Синхронизация

Проблема синхронизации

Синхронизация в компьютерных системах — это координация работы процессов таким образом, чтобы последовательность их операций была предсказуемой. Как правило, синхронизация необходима при совместном доступе к разделяемым ресурсам. **Критическая секция** — часть программы, в которой есть обращение к совместно используемым данным. При нахождении в критической секции двух (или более) процессов, возникает состояние гонки/состязания за ресурсы. **Состояние гонки** — это состояние системы, при котором результат выполнения операций зависит от того, в какой последовательности будут выполняться отдельные процессы в этой системе, но управлять этой последовательностью нет возможности. Иными словами, это противоположность правильной синхронизации.

Для избежания гонок необходимо выполнение таких условий:

- **Взаимного исключения**: два процесса не должны одновременно находиться в одной критической области
- **Прогресса**: процесс, находящийся вне критической области, не может блокировать другие процессы
- **Ограниченного ожидания**: невозможна ситуация, в которой процесс вечно ждет попадания в критическую область
- Также в программе, в общем случае, не должно быть предположений о скорости или количестве процессоров

Пример условий гонок: i++

```
movl i, $eax  // i - метка адреса переменной i в памяти
incl $eax
movl $eax, i
```

Поскольку процессор не может модифицировать значения в памяти непосредственно, для этого ему нужно загрузить значение в регистр, изменить его, а затем снова выгрузить в память. Если в процессе выполнения этих трех инструкций процесс будет прерван, может возникнуть непредсказуемый результат, т.е. имеет место условие гонки.

Классические задачи синхронизации

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Классические задачи синхронизации — это модельные задачи, на которых исследуются различные ситуации, которые могут возникать в системах с разделяемым доступом и конкуренцией за общие ресурсы. К ним относятся задачи: Производитель-потребитель, Читатели-писатели, Обедающие философы, Спящий парикмахер, Курильщики сигарет, Проблема Санта-Клауса и др.

Задача **Производитель-потребитель** (также известна как задача ограниченного буфера): 2 процесса — производитель и потребитель — работают с общим ресурсом (буфером), имеющим максимальный размер N. Производитель записывает в буфер данные последовательно в ячейки 0,1,2,..., пока он не заполниться, а потребитель читает данные из буфера в обратном порядке, пока он не опустеет. Запись и считывание не могут происходить одновременно.

Наивное решение

```
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 синхронизации в этом варианте: если производитель будет прерван потребителем после того, как запишет данные в буфер `buf[count] = item`, но

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

до того, как увеличит счетчик, то потребитель считает из буфера элемент перед только что записанным, т.е. в буфере образуется дырка. После того как производитель таки увеличит счетчик, счетчик как раз будет указывать на эту дырку. Симметричная проблема есть и у потребителя.

Также у этой задачи может быть множество модификаций: например, количество производителей и потребителей может быть больше 1. В этом случае добавляются новые проблемы синхронизации.

Еще одна проблема этого решения — это бессмысленная трата вычислительных ресурсов: циклы `while (count == 0) /* do nothing */ ;` — т.н. **занятое ожидание** (busy loop, busy waiting) или же **поллинг** (pooling) — это ситуация, когда процесс не выполняет никакой полезной работы, но занимает процессор и не дает в это время работать на нем другим процессам. Таких ситуаций нужно по возможности избегать.

Алгоритмы программной синхронизации

Программный алгоритм синхронизации — это алгоритм взаимного исключения, который не основан на использовании специальных команд процессора для запрета прерываний, блокировки шины памяти и т. д. В нем используются только общие переменные памяти и цикл для ожидания входа в критическую секцию исполняемого кода. В большинстве случаев такие алгоритмы не эффективны, т.к. используют поллинг для проверки условий синхронизации.

Пример программного 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;
}
```

См. также алгоритм Деккера, алгоритм пекарни Лемпорта и др.

Аппаратные инструкции синхронизации

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Аппаратные инструкции синхронизации реализуют **атомарные** примитивные операции, на основе которых можно строить механизмы синхронизации более высокого уровня. Атомарность означает, что вся операция выполняется как целое и не может быть прерывана посредине. Атомарные примитивные операции для синхронизации, как правило, выполняют вместе 2 действия: запись значения и проверку предыдущего значения. Это дает возможность проверить условие и сразу записать такое значение, которое гарантирует, что условие больше не будет выполняться.

Try-and-set lock (TSL)

Инструкции типа `try-and-set` записывают в регистр значение из памяти, а в память — значение 1. Затем они сравнивают значение в регистре с 0. Если в памяти и был 0 (т.е. доступ к критической области был открыт), то сравнение пройдет успешно, и в то же время в память будет записан 1, что гарантирует, что в следующий раз сравнение уже не будет успешным, т.е. доступ закроется.

Реализация критической секции с помощью `TSL`:

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

То же самое на C:

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

Compare-and-swap (CAS)

Инструкции типа `compare-and-swap` записывают в регистр новое значение и при этом проверяют, что старое значение в регистре равно запомненому ранее значению.

В x86 называется `CMPXCHG`.

Аналог на C:

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

```
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 инструкции не могут отследить ситуацию, когда значение в регистре было изменено на новое, а потом снова было возвращено к предыдущему значению. В большинстве случаев это не влияет на работу алгоритма, а в тех случаях, когда влияет, необходимо использовать инструкции с проверкой на такую ситуацию, такие как `LL/SC`.

Другие аппаратные инструкции

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

Системные механизмы синхронизации

С помощью аппаратных инструкций можно реализовать более высокоуровневые конструкции, которые могут ограничивать доступ в критическую область, а также сигнализировать о каких-то событиях.

Самым простым вариантом ограничения доступа в критическую область является **переменная-замок**: если ее значение равно 0, то доступ открыт, а если 1 — то закрыт. У нее есть 2 атомарные операции:

- заблокировать (**lock**) — проверить, что значение равно 0, и устанавливает его в 1 или же ждать, пока оно не станет 0
- разблокировать (**unlock**), которая устанавливает значение в 1

Также полезной может быть операция попробовать заблокировать (**trylock**), которая не ждет пока значение замка станет 0, а сразу возвращает ответ о невозможности заблокировать замок.

Спинлок

Спинлок — это замок, ожидание при блокировке которого реализовано в виде занятого ожидания, т.е. поток "крутится" в цикле, ожидая разблокировки замка.

Реализация на ассемблере с помощью CAS:

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

```
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
```

Использование спинлока целесообразно только в тех областях кода, которые не могут вызвать блокировку, иначе все время, отведенное планировщиком потоку, ожидающему на спинлоке, будет потрачено на ожидание и при этом другие потоки не будут работать.

Семафоры

Семафор — это примитив синхронизации, позволяющий ограничить доступ к критической секции только для N потоков. При этом, как правило, семафор позволяет реализовать это без использования занятого ожидания.

Концептуально семафор включает в себя счетчик и очередь ожидания для потоков. Интерфейс семафора состоит из двух основных операций: опустить (**down**) и поднять (**up**). Операция опустить атомарно проверяет, что счетчик больше 0 и уменьшает его. Если счетчик равен 0, поток блокируется и ставиться в очередь ожидания. Операция поднять увеличивает счетчик и

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

посылает ожидающим потокам сигнал пробудиться, после чего один из этих потоков сможет повторить операцию опустить.

Бинарный семафор — это семафор с N = 1.

Мьютекс (mutex)

Мьютекс — от словосочетания mutual exclusion, т.е. взаимное исключение — это примитив синхронизации, напоминающий бинарный семафор с дополнительным условием: разблокировать его должен тот же поток, который и заблокировал.

Реализация мьютекса с помощью примитива CAS:

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

Стоит отметить, что не все системы предоставляют гарантии того, что мьютекс будет разблокирован именно заблокировавшим его потоком. В приведенном выше примере это условие как раз не проверяется.

Мьютекс с возможностью повторного входа (re-entrant mutex) — это мьютекс, который позволяет потоку несколько раз блокировать его.

RW lock

RW lock — это особый вид замка, который позволяет разграничить потоки, которые выполняют только чтение данных, и которые выполняют их модификацию. Он имеет операции заблокировать на чтение (**rdlock**), которая может одновременно выполняться несколькими потоками, и заблокировать на запись (**wrlock**), которая может выполняться только 1 потоком. Также правильные реализации RW-замка позволяют избежать проблемы инверсии приоритетов (см. ниже).

Переменные условия и мониторы

Монитор — это механизм синхронизации в объектно-ориентированном программировании, при использовании которого объект помечается как синхронизированный и компилятор добавляет к вызовам всех его методов (или только выделенных синхронизированных методов) блокировку с помощью

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

мьютекса. При этом код, использующий этот объект, не должен заботиться о синхронизации. В этом смысле монитор является более высокоуровневой конструкцией, чем семафоры и мьютексы.

Переменная условия (condition variable) — примитив синхронизации, позволяющий реализовать ожидание какого-то события и оповещение о нем. Над ней можно выполнять такие действия:

- ожидать (**wait**) сообщения о каком-то событии
- сигнализировать событие всем потокам, ожидающим на данной переменной. Сигнализация может быть блокирующей (**signal**) — в этом случае управление переходит к ожидающему потоку, — и неблокирующей (**notify**) — в этом случае управление остается у сигнализирующего потока

Большинство мониторов поддерживают внутри себя использование переменных условия. Это позволяет нескольким потоком заходить в монитор и передавать управление друг другу через эту переменную.

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

Интерфейс синхронизации

POSIX Threads (Pthreads) — это часть стандарта POSIX, которая описывает базовые примитивы синхронизации, поддерживаемые в Unix системах. Эти примитивы включают семафор, мьютекс и переменные условия.

Futex (быстрый замок в пользовательском пространстве) — это реализация мьютекса в Unix-системах, которая оптимизирована для минимального использования функций ядра ОС, за счет чего достигается более быстрая работа с ним. С помощью фьютексов в Linux реализованы семафоры и переменные условия.

Над фьютексами можно делать такие базовые операции:

- ожидать — `wait(addr, val)` — проверяет, что значение по адресу `addr` равно `val`, и, если это так, то переводит поток в состояние ожидания, а иначе продолжает работу (т.е. входит в критическую область)
- разбудить — `wake(addr, n)` — посылает `n` потокам, ожидающих на фьютексе по адресу `addr` опопвещение о необходимости проснуться

Проблемы синхронизации

Помимо условий гонки и занятого ожидания неправильная синхронизация

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

может привести к следующим проблемам:

Тупик/взаимная блокировка (deadlock) — ситуация, когда 2 или более потоков ожидают разблокировки замков друг от друга и не могут продвинуться. Простейший пример кода, который может вызвать взаимную блокировку:

```
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);
}
```

Живой блок (livelock) — ситуация с более, чем двумя потоками, при которой потоки ожидают разблокировки друг от друга, при этом могут менять свое состояние, но не могут продвинуться глобально в своей работе. Такая ситуация более сложная, чем тупик и возникает значительно реже: как правило, она связанна с временными особенностями работы программы.

Инверсия приоритета — ситуация, когда более поток с большим приоритетом вынужден ожидать потоки с меньшим приоритетом из-за неправильной синхронизации.

Голодание — ситуация, когда поток не может получить доступ к общему ресурсу и не может продвинуться. Такая ситуация может быть следствием как тупика, так и инверсии приоритета.

Способы предотвращения тупиковых ситуаций

Все способы борьбы с тупиками не являются универсальными и могут работать только при определенных условиях. К ним относятся:

- монитор тупиков, который следит за тем, чтобы ожидание на каком-то из замков не длилось слишком долго, и рестартует ожидающий поток в таком случае

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

- нумерация замков и их блокировка только в монотонном порядке
- Алгоритм банкира

Неблокирующая синхронизация

Неблокирующая синхронизация — это группа подходов, которые ставят своей целью решить проблемы синхронизации альтернативным путем без явного использования замков и основанных на них механизмов. Эти подходы создают новую **конкурентную парадигму** программирования, что можно сравнить с появлением структурной парадигмы как отрицанием подхода к написанию программ с использованием низкоуровневой конструкции `goto` и перехода к использованию структурных блоков: итерация, условное выражение, цикл.

Ничего общего (shared-nothing)

Архитектуры программ без общего состояния рассматривают вопросы построения систем из взаимодействующих компонент, которые не имеют разделяемых ресурсов и обмениваются информацией только через передачу сообщений. Такие системы, как правило, являются намного менее связными, и поэтому лучше поддаются масштабированию и являются менее чувствительными к отказу отдельных компонент.

Теоретические работы на этот счет: Взаимодействующие параллельные процессы (Communicating Parallel Processes, CPP) и модель Акторов. Практическая реализация этой концепции — язык Erlang. В этой модели единицей вычисления является легковесный процесс, который имеет "почтовый ящик", на который ему могут отправляться сообщения от других процессов, если они знают его ID в системе. Отправка сообщений является неблокирующей (асинхронной), а прием является синхронным: т.е. процесс может заблокироваться в ожидании сообщения. При этом время блокировки может быть ограниченно программно.

CSP

Взаимодействующие последовательные процессы (Communicating Sequential Processes, CSP) — это еще один подход к организации взаимодействия без использования замков. Единицами взаимодействия в этой модели являются процессы и каналы. В отличие от модели CPP, пересылка данных через канал в этой модели происходит, как правило, синхронно, что дает возможность установить определенную последовательность выполнения процессов. Данная концепция реализована в языке программирования Go.

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

Программная транзакционная память

Транзакция — это группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. В теории баз данных существует концепция `ACID`, которая описывает условия, которые накладываются на транзакции, чтобы они имели полезную семантику. **A** означает **атомарность** — транзакция должна выполняться как единое целое. **C** означает **целостность** (consistency) — в результате транзакции будут либо изменены все данные, с которыми работает транзакция, либо никакие из них. **I** означает **изоляция** — изменения данных, которые производяться во время транзакции, станут доступны другим процессам только после ее завершения. **D** означает **сохранность** — после завершения транзакции система БД гарантирует, что ее результаты сохраняться в долговременном хранилище данных. Эти свойства, за исключением, разве что, последнего могут быть применимы не только к БД, но и к любым операциям работы с данными. Программная система, которая реализует транзакционные механизмы для работы с разделяемой памятью — это Программная транзакционная память (Software Transactional Memory, STM). STM лежит в основе языка программирования Clojure, а также доступна в языках Haskell, Python (в реализации PyPy) и других.

Литература

- 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 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting