Синхронізація

Проблема синхронізації

Синхронізація в комп'ютерних системах — це координація роботи процесів таким чином, щоб послідовність їх операцій була передбачуваною. Як правило, синхронізація необхідна при спільному доступі до ресурсів, що розділяються. **Критична секція** — частина програми, в якій є звернення до спільно використовуваних даних. При знаходженні в критичній секції двох (або більше) процесів, виникає стан гонки/змагання за ресурси. **Стан гонки** — це стан системи, при якому результат виконання операцій залежить від того, в якій послідовності виконуватимуться окремі процеси в цій системі, але керувати цією послідовністю немає можливості. Іншими словами, це протилежність правильної синхронізації.

Для уникнення гонок необхідне виконання таких умов:

- **Взаємного виключення**: два процеси не повинні одночасно перебувати в одній критичній області
- **Прогресу**: процес, що знаходиться поза критичною областю, не може блокувати інші процеси
- **Обмеженого очікування**: неможлива ситуація, в якій процес вічно чекає попадання в критичну область
- Також в програмі, в загальному випадку, не повинно бути припущень про швидкість або кількості процесорів

Приклад умови гонок: i++

```
movl i, $eax  // i - мітка адреси змінної i в пам'яти
incl $eax
movl $eax, i
```

Оскільки процесор не може модифікувати значення в пам'яті безпосередньо, для цього йому потрібно завантажити значення в регістр, змінити його, а потім знову вивантажити в пам'ять. Якщо в процесі виконання цих трьох інструкцій процес буде перерваний, може виникнути непередбачуваний результат, тобто має місце умова гонки.

Класичні задачі синхронізації

Класичні задачі синхронізації — це модельні задачі, на яких досліджуються різні

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

ситуації, які можуть виникати в системах з розділюємим доступом та конкуренцією за спільні ресурси. До них відносяться задачі: Виробник-споживач, Читачі-письменники, Обідаючі філософи, Сплячий перукар, Курці сигарет, Проблема Санта-Клауса та ін.

Задача **Виробник-споживач** (також відома як задача обмеженого буфера): 2 процеси — виробник і споживач — працюють із загальним ресурсом (буфером), що має максимальний розмір N. Виробник записує в буфер дані послідовно в чарунки 0,1,2, ..., поки він не заповниться, а споживач читає дані з буфера у зворотному порядку, поки він не спорожніє. Запис і зчитування не можуть відбуватися одночасно.

Наївний розв'язок

```
int buf[N];
int count = 0;
void producer() {
    while (1) {
        int item = produce_item();
        while (count == N - 1)
            /* do nothing */ ;
        buf[count] = item;
        count++;
    }
}
void consumer() {
    while (1) {
        while (count == 0)
            /* do nothing */ ;
        int item = buf[count - 1];
        count--;
        consume_item(item);
    }
}
int main() {
    make_thread(&producer);
    make_thread(&consumer);
}
```

Проблема синхронізації в цьому варіанті: якщо виробник буде перерваний споживачем після того, як запише дані в буфер `buf[count] = item`, але до того, як збільшить лічильник, то споживач зчитає з буфера елемент перед тільки що записаним, тобто в буфері утворюється дірка. Після того як виробник таки збільшить лічильник, лічильник якраз буде вказувати на цю дірку. Симетрична

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

проблема є і у споживача.

Також у цієї задачі може бути багато модифікацій: наприклад, кількість виробників і споживачів може бути більше 1. У цьому випадку додаються нові проблеми синхронізації.

Ще одна проблема цього рішення — це безглузда витрата обчислювальних ресурсів: цикли `while (count == 0) / * do nothing * /;` - т.зв. **зайняте очікування** (busy loop, busy waiting) або ж **поллінг** (pooling) — це ситуація, коли процес не виконує ніякої корисної роботи, але займає процесор і не дає в цей час працювати на ньому іншим процесам. Таких ситуацій треба по можливості уникати.

Алгоритми програмної синхронізації

Програмний алгоритм синхронізації — це алгоритм взаємного виключення, який не заснований на використанні спеціальних команд процесора для заборони переривань, блокування шини пам'яті і т.д. У ньому використовуються тільки загальні змінні пам'яті та цикл для очікування входу в критичну секцію виконуваного коду. У більшості випадків такі алгоритми не ефективні, тому що використовують поллінг для перевірки умов синхронізації.

Приклад програмного адгоритма — алгоритм Петерсона:

```
int interested[2];
int turn;
void enter_section(int process_id) {
    int other = 1 - process_id;
    interested[process_id] = 1;
    turn = other;
    while (turn == other && interested[other])
    /* busy waiting */;
}
void leave_section(int process_id) {
    interested[process_id] = 0;
}
```

Див. також алгоритм Деккера, алгоритм пекарні Лемпорта та ін.

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Апаратні інструкції синхронізації

Апаратні інструкції синхронізації реалізують **атомарні** примітивні операції, на основі яких можна будувати механізми синхронізації більш високого рівня. Атомарність означає, що вся операція виконується як ціле і не може бути перерваною посередині. Атомарні примітивні операції для синхронізації, як правило, виконують разом 2 дії: запис значення і перевірку попереднього значення. Це дає можливість перевірити умову і відразу записати таке значення, яке гарантує, що умова більше не буде виконуватися.

Try-and-set lock (TSL)

Інструкції типу `try-and-set` записують в регістр значення з пам'яті, а в пам'ять — значення 1. Потім вони порівнюють значення в регістрі з 0. Якщо в пам'яті і був 0 (тобто доступ до критичної області був відкритий), то порівняння пройде успішно, і в той же час в пам'ять буде записаний 1, що гарантує, що наступного разу порівняння вже не буде успішним, тобто доступ закриється.

Реалізація критичної секції за допомогою `TSL`:

```
enter_region:
        TSL REGISTER, LOCK
        CMP REGISTER, 0
        JNE ENTER_REGION
        RET
leave_region:
        MOV LOCK, 0
        RET
```

Те ж саме на C:

```
void lock(int *lock) {
    while (!test_and_set(lock))
        /* busy waiting */;
}
```

Compare-and-swap (CAS)

Инструкції типу `compare-and-swap` записують у регістр нове значення і при цьому перевіряють, що старе значення в регістрі рівне значенню, запам'ятованому раніше.

В x86 називається `CMPXCHG`.

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Аналог на C:

```
int compare_and_swap(int* reg, int oldval, int newval) {
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

Проблема ABA (ABBA): CAS інструкції не можуть відстежити ситуацію, коли значення в регістрі було змінено на нове, а потім знову було повернуто до попереднього значення. У більшості випадків це не впливає на роботу алгоритму, а в тих випадках, коли впливає, необхідно використовувати інструкції з перевіркою на таку ситуацію, такі як `LL/SC`.

Інші аппаратні инструкції

- Подвійний CAS
- Fetch-and-add
- Load-link/store-conditional (LL/SC)

Системні механізми синхронізації

За допомогою апаратних інструкцій можна реалізувати більш високорівневі конструкції, які можуть обмежувати доступ в критичну область, а також сигналізувати про якісь події.

Найпростішим варіантом обмеження доступу в критичну область є **змінна-замок**: якщо її значення дорівнює 0, то доступ відкритий, а якщо 1 — то закритий. У неї є 2 атомарні операції:

- заблокувати (**lock**) — перевірити, що значення дорівнює 0, і встановлює його в 1 або ж чекати, поки воно не стане 0
- розблокувати (**unlock**), яка встановлює значення в 1

Також корисною може бути операція спробувати заблокувати (**trylock**), яка не чекає поки значення замку стане 0, а відразу повертає відповідь про неможливість заблокувати замок.

Спінлок

Спінлок — це замок, очікування при блокуванні якого реалізовано у вигляді зайнятого очікування, тобто потік "крутиться" в циклі, очікуючи розблокування

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

замка.

Реалізація на асемблері за допомогою CAS:

```
lock:  # 1 = locked, 0 = unlocked.
    dd 0
spin_lock:
    mov eax, 1
    loop:
        xchg eax, [lock]
        # Atomically swap the EAX register with
        # the lock variable.
        # This will always store 1 to the lock,
        # leaving previous value in the EAX register
        test eax, eax
        # Test EAX with itself. Among other things, this
        # sets the processor's Zero Flag if EAX is 0.
        # If EAX is 0, then the lock was unlocked and
        # we just locked it. Otherwise, EAX is 1
        # and we didn't acquire the lock.
        jnz loop
        # Jump back to the XCHG instruction if Zero Flag
        # is not set, the lock was locked,
        # and we need to spin.
        ret
spin_unlock:
    mov eax, 0
    xchg eax, [lock]
    # Atomically swap the EAX register with
    # the lock variable.
    ret
```

Використання спінлока доцільно тільки в тих областях коду, які не можуть викликати блокування, інакше весь час, відведений планувальником потоку, що очікує на спінлоці, буде витрачено на очікування і при цьому інші потоки не будуть працювати.

Семафори

Семафор — це примітив синхронізації, що дозволяє обмежити доступ до критичної секції тільки для N потоків. При цьому, як правило, семафор дозволяє реалізувати це без використання зайнятого очікування.

Концептуально семафор включає в себе лічильник і чергу очікування для

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

потоків. Інтерфейс семафора складається з двох основних операцій: опустити (**down**) і підняти (**up**). Операція опустити атомарно перевіряє, що лічильник більше 0 і зменшує його. Якщо лічильник дорівнює 0, потік блокується і ставитися в чергу очікування. Операція підняти збільшує лічильник і посилає чекаючим потокам сигнал прокинутися, після чого один з цих потоків зможе повторити операцію опустити.

Бінарний семафор — це семафор з N = 1.

Мьютекс (mutex)

Мьютекс — від словосполучення mutual exclusion, тобто взаємне виключення — це примітив синхронізації, що нагадує бінарний семафор з додатковою умовою: розблокувати його повинен той же потік, який і заблокував.

Реалізація мьютекса за допомогою примітиву CAS:

```
void acquire_mutex(int *mutex) {
    while (cas(mutex, 1, 0)) /* busy waiting */;
}
void release_mutex(int *mutex) {
    *mutex =  1;
}
```

Варто відзначити, що не всі системи надають гарантії того, що мьютекс буде розблоковано саме потоком, що його заблокував. У наведеному вище прикладі ця умова якраз не перевіряється.

Мьютекс з можливістю повторного входу (re-entrant mutex) — це мьютекс, який дозволяє потоку кілька разів блокувати його.

RW lock

RW lock — це особливий вид замку, який дозволяє розмежувати потоки, які виконують тільки читання даних, і які виконують їх модифікацію. Він має операції заблокувати на читання (**rdlock**), яка може одночасно виконуватися декількома потоками, і заблокувати на запис (**wrlock**), яка може виконуватися тільки 1 потоком. Також правильні реалізації RW-замку дозволяють уникнути проблеми інверсії пріоритетів (див. нижче).

Змінні умови і монітори

Монітор — це механізм синхронізації в об'єктно-орієнтованому програмуванні, при використанні якого об'єкт позначається як синхронізований і компілятор

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

додає до викликів всіх його методів (або тільки виділених синхронізованих методів) блокування за допомогою мьютекса. При цьому код, що використовує цей об'єкт, не повинен піклуватися про синхронізацію. У цьому сенсі монітор є більш високорівневої конструкцією, ніж семафори і мьютекси.

Змінна умови (condition variable) — примітив синхронізації, що дозволяє реалізувати очікування якоїсь події і оповіщення про неї. Над нею можна виконувати такі дії:

- очікувати (**wait**) повідомлення про якусь подію
- сигналізувати подію всім потокам, що очікують на даній змінній. Сигналізація може бути блокуючою (**signal**) — у цьому випадку управління переходить до чекаючого потоку, - і неблокуючою (**notify**) — у цьому випадку управління залишається у сигналізуючого потоку

Більшість моніторів підтримують усередині себе використання змінних умови. Це дозволяє декільком потоком заходити в монітор і передавати керування один одному через цю змінну.

Див. http://en.wikipedia.org/wiki/Monitor_(synchronization)

Інтерфейс синхронізації

POSIX Threads (Pthreads) — це частина стандарту POSIX, яка описує базові примітиви синхронізації, підтримувані в Unix системах. Ці примітиви включають семафор, мьютекс і змінні умови.

Futex (швидкий замок у просторі користувача) — це реалізація мьютекса в Unix-системах, яка оптимізована для мінімального використання функцій ядра ОС, за рахунок чого досягається більш швидка робота з ним. За допомогою фьютексів в Linux реалізовані семафори і змінні умови.

Над фьютексамі можна робити такі базові операції:

- очікувати - `wait (addr, val)` - перевіряє, що значення за адресою `addr` рівне `val`, і, якщо це так, то переводить потік в стан очікування, а інакше продовжує роботу (тобто входить в критичну область)
- розбудити - `wake (addr, n)` - відправляє n потокам, які очікують на фьютексе за адресою `addr`, повідомлення про необхідність пробудитися

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Проблеми синхронізації

Крім умов гонки і зайнятого очікування неправильна синхронізація може призвести до наступних проблем:

Тупик/взаємне блокування (deadlock) — ситуація, коли 2 або більше потоків очікують розблокування замків один від одного і не можуть просунутися. Найпростіший приклад коду, який може викликати взаємне блокування:

```
void thread1() {
    acquire(mutex1);
    do_something1();
    acquire(mutex2);
    do_something_else1();
    release(mutex2);
    release(mutex1);
}
void thread2() {
    acquire(mutex2);
    do_something2();
    acquire(mutex1);
    do_something_else2();
    release(mutex1);
    release(mutex2);
}
```

Живий блок (livelock) — ситуація з більш, ніж двома потоками, при якій потоки чекають розблокування один від одного, при цьому можуть змінювати свій стан, але не можуть просунутися глобально в своїй роботі. Така ситуація більш складна, ніж тупик і виникає значно рідше: як правило, вона пов'язана з часовими особливостями роботи програми.

Інверсія пріоритету — ситуація, коли потік з більштм пріоритетом змушений чекати потоки з меншим пріоритетом через неправильну синхронізацію

Голодування — ситуація, коли потік не може отримати доступ до загального ресурсу і не може просунутися. Така ситуація може бути наслідком як тупика, так і інверсії пріоритету.

Способи запобігання тупикових ситуацій

Всі способи боротьби з тупиками не є універсальними і можуть працювати тільки за певних умов. До них відносяться:

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

- монітор тупиків, який стежить за тим, щоб очікування на якомусь із замків не тривало занадто довго, і рестартує чекаючий потік в такому випадку
- нумерація замків і їх блокування тільки в монотонному порядку
- алгоритм банкіра
- та ін.

Неблокуюча синхронізація

Неблокуюча синхронізація — це група підходів, які ставлять своєю метою вирішити проблеми синхронізації альтернативним шляхом без явного використання замків і заснованих на них механізмів. Ці підходи створюють нову **конкурентную парадигму** програмування, що можна порівняти з появою структурної парадигми як запереченням підходу до написання програм з використанням низькорівневої конструкції `goto` і переходу до використання структурних блоків: ітерація, умовний вираз, цикл.

Нічого спільного (shared-nothing)

Архітектури програм без загального стану розглядають питання побудови систем з взаємодіючих компонент, які не мають поділюваних ресурсів та обмінюються інформацією тільки через передачу повідомлень. Такі системи, як правило, є набагато менш зв'язними, і тому краще піддаються масштабуванню і є менш чутливими до відмови окремих компонент.

Теоретичні роботи на цей рахунок: Взаємодіючі паралельні процеси (Communicating Parallel Processes, CPP) та модель Акторів. Практична реалізація цієї концепції — мова Erlang. У цій моделі одиницею обчислення є легковагий процес, який має "поштову скриньку", на яку йому можуть відправлятися повідомлення від інших процесів, якщо вони знають його ID в системі. Відправлення повідомлень є неблокуючим (асинхронної), а прийом є синхронним: тобто процес може заблокуватися в очікуванні повідомлення. При цьому час блокування може бути обмеженим програмно.

CSP

Взаємодіючі послідовні процеси (Communicating Sequential Processes, CSP) — це ще один підхід до організації взаємодії без використання замків. Одиницями взаємодії в цій моделі є процеси і канали. На відміну від моделі CPP, пересилання даних через канал в цій моделі відбувається, як правило, синхронно, що дає можливість встановити певну послідовність виконання процесів. Дана концепція реалізована в мові програмування Go.

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Програмна транзакційна пам'ять

Транзакція — це група послідовних операцій, яка являє собою логічну одиницю роботи з даними. Транзакція може бути виконана або цілком і успішно, дотримуючись цілісність даних і незалежно від паралельно виконуваних інших транзакцій, або не виконана взагалі і тоді вона не повинна призвести до жодного ефекту. У теорії баз даних існує концепція `ACID`, яка описує умови, які накладаються на транзакції, щоб вони мали корисну семантику. **A** означає **атомарність** — транзакція повинна виконуватися як єдине ціле. **C** означає **цілісність** (consistency) — в результаті транзакції будуть або змінені всі дані, з якими працює транзакція, або ніякі з них. **I** означає **ізоляція** — зміни даних, які виготовляються під час транзакції, стануть доступні іншим процесам тільки після її завершення. **D** означає **збереження** — після завершення транзакції система БД гарантує, що її результати зберігатися в довготривалому сховищі даних. Ці властивості, за винятком, хіба що, останнього можуть бути застосовні не тільки до БД, але і до будь-яких операцій роботи з даними. Програмна система, яка реалізує транзакційні механізми для роботи з пам'яттю — це Програмна транзакційна пам'ять (Software Transactional Memory, STM). STM лежить в основі мови програмування Clojure, а також доступна в мовах Haskell, Python (в реалізації PyPy) та інших.

Література

- A Beautiful Race Condition
- Java's Atomic and volatile, under the hood on x86
- Mutexes and Condition Variables using Futexes
- Common Pitfalls in Writing Lock-Free Algorithms
- Values and Change
- Beautiful Concurrency
- Programming Erlang: 8. Concurrent Programming

* pdf generated on 2015-01-11 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting