a při nežádoucím souběhu nedojde k poškození dat v *kritické oblasti* současným zápisem. Další proces se dostane do kritické sekce až poté, co z ní předchozí vystoupí a hodnotu zámku nastaví na nepravdu (*false*), přičemž čekající proces provádí pokusy o vstup ve smyčce (tzv. aktivní čekání).

Charakteristika

TSL je velmi jednoduchá metoda vytváření zámku, která může být snadno implementována jako instrukce procesoru. Většina současných procesorů nějakou formu instrukce TSL obsahuje. TSL může být implementováno i softwarově, avšak pak je nutné zajistit její atomicitu.

Negativem metody využívající TSL je aktivní čekání (*busy waiting*, též *spinlock*), takže se hodí jen pro ochranu krátkých úseků kódu kritické sekce. Může však být základem pro řešení využívajících semaforů a fronty, která aktivní čekání odstraňuje. Operace TSL může být strojová instrukce v procesoru, může být též implementována softwarově nebo poskytována jiným obvodem (DPRAM).

Maurice Herlihy v roce 1991 dokázal, že TSL má konečnou dobu čekání při řešení souběžného přístupu na rozdíl od metody compare-and-swap (*porovnej a vyměň*), avšak ne pro více, než dva procesy.

Použití

Instrukce TSL je používána před vstupem do kritické sekce. Instrukce při svém vyvolání nastaví proměnnou Lock na pravdu (zamčeno) a vrátí předchozí stav této proměnné. Proto ji můžeme umístit do čekací smyčky (tzv. *aktivní čekání*), která bude řešit vstup do kritické sekce. Po výstupu z kritické sekce je proměnná Lock nastavena na nepravdu (odemčeno).

```
while TestAndSet(Lock) do { nothing } ;
   ...kód kritické sekce...
Lock:=false;
```

Hardwarová implementace TSL

Instrukce TSL implementované pomocí DPRAM (anglicky *Dual Port RAM*) může pracovat mnoha způsoby. Zde jsou popsány dvě varianty, které popisují DPRAM se dvěma porty, které dovolují dvěma odděleným elektronickým komponentám (například dva procesory) přístup do celé paměti v DPRAM.

Varianta 1

Pokud procesor 1 zpracovává TSL instrukci, DPRAM si o tomto nejdříve udělá "vnitřní poznámku" uložením adresy umístění v paměti na speciální místo. Pokud v tomto okamžiku procesor 2 také spustí zpracování instrukce TSL pro stejné umístění v paměti, DPRAM nejdříve zkontroluje "vnitřní poznámku" tohoto místa paměti, rozpoznává hrozící kolizi a oznámí procesoru 2, že musí počkat a a zkusit to znovu. Toto je implementace techniky nazývané „busy waiting“ nebo „spinlock“. Protože se toto odehrává na hardwarové úrovni, čekání procesoru 2 je velmi krátké.

Ať se procesor 2 pokoušel nebo nepokoušel přistupovat na adresu, DPRAM umožní procesoru 1 provedení svou operaci TSL. Pokud TSL procesoru 1 uspěje, DPRAM nastaví adresu paměti na hodnotu danou procesorem 1. Potom DPRAM vymaže jeho "vnitřní poznámku" kterou procesor 1 svou TSL instrukcí zapsal. Následně procesor 2 neúspěšně dokončí svou TSL instrukci.

Varianta 2

Procesor 1 zpracovává TSL instrukci na adrese A. DPRAM neukládá ihned požadovaná data, místo toho přesune původní data do speciálního registru a změní obsah paměti A na hodnotu, která má význam příznaku TSL instrukce na této adrese. Pokud v tento okamžik procesor 2 spustí zpracování TSL instrukce na stejné adrese, pak DPRAM detekuje hodnotu s významem již probíhající TSL instrukce a podobně jako ve variantě jedna oznámí procesoru 2, že musí čekat.

Ať se procesor 2 pokoušel nebo nepokoušel přistupovat na adresu, DPRAM umožní procesoru 1 provedení svou operaci TSL. Pokud TSL procesoru 1 uspěje, DPRAM nastaví adresu paměti na hodnotu danou procesorem 1. Pokud TSL neuspěje, DPRAM vrací zpět do paměti hodnotu uloženou ve speciálním registru. Obě operace odstraní z paměti hodnotu s významem probíhající instrukce TSL. Pokud v tomto okamžiku spustí TSL, pak uspěje.

Softwarové realizace TSL

Mnoho procesorů má ve svém instrukčním souboru TSL instrukce. Procesory, které tyto instrukce nemají, používají atomické (nepřerušitelné) výměny nebo jiné atomické instrukce pro načtení, modifikaci zpětné uložení.

TLS instrukce s využitím logické proměnné se chová podle níže uvedeného kódu funkce. Velmi důležitou vlastností této funkce je, že žádný proces ji nemůže přerušit během jejího provádění a proto se chová tak, jak se očekává. Tento kód je určen jen pro pochopení funkce TSL, atomista (nepřerušitelnost) vyžaduje určitou hardwarovou podporu a z toho důvodu nemůže být takto jednoduše implementována.

Pozn. v tomto příkladu je lock předán referencí, ale přiřazení do initial vytváří novou hodnotu.

```
function TestAndSet(boolean lock) {
   boolean initial = lock;
   lock = true;
   return initial;
}
```

Výše uvedený kód není atomický ve smyslu TSL instrukce. Také se liší od výše uvedeného popisu DPRAM a hardwarové realizace TSL, kdy v kódu není provedeno nastavení paměti na základě testu oproti DPRAM hw realizaci TSL, kdy testovanou hodnotu a v případě úspěchu vkládanou hodnotu definuje procesor.

V dalším příkladě může být hodnota nastavena pouze na 1. Ale pokud je 0 a 1 považována za platný obsah definovaného místa adresy a test je prováděn na nenulovou hodnotu, pak je toto rovnocenné případu popisovaném pro hw implementaci v DPRAM (resp. případu DPRAM omezené těmito podmínkami).

Z tohoto pohledu, lze toto korektně nazvat TSL v plném smyslu tohoto významu. Důležitým bodem k povšimnutí je účel a princip TSL, kdy je hodnota je testována a nastavena v jedné atomické operaci tak že žádné jiný programové vlákno nesmí způsobit změnu cílové paměti po jejím testu, ale jen před jejím nastavením, což by mohlo porušit logický požadavek že paměť bude nastavena jen pokud má určitou hodnotu.

Příklad realizace v programovacím jazyce C:

```
#define LOCKED 1
int TestAndSet(int* lockPtr) {
    int oldValue;
    oldValue = SwapAtomic(lockPtr, LOCKED);
    return oldValue == LOCKED;
}
```

Kde SwapAtomic atomicky nejdříve čte aktuální hodnotu přes ukazatel lockPtr a pak uloží 1 do umístění (pozn. překladatele: prostě prohodí *lockPtr a s konstantou 1, některé procesory mají pro toto strojovou instrukci).

Kód také ukazuje, že TSL je opravdu složena ze dvou operací: atomická výměna a test. Jen výměna musí být atomická. (Je to pravda, protože podržení hodnoty pro porovnání již nezmění výsledek testu, jakmile je hodnota pro test získána).

Realizace vzájemných vyloučení s TSL

Jedna možnost realizace vzájemného vyloučení používajících TSL je následující:

```
boolean lock = false
function Critical(){
    while TestAndSet(lock)
    skip //přeskočit dokud je zámek vyžadován
    critical section //pouze jeden proces může být v této sekci
    lock = false //uvolnit zámek, když proces skončí s kritickou sekcí
}
```

V pseudoprogramovacím jazyce C by to mohlo být:

```
volatile int lock = 0;
void Critical() {
    while (TestAndSet(&lock) == 1);
    critical section //pouze jeden proces může být v této sekci
    lock = 0 //uvolnit zámek, když proces skončí s kritickou sekcí
}
```

Všiměte si modifikátoru volatile. Pokud by nebyl použit, pak kompilátor optimalizoval přístup k proměnné použitím dočasné lokální proměnné, vygenerovaný kód by pak byl nefunkční.

Naopak přítomnost volatile nezajišťuje, že čtení a zápis jsou prováděny přímo v paměti. Některé překladače používají tzv. paměťové bariéry k zajištění přímého provádění čtení a zápisu paměti. Protože je v tomto bodě C/C++ poněkud vágně definováno, ne všechny kompilátory to tak dělají. Proto nahlédněte do dokumentace ke svému kompilátoru.

Jiná varianta implementace vzájemného vyloučení je známa jako „Test and Test-and-set“, je daleko efektivnější na víceprocesorových počítačích. Používá stejné Test and Set instrukce, jako výše popsaná metoda, ale vylepšuje koherenci vyrovnávacích pamětí.

Implementace semaforů pomoci TSL

Test and set je možné použít pro implementaci tzv. semaforů. V jednoprocesorových systémech není tato technika potřeba (kromě případů, kdy více procesů přistupuje ke stejným datům), při implementaci semaforu je postačující zakázat přerušení před přístupem k semaforu. Avšak u víceprocesorových strojů je nevhodné, ne-li nemožné zakázat přerušení na všech procesorech ve stejný čas. Bez zakázaného přerušení tedy může dva nebo více procesorů přistupovat k paměti semaforů ve stejný čas. V tomto případě může být použito test-and-set instrukcí.

Reference

Související články

- Souběh
- Kritická sekce

Semafor (synchronizace)

Semafor je v informatice široce používané synchronizační primitivum, které obsahuje celočíselný čítač. Semafor se využívá zejména jako ochrana proti souběhu tím, že chrání přístup do kritické sekce, k čemuž používá dvojici operací V (*up*) a P (*down*). Je tak zobecněním instrukce TSL, která používá proměnnou typu boolean. Semafory poprvé popsal holandský informatik Edsger Dijkstra v roce 1965.

Implementace

Implementace semaforu je založena na atomických operacích V (*verhogen*, též označováno jako *up*) a P (*proberen*, též označováno jako *down*). Operace *down* otestuje stav čítače a v případě že je nulový, zahájí čekání. Je-li nenulový, je čítač snížen o jedničku a vstup do kritické sekce je povolen. Při výstupu z kritické sekce je vyvolána operace *up*, která odblokuje vstup do kritické sekce pro další (čekající) proces. Čítač je možné si představit jako omezení počtu procesů, které mohou zároveň vstoupit do kritické sekce nebo například jako počitadlo volných prostředků. Tato implementace neodstraňuje problém aktivního čekání.

Odstranění aktivního čekání

V případě, že je při vyvolání operace *down* čítač nulový, je nutné volající proces zablokovat. Čekání je implementováno jako nekonečná smyčka (tzv. aktivní čekání), která může být přerušena pouze vnějším zásahem jiného procesu do počitadla pomocí volání operace *up*. Neustálé testování stavu proměnné je možné nahradit pomocí fronty čekajících procesů. Proces je místo aktivního čekání (tj. neustálého kontrolování stavu proměnné) zařazen do fronty, ve které je uspán. Funkce *up* je rozšířena o průchod touto frontou, kdy je kromě zvýšení počitadla aktivován pouze proces, který je ve frontě první. Tento proces sníží počitadlo a vstoupí do kritické sekce. Ostatní procesy dále čekají v uspaném stavu.

Příklad implementace v pseudokódu:

```
P(Semaphore s)
{
  čekej dokud není s > 0 pak s = s-1; /* musí být atomické, jakmile je zjištěno, že s > 0 */
}

V(Semaphore s)
{
  s = s+1;   /* musí být atomické */
}

Init(Semaphore s, Integer v)
{
  s = v;
}
```

Odkazy

Reference

Související články

- Monitor (synchronizace)

Monitor (synchronizace)

Monitor je synchronizační primitivum, které se používá pro řízení přístupu ke sdíleným prostředkům. Jeho zvláštností je, že jde o speciální konstrukci programovacího jazyka (musí ho tedy implementovat překladač), typicky implementovanou pomocí jiného synchronizačního primitiva. Výhodou monitoru oproti jiným primitivům je jeho vysokoúrovňovost – snadněji se používá a je bezpečnější. Při jeho použití je méně pravděpodobné, že programátor udělá chybu.

Monitor se skládá z dat, ke kterým je potřeba řídit přístup, a množiny funkcí, které nad těmito daty operují.

Vzájemné vyloučení

Monitor se podobá třídě z OOP. Odlišností je to, že překladač doplní monitor o zámek, díky němuž se dosáhne vzájemné vyloučení – v jednu chvíli může být uvnitř monitoru jen jeden proces.

Když chce proces vstoupit do monitoru (tj. zavolat jeho funkci), musí nejdříve získat zámek. Pokud zámek v tu chvíli drží někdo jiný, tak se proces zablokuje a čeká, dokud se zámek neuvolní (tj. dokud jiný proces neopustí monitor nebo nezačne čekat na podmíněnou proměnnou).

Celý proces zamykání je pro programátora transparentní. V programu se funkce monitoru volají stejně jako ostatní funkce. Kód, který provádí zamykání a odemykání, vygeneruje překladač.

Monitor většinou splňuje dodatečné podmínky:

- data monitoru jsou přístupná jen z jeho funkcí,
- funkce monitoru nepoužívají data mimo monitor,
- každá funkce zajistí, že před uvolněním zámku jsou data v konzistentním stavu.

Pokud jsou tyto podmínky splněny, platí, že žádný proces nenajde data v nekonzistentním stavu, což je přesně důvod pro zavedení synchronizačního primitiva.

Jako jednoduchý příklad poslouží monitor pro bankovní účet:

```
monitor účet {
  int zůstatek := 0

  function vybrat(int částka) {
    if částka < 0 then error "Vybíraná částka nesmí být záporná"
    else if zůstatek < částka then error "Nedostatečný zůstatek"
    else zůstatek := zůstatek - částka
  }

  function vložit(int částka) {
    if částka < 0 then error "Vkládaná částka nesmí být záporná"
    else zůstatek := zůstatek + částka
  }
```

```
}
```

Pokud by bankovní účet nebyl v monitoru, mohlo by dojít k souběhu (race condition): Řekněme, že na účtu je 200 Kč a dva procesy chtějí najednou každý vybrat 150 Kč. Bez synchronizace může dojít k tomu, že ověření zůstatku proběhne u obou procesů dříve, než jsou peníze odečteny. Každý proces tak vidí na účtu 200 Kč a povolí odečtení částky. Po skončení obou operací je ale účet v nekonzistentním stavu – obsahuje –100 Kč.

Díky monitoru ale jeden proces získá zámek dřív a druhý proces zatím musí čekat. První proces ověří zůstatek, odečte 150 Kč a vyskočí z monitoru. Až poté se do monitoru dostane druhý proces, který ohlásí nedostatek zůstatku na účtu.

Podmíněné proměnné

Občas je potřeba, aby proces, který je právě v monitoru, počkal na nějakou událost. Monitor poskytuje tuto funkcionalitu pomocí tzv. *podmíněných proměnných*.

Když funkce monitoru potřebuje počkat na splnění podmínky, vyvolá operaci *wait* na podmíněné proměnné, která je s touto podmínkou svázána. Tato operace proces zablokuje, zámek držený tímto procesem je uvolněn a proces je odstraněn ze seznamu běžících procesů a čeká, dokud není podmínka splněna. Jiné procesy zatím mohou vstoupit do monitoru (zámek byl uvolněn). Pokud je jiným procesem podmínka splněna, může funkce monitoru „signalizovat“, tj. probudit čekající proces pomocí operace *notify*.

„Zajímavých událostí“ může být více, každá může být spojená s vlastní podmíněnou proměnnou. Operace *notify* budí jen ty procesy, které provedly *wait* na stejné proměnné.

Následující monitor používá podmíněné proměnné k implementaci komunikačního kanálu, který v jednom okamžiku obsahuje jen jedno číslo.

```
monitor kanál {
  int obsah
  boolean naplněn := false
  condition posláno
  condition přijato

  function poslat(int data) {
    while naplněn then wait(přijato)
    obsah := data
    naplněn := true
    notify(posláno)
  }

  function přijmout() {
    var int data

    while not naplněn then wait(posláno)
    data := obsah
    naplněn := false
    notify(přijato)
    return data
  }
}
```

V dřívějších implementacích signalizování způsobilo, že čekající proces se okamžitě rozběhl a získal zámek (signalizující proces se zablokoval, dokud se vzbuzený proces zámku zase nevzdal), čímž bylo zaručeno, že podmínka je stále ještě splněna. Implementace tohoto chování je komplikovaná a má velkou režii. Je také nekompatibilní s plánovači, které mohou proces kdykoliv přeplánovat. Z těchto důvodů existuje několik různých sémantik podmíněných proměnných, co se týče signalizování. Ve většině moderních implementací signalizace neblokuje signalizující proces, pouze způsobí, že proces čekající na podmíněnou proměnou bude čekat na získání zámku (prakticky je proces přeřazen z jedné fronty do druhé).

Tento přístup má dva vedlejší efekty. Signalizující proces nemusí před signalizací uvést monitor do konzistentního stavu, protože stále drží zámek. Na druhou stranu nelze zaručit, že důvod signalizace stále platí ve chvíli, kdy se probuzený proces rozběhne. Mezitím totiž mohl jiný proces podmínku opět zneplatnit. Čekání na podmínku tedy musí být nezbytně implementováno pomocí *dvojí kontroly* – podmínka se testuje před i po volání *wait* a pokud není splněna, opět se volá *wait*. To se řeší použitím smyčky namísto jednoduchého podmíněného příkazu.

Většina implementací také poskytuje operaci *notifyAll*, která probudí všechny procesy čekající na danou podmíněnou proměnnou.

Navzdory svému názvu nenabývají podmíněné proměnné hodnot *true* a *false*. Vlastně nemají žádnou přístupnou hodnotu. Bývají to fronty procesů, které čekají na splnění podmínky, ale tato fronta je zvenčí nepřístupná, jediné, co lze s podmíněnými proměnnými provést, je volat na nich operace *wait* a *notify*. Samotná podmínka (např. zda sdílená datová struktura je plná/prázdná) musí být uchovávána v běžné proměnné.

Podpora v programovacích jazycích

V některých objektově-orientovaných programovacích jazycích jsou monitory hlavním synchronizačním primitivem kvůli své vysokoúrovňovosti: objekty mohou být přímo svázány s monitory, každý objekt automaticky vlastní svůj unikátní monitor.

Java

V programovacím jazyku Java jsou monitory hlavním synchronizačním primitivem. Každý objekt má automaticky přiřazen svůj monitor. Funkce, které patří do monitoru, jsou označeny pomocí klíčového slova `synchronized`. Do monitoru libovolného objektu však lze obalit libovolný blok kódu pomocí konstrukce `synchronized(`*`objekt`*`) { ... }` (ve skutečnosti je označení celé metody tímto klíčovým slovem jen syntaktická zkratka pro tuto konstrukci použitou na aktuální objekt – `this`).

Java nepodporuje (alespoň ne přímo v jazyce) podmíněné proměnné. Operace *wait*, *notify* a *notifyAll* jsou implementovány jako metody třídy `Object`, která je společným předkem všech tříd. V podstatě tak každý objekt obsahuje právě jednu podmíněnou proměnnou, která udržuje seznam čekajících vláken.

Stejně jako zámek, který zajišťuje vzájemné vyloučení, je tato podmíněná proměnná pro programátora transparentní a nepřístupná.

Původně byly monitory jediným v Javě dostupným synchronizačním primitivem (díky ekvivalenci ale bylo možné ostatní primitiva simulovat). Ve verzi 1.5 byl do jazyka přidán balíček `java.util.concurrent`, který obsahuje i jiná synchronizační primitiva.

Příklad

Implementace příkladu s komunikačním kanálem uvedeného výše v programovacím jazyce Java:

```
class Kanál {
   private int data;
   private boolean naplněn = false;

   public synchronized int přijmout() {
       while (!naplněn) {
           try {
               wait();
           } catch (InterruptedException e) {}
       }
       naplněn = false;
       return data;
   }

   public synchronized void poslat(int hodnota) {
       data = hodnota;
       naplněn = true;
       notify();
   }
}
```

.NET

V prostředí .NET (a tedy např. programovací jazyk C#) se používá prakticky stejný princip: každý objekt má svůj monitor, obdobou klíčového slova `synchronized` je v C# klíčové slovo `lock`. Navíc však .NET poskytuje i nízkoúrovňovější přístup k monitorům pomocí třídy `Monitor` a jejích metod (např. `Enter` a `Exit`), které dovolují o něco flexibilnější (ale „nebezpečnější“) řízení vzájemného vyloučení. Operace *wait*, *notify* a *notifyAll* jsou v .NETu podporovány taktéž prostřednictvím metod třídy `Monitor` (konkrétně `Wait`, `Pulse`, `PulseAll`).

.NET již od počátku podporuje i další synchronizační primitiva a další nástroje pro synchronizaci a paralelní programování.

Historie

Jako první popsal a implementoval monitory dánsko-americký počítačový vědec Per Brinch Hansen, přičemž vycházel z myšlenek C. A. R. Hoara. Ten pak následně vyvinul teoretický rámec a dokázal jejich ekvivalenci se semafory.

Odkazy

Související články

- Semafor (synchronizace)

Externí odkazy

- *Monitors: An Operating System Structuring Concept* [1], C. A. R. Hoare, Communications of the ACM, Vol. 17, No. 10. October 1974, pp. 549-557

Reference

[1] http://www.acm.org/classics/feb96/

Zdroje článků a přispěvatelé

Souběh *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=10590740 *Přispěvatelé*: Japo, Jx, Martin Zidu, Milan Keršláger, Riha, Tchoř, Vrba, 9 anonymní úpravy

Kritická sekce *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11223636 *Přispěvatelé*: Draffix, Milan Keršláger, Miltim

Vzájemné vyloučení *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11239132 *Přispěvatelé*: BobM, Draffix, Milan Keršláger, Miltim, 4 anonymní úpravy

Spinlock *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=9853928 *Přispěvatelé*: Beren, BilboqCyborg, Che, Gpvos, Hkmaly, SZtorkie, 1 anonymní úpravy

TSL *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11230369 *Přispěvatelé*: Bezdek9, Elm, Hadonos, Jana Lánová, Jvs, Milan Keršláger, Zzaapp, 2 anonymní úpravy

Semafor (synchronizace) *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11228397 *Přispěvatelé*: Garyczek, Hkmaly, Matěj Suchánek, Milan Keršláger, Muck, Utar, Venca24, 4 anonymní úpravy

Monitor (synchronizace) *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=10332742 *Přispěvatelé*: Fjares, Hkmaly, Hobr, Jj14, Miltim, Miraceti, Mormegil, Stalker, Venca24, 7 anonymní úpravy

Licence