

Операционные Системы

Синхронизация потоков

May 5, 2017

Пример

```
1      .data
2  counter:
3      .int 0
4
5      .text
6  add_one:
7      movq counter, %rax
8      inc %rax
9      movq rax, counter
10     retq
```

```
1  int counter;
2
3  int add_one(void)
4  {
5      return ++counter;
6  }
```

Вариант 1

```
1  movq counter, %rax
2  inc %rax
3  movq rax, counter
4
5
6
```

```
1
2
3
4  movq counter, %rax
5  inc %rax
6  movq rax, counter
```

Вариант 2

```
1  movq counter, %rax
2
3  inc %rax
4  movq rax, counter
5
6
```

```
1
2  movq counter, %rax
3
4
5  inc %rax
6  movq rax, counter
```

Состояние гонки

- ▶ Состояние гонки - результат зависит от порядка выполнения инструкций

Состояние гонки

- ▶ Состояние гонки - результат зависит от порядка выполнения инструкций
 - ▶ порядок зависит от слишком многих факторов;

Состояние гонки

- ▶ Состояние гонки - результат зависит от порядка выполнения инструкций
 - ▶ порядок зависит от слишком многих факторов;
 - ▶ решения планировщика, влияние других потоков, прерывания...

Состояние гонки

- ▶ Состояние гонки - результат зависит от порядка выполнения инструкций
 - ▶ порядок зависит от слишком многих факторов;
 - ▶ решения планировщика, влияние других потоков, прерывания...
 - ▶ могут быть трудно воспроизводимы - не поддаются тестированию.

Критическая секция

- ▶ Критическая секция

Критическая секция

- ▶ Критическая секция
 - ▶ участок кода, обращающийся к разделяемым несколькими потоками данным;

Критическая секция

- ▶ Критическая секция
 - ▶ участок кода, обращающийся к разделяемым несколькими потоками данным;
 - ▶ если не более, чем один поток может одновременно находиться в критической секции, то не будет состояния гонки.

Блокировка

- ▶ Блокировка (lock) - некоторый объект и пара методов для работы с ним

Блокировка

- ▶ Блокировка (lock) - некоторый объект и пара методов для работы с ним
 - ▶ lock - метод захвата блокировки;

Блокировка

- ▶ Блокировка (lock) - некоторый объект и пара методов для работы с ним
 - ▶ lock - метод захвата блокировки;
 - ▶ unlock - метод освобождения блокировки.

Свойство взаимного исключения

- ▶ Взаимное исключение (mutual exclusion)

Свойство взаимного исключения

- ▶ Взаимное исключение (mutual exclusion)
 - ▶ потоки всегда вызывают lock и unlock парами (сначала lock, а потом unlock);

Свойство взаимного исключения

- ▶ Взаимное исключение (mutual exclusion)
 - ▶ потоки всегда вызывают lock и unlock парами (сначала lock, а потом unlock);
 - ▶ не более одного потока может одновременно находиться между lock-ом и unlock-ом.

Свойство взаимного исключения

```
struct lock l;  
int counter;  
  
int add_one(void)  
{  
    int res;  
  
    lock(&l);  
    res = ++counter;  
    unlock(&l);  
}
```

Свойство взаимного исключения

```
1  struct lock lock0;
2  int counter0;
3
4  int add_one0(void)
5  {
6      int res;
7
8      lock(&lock0);
9      res = ++counter0;
10     unlock(&lock0);
11     return res;
12 }
```

```
1  struct lock lock1;
2  int counter1;
3
4  int add_one1(void)
5  {
6      int res;
7
8      lock(&lock1);
9      res = ++counter1;
10     unlock(&lock1);
11     return res;
12 }
```

СВОЙСТВО ЖИВОСТИ

```
struct lock {  
};  
  
void lock(struct lock *unused)  
{  
    (void) unused;  
    while (1);  
}  
  
void unlock(struct lock *unused)  
{  
    (void) unused;  
}
```

Свойство живости

- ▶ Свойство живости (deadlock freedom)

Свойство живости

- ▶ Свойство живости (deadlock freedom)
 - ▶ если один из потоков вызвал lock, то какой-то из потоков, вызвавших lock, захватит блокировку;

Свойство живости

- ▶ Свойство живости (deadlock freedom)
 - ▶ если один из потоков вызвал lock, то какой-то из потоков, вызвавших lock, захватит блокировку;
 - ▶ поток не ждет в lock, если он единственный пытается захватить блокировку;

Свойство живости

- ▶ Свойство живости (deadlock freedom)
 - ▶ если один из потоков вызвал lock, то какой-то из потоков, вызвавших lock, захватит блокировку;
 - ▶ поток не ждет в lock, если он единственный пытается захватить блокировку;
 - ▶ если поток ждет, значит другому потоку повезло захватить блокировку.

На что нельзя полагаться?

- ▶ Скорость работы потоков:

На что нельзя полагаться?

- ▶ Скорость работы потоков:
 - ▶ мы не знаем, сколько времени потребуется потоку, чтобы выполнить какой-то код;

На что нельзя полагаться?

- ▶ Скорость работы потоков:
 - ▶ мы не знаем, сколько времени потребуется потоку, чтобы выполнить какой-то код;
 - ▶ мы не можем полагать, что какой-то поток быстрее.

На что можно полагаться?

- ▶ Потоки работают корректно:

На что можно полагаться?

- ▶ Потоки работают корректно:
 - ▶ поток не находится между lock и unlock бесконечно;

На что можно полагаться?

- ▶ Потоки работают корректно:
 - ▶ поток не находится между lock и unlock бесконечно;
 - ▶ поток не "падает", находясь между lock и unlock;

На что можно полагаться?

- ▶ Потоки работают корректно:
 - ▶ поток не находится между lock и unlock бесконечно;
 - ▶ поток не "падает", находясь между lock и unlock;
 - ▶ и так далее...

Атомарный Read/Write регистр

- ▶ Атомарный RW регистр - ячейка памяти и пара операций

Атомарный Read/Write регистр

- ▶ Атомарный RW регистр - ячейка памяти и пара операций
 - ▶ write - "атомарно" записывает значение в регистр;

Атомарный Read/Write регистр

- ▶ Атомарный RW регистр - ячейка памяти и пара операций
 - ▶ write - "атомарно" записывает значение в регистр;
 - ▶ read - "атомарно" читает последнее записанное значение;

Атомарный Read/Write регистр

- ▶ Атомарный RW регистр - ячейка памяти и пара операций
 - ▶ write - "атомарно" записывает значение в регистр;
 - ▶ read - "атомарно" читает последнее записанное значение;
 - ▶ все операции (read/write) упорядочены.

Взаимное исключение для 2-х потоков

- ▶ Есть всего два потока

Взаимное исключение для 2-х потоков

- ▶ Есть всего два потока
 - ▶ потоки имеют идентификаторы 0 и 1;

Взаимное исключение для 2-х потоков

- ▶ Есть всего два потока
 - ▶ потоки имеют идентификаторы 0 и 1;
 - ▶ внутри потока мы можем узнать его идентификатор (пусть за это отвечает функция `threadId`).

Альтернатива

```
struct lock {
    atomic_int last;
};

void lock_init(struct lock *lock)
{
    atomic_store(&lock->last, 0);
}

void lock(struct lock *lock)
{
    while (atomic_load(&lock->last) == threadId());
}

void unlock(struct lock *lock)
{
    atomic_store(&lock->last, threadId());
}
```

Свойство взаимного исключения

- ▶ Для приведенного алгоритма взаимное исключение гарантируется

Свойство взаимного исключения

- ▶ Для приведенного алгоритма взаимное исключение гарантируется
 - ▶ lock может вернуть управление только потоку с идентификатором, не равным $\text{lock} \rightarrow \text{last}$;

Свойство взаимного исключения

- ▶ Для приведенного алгоритма взаимное исключение гарантируется
 - ▶ lock может вернуть управление только потоку с идентификатором, не равным $\text{lock} \rightarrow \text{last}$;
 - ▶ только поток с $\text{threadId}() \neq \text{lock} \rightarrow \text{last}$ может изменить значение $\text{lock} \rightarrow \text{last}$.

Свойство живости

- ▶ Пусть поток 1 вообще никогда не пытается захватить лок

Свойство живости

- ▶ Пусть поток 1 вообще никогда не пытается захватить лок
 - ▶ если поток 0 вызовет lock, то он зависнет навсегда;

Свойство живости

- ▶ Пусть поток 1 вообще никогда не пытается захватить лок
 - ▶ если поток 0 вызовет lock, то он зависнет навсегда;
 - ▶ т. е. свойство живости не выполняется.

Флаги намерения

```
struct lock {
    atomic_int flag[2];
};

void lock_init(struct lock *lock)
{
    atomic_store(&lock->flag[0], 0);
    atomic_store(&lock->flag[1], 0);
}

void lock(struct lock *lock)
{
    const int me = threadId();
    const int other = 1 - me;

    atomic_store(&lock->flag[me], 1);
    while (atomic_load(&lock->flag[other]));
}

void unlock(struct lock *lock)
{
    const int me = threadId();

    atomic_store(&lock->flag[me], 0);
}
```

Корректность

- ▶ Гарантируется ли взаимное исключение?

Корректность

- ▶ Гарантируется ли взаимное исключение?
- ▶ Гарантируется ли живость?

Алгоритм Петерсона для 2-х потоков

```
struct lock {
    atomic_int last;
    atomic_int flag[2];
};

void lock(struct lock *lock)
{
    const int me = threadId();
    const int other = 1 - me;

    atomic_store(&lock->flag[me], 1);
    atomic_store(&lock->last, me);

    while (atomic_load(lock->flag[other])
           && atomic_load(&lock->last) == me);
}

void unlock(struct lock *lock)
{
    const int me = threadId();

    atomic_store(&lock->flag[me], 0);
}
```

Взаимное исключение

- ▶ Доказательство от противного - пусть два потока одновременно находятся в критической секции

Взаимное исключение

- ▶ Доказательство от противного - пусть два потока одновременно находятся в критической секции
 - ▶ оба потока записывали значение в атомарный регистр `last`;

Взаимное исключение

- ▶ Доказательство от противного - пусть два потока одновременно находятся в критической секции
 - ▶ оба потока записывали значение в атомарный регистр `last`;
 - ▶ один из них должен был быть первым, а другой последним;

Взаимное исключение

- ▶ Доказательство от противного - пусть два потока одновременно находятся в критической секции
 - ▶ оба потока записывали значение в атомарный регистр `last`;
 - ▶ один из них должен был быть первым, а другой последним;
 - ▶ для определенности пусть последним был поток 1.

Взаимное исключение

- ▶ Итак нам известно следующее:

Взаимное исключение

- ▶ Итак нам известно следующее:
 - ▶ `lock->last == 1` - последним туда записал поток 1;

Взаимное исключение

- ▶ Итак нам известно следующее:
 - ▶ `lock->last == 1` - последним туда записал поток 1;
 - ▶ `lock->flag[0] = 1` и `lock->flag[1] == 1`.

Взаимное исключение

- ▶ Как в таких условиях поток 1 мог пройти мимо цикла в lock и войти в критическую секцию?

Взаимное исключение

- ▶ Как в таких условиях поток 1 мог пройти мимо цикла в lock и войти в критическую секцию?
 - ▶ очевидно, никак.

Живость

- ▶ Пусть поток 0 пытается войти в критическую секцию, возможны две ситуации:

Живость

- ▶ Пусть поток 0 пытается войти в критическую секцию, возможны две ситуации:
 - ▶ при проверке условия цикла
`lock->flag[1] == 0;`

Живость

- ▶ Пусть поток 0 пытается войти в критическую секцию, возможны две ситуации:
 - ▶ при проверке условия цикла `lock->flag[1] == 0;`
 - ▶ при проверке условия цикла `lock->flag[1] == 1.`

Живость

- ▶ В первом случае ($\text{lock} \rightarrow \text{flag}[1] == 0$)

Живость

- ▶ В первом случае ($\text{lock} \rightarrow \text{flag}[1] == 0$)
 - ▶ поток 1 даже не пытался захватить блокировку;

Живость

- ▶ В первом случае ($\text{lock} \rightarrow \text{flag}[1] == 0$)
 - ▶ поток 1 даже не пытался захватить блокировку;
 - ▶ условие цикла, очевидно, ложно, и поток 0 входит в критическую секцию

Живость

- ▶ Во втором случае ($\text{lock} \rightarrow \text{flag}[1] == 1$)

Живость

- ▶ Во втором случае ($\text{lock} \rightarrow \text{flag}[1] == 1$)
 - ▶ оба потока изъявили намерение войти в критическую секцию;

Живость

- ▶ Во втором случае ($\text{lock} \rightarrow \text{flag}[1] == 1$)
 - ▶ оба потока изъявили намерение войти в критическую секцию;
 - ▶ нужно показать, что хотя бы один из них рано или поздно войдет в критическую секцию (или уже там).

Живость

- ▶ Оба потока после записи в `lock->flag[x]` должны в какой-то момент записать в `lock->last`

Живость

- ▶ Оба потока после записи в `lock->flag[x]` должны в какой-то момент записать в `lock->last`
 - ▶ не трудно увидеть, что если `lock->flag[0] == 1` и `lock->flag[1] == 1`,
 - ▶ то тот из них, кто сделал это первым, войдет в критическую секцию.

N потоков

- ▶ Реализовав взаимное исключение для 2-х потоков, мы можем реализовать взаимное исключение для любого числа потоков

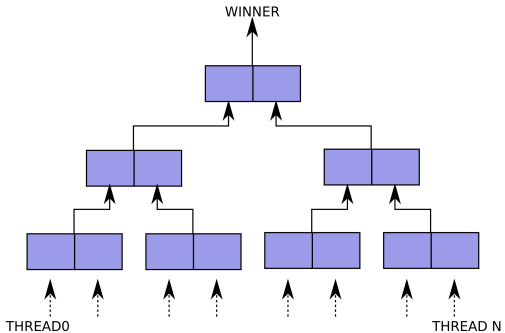
N потоков

- ▶ Реализовав взаимное исключение для 2-х потоков, мы можем реализовать взаимное исключение для любого числа потоков
 - ▶ организуем турнир для N потоков;

N потоков

- ▶ Реализовав взаимное исключение для 2-х потоков, мы можем реализовать взаимное исключение для любого числа потоков
 - ▶ организуем турнир для N потоков;
 - ▶ потоки конкурируют друг с другом на "выбывание".

N потоков



Алгоритм Петерсона для N потоков

```
struct lock_one {
    atomic_int last;
    atomic_int flag[N];
};

int flags_clear(const struct lock_one *lock)
{
    const int me = threadId();

    for (int i = 0; i != N; ++i) {
        if (i != me && atomic_load(&lock->flag[i]))
            return 0;
    }
    return 1;
}

void lock_one(struct lock_one *lock)
{
    const int me = threadId();

    atomic_store(&lock->flag[me], 1);
    atomic_store(&lock->last, me);

    while (!flags_clear(lock)
        && atomic_load(&lock->last) == me);
}

void unlock_one(struct lock_one *lock)
{
    const int me = threadId();

    atomic_store(&lock->flag[me], 0);
}
```

Алгоритм Петерсона для N потоков

```
struct lock {  
    struct lock_one lock[N - 1];  
};  
  
void lock(struct lock *lock)  
{  
    for (int i = 0; i != N - 1; ++i)  
        lock_one(&lock->lock[i]);  
}  
  
void unlock(struct lock *lock)  
{  
    for (int i = N - 2; i >= 0; --i)  
        unlock_one(&lock->lock[i]);  
}
```

Алгоритм Петерсона для N потоков

```
struct lock {
    atomic_int level[N];
    atomic_int last[N - 1];
};

void lock(struct lock *lock)
{
    const int me = threadId();

    for (int i = 0; i != N - 1; ++i) {
        atomic_store(&lock->level[me], i + 1);
        atomic_store(&lock->last[i], me);

        while (!flags_clear(lock, i)
               && atomic_load(&lock->last[i]) == me);
    }
}

void unlock(struct lock *lock)
{
    const int me = threadId();

    atomic_store(&lock->level[me], 0);
}
```

Честность

- ▶ Не хочется, чтобы потоки голодали!

Честность

- ▶ Не хочется, чтобы потоки голодали!
 - ▶ если поток захотел захватить блокировку, то когда-нибудь ему это удастся;

Честность

- ▶ Не хочется, чтобы потоки голодали!
 - ▶ если поток захотел захватить блокировку, то когда-нибудь ему это удастся;
 - ▶ сравните с жизнью - среди потоков, пытающихся захватить блокировку, одному это удастся.

Супер честность

- ▶ k -ограниченное ожидание:

Супер честность

- ▶ k -ограниченное ожидание:
 - ▶ после того как поток "изъявил" желание захватить блокировку (встал в очередь), не более k потоков могут пролезть вперед него без очереди.

Алгоритм Петерсона на примере 3 потоков

№	level[0]	level[1]	level[2]	last[0]	last[1]
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	1	0
3	1	1	1	2	0
4	2	1	1	2	0
5	0	1	1	2	0
6	1	1	1	0	0
7	1	1	2	0	2
8	1	1	0	0	2
9	1	1	1	2	2
10	2	1	1	2	0

Атомарный Read/Modify/Write регистр

- ▶ Атомарный RMW регистр позволяет за одну операцию

Атомарный Read/Modify/Write регистр

- ▶ Атомарный RMW регистр позволяет за одну операцию
 - ▶ прочесть значение в регистре;

Атомарный Read/Modify/Write регистр

- ▶ Атомарный RMW регистр позволяет за одну операцию
 - ▶ прочитать значение в регистре;
 - ▶ преобразовать некоторым образом прочитанное значение;

Атомарный Read/Modify/Write регистр

- ▶ Атомарный RMW регистр позволяет за одну операцию
 - ▶ прочитать значение в регистре;
 - ▶ преобразовать некоторым образом прочитанное значение;
 - ▶ записать преобразованное значение назад.

Атомарный Read/Modify/Write регистр

```
int atomic_rmw(int *reg, int (*f)(int))
{
    const int old = *reg;
    const int new = f(old);

    *reg = new;
    return old;
}
```

Атомарный Read/Modify/Write регистр

- ▶ `atomic_exchange` - возвращает старое значение, записывает новое;

Атомарный Read/Modify/Write регистр

- ▶ `atomic_exchange` - возвращает старое значение, записывает новое;
- ▶ `atomic_fetch_{add|sub|or|and|xor}` - выполняет арифметическое действие над атомарным регистром;

Атомарный Read/Modify/Write регистр

- ▶ `atomic_exchange` - возвращает старое значение, записывает новое;
- ▶ `atomic_fetch_{add|sub|or|and|xor}` - выполняет арифметическое действие над атомарным регистром;
- ▶ `atomic_compare_exchange` - записывает новое значение, если старое значение равно заданному.

Реализация RMW регистра

- ▶ Архитектура может поддерживать RMW операции (x86 - одна из них)

Реализация RMW регистра

- ▶ Архитектура может поддерживать RMW операции (x86 - одна из них)
 - ▶ `xchg;`

Реализация RMW регистра

- ▶ Архитектура может поддерживать RMW операции (x86 - одна из них)
 - ▶ xchg;
 - ▶ lock add, lock sub, lock or, lock and, lock xor;

Реализация RMW регистра

- ▶ Архитектура может поддерживать RMW операции (x86 - одна из них)
 - ▶ xchg;
 - ▶ lock add, lock sub, lock or, lock and, lock xor;
 - ▶ lock cmpxchg.

Реализация RMW регистра

- ▶ Архитектура может поддерживать LL/SC (например, ARM):

Реализация RMW регистра

- ▶ Архитектура может поддерживать LL/SC (например, ARM):
 - ▶ LL (load-link, load-linked, load-locked) - загружает значение из памяти;

Реализация RMW регистра

- ▶ Архитектура может поддерживать LL/SC (например, ARM):
 - ▶ LL (load-link, load-linked, load-locked) - загружает значение из памяти;
 - ▶ SC (store-conditional) - записывает новое значение в ячейку, но только если после LL эту ячейку никто не трогал;

Реализация RMW регистра

- ▶ Архитектура может поддерживать LL/SC (например, ARM):
 - ▶ LL (load-link, load-linked, load-locked) - загружает значение из памяти;
 - ▶ SC (store-conditional) - записывает новое значение в ячейку, но только если после LL эту ячейку никто не трогал;
 - ▶ LL/SC идут парами и работают вместе как одна RMW операция.

Взаимное исключение с использованием RWM регистра

```
#define LOCKED 1
#define UNLOCKED 0

struct lock {
    atomic_int locked;
};

void lock(struct lock *lock)
{
    while (atomic_exchange(&lock->locked, LOCKED) !=
           ↪ UNLOCKED);
}

void unlock(struct lock *lock)
{
    atomic_store(&lock->locked, UNLOCKED);
}
```

И снова про честность

- ▶ Что если блокировка находится под нагрузкой (high contention)?

И снова про честность

- ▶ Что если блокировка находится под нагрузкой (high contention)?
 - ▶ т. е. блокировка практически всегда занята;

И снова про честность

- ▶ Что если блокировка находится под нагрузкой (high contention)?
 - ▶ т. е. блокировка практически всегда занята;
 - ▶ некоторый поток может получать CPU только тогда, когда блокировка занята;

И снова про честность

- ▶ Что если блокировка находится под нагрузкой (high contention)?
 - ▶ т. е. блокировка практически всегда занята;
 - ▶ некоторый поток может получать CPU только тогда, когда блокировка занята;
 - ▶ такой поток будет голодать - блокировка не честная.

Ticket lock

```
struct lock {
    atomic_uint ticket;
    atomic_uint next;
};

void lock(struct lock *lock)
{
    const unsigned ticket = atomic_fetch_add(&lock->
        ↪ ticket, 1);

    while (atomic_load(&lock->next) != ticket);
}

void unlock(struct lock *lock)
{
    atomic_fetch_add(&lock->next, 1);
}
```


И снова о прерываниях

- ▶ Пусть у нас есть устройство, которое получает данные из сети

И снова о прерываниях

- ▶ Пусть у нас есть устройство, которое получает данные из сети
 - ▶ устройство сигнализирует процессору - генерирует прерывание;

И снова о прерываниях

- ▶ Пусть у нас есть устройство, которое получает данные из сети
 - ▶ устройство сигнализирует процессору - генерирует прерывание;
 - ▶ процессор вызывает обработчик прерывания - функцию ядра ОС;

И снова о прерываниях

- ▶ Пусть у нас есть устройство, которое получает данные из сети
 - ▶ устройство сигнализирует процессору - генерирует прерывание;
 - ▶ процессор вызывает обработчик прерывания - функцию ядра ОС;
 - ▶ обработчик прерывания должен забрать данные с устройства и положить их в буфер, из которого какой-то поток сможет их забрать.

И снова о прерываниях

- ▶ Что если к этому буферу могут обращаться из нескольких потоков?

И снова о прерываниях

- ▶ Что если к этому буферу могут обращаться из нескольких потоков?
 - ▶ мы должны защитить буфер блокировкой;

И снова о прерываниях

- ▶ Что если к этому буферу могут обращаться из нескольких потоков?
 - ▶ мы должны защитить буфер блокировкой;
 - ▶ потоки и обработчики прерываний должны захватывать эту блокировку перед обращением к буферу;

И снова о прерываниях

- ▶ Что если к этому буферу могут обращаться из нескольких потоков?
 - ▶ мы должны защитить буфер блокировкой;
 - ▶ потоки и обработчики прерываний должны захватывать эту блокировку перед обращением к буферу;
 - ▶ что если обработчик прерывания устройства прервал поток, который захватил блокировку?

Deadlock

- ▶ Прерванный поток и обработчик прерывания ждут друг друга:

Deadlock

- ▶ Прерванный поток и обработчик прерывания ждут друг друга:
 - ▶ обработчик прерывания не может захватить блокировку, потому что ее держит прерванный поток;

Deadlock

- ▶ Прерванный поток и обработчик прерывания ждут друг друга:
 - ▶ обработчик прерывания не может захватить блокировку, потому что ее держит прерванный поток;
 - ▶ пока обработчик прерывания не завершится, прерванный поток не получит управление и не сможет отпустить блокировку.

Мораль

- ▶ Если блокировка защищает данные, к которым обращается обработчик прерывания, то нужно выключать прерывания

Мораль

- ▶ Если блокировка защищает данные, к которым обращается обработчик прерывания, то нужно выключать прерывания
 - ▶ если прерывания отключены, то deadlock между обработчиком прерывания и прерванным потоком не может возникнуть.

Однопроцессорные системы

- ▶ Представим систему с всего одним ядром/процессором

Однопроцессорные системы

- ▶ Представим систему с всего одним ядром/процессором
 - ▶ запретив прерывания и переключение потоков, мы получаем CPU в монопольное пользование;

Однопроцессорные системы

- ▶ Представим систему с всего одним ядром/процессором
 - ▶ запретив прерывания и переключение потоков, мы получаем CPU в монопольное пользование;
 - ▶ все рассмотренные ранее алгоритмы просто не нужны.

Разделение на читателей и писателей

- ▶ Не все запросы к разделяемым данным одинаковы

Разделение на читателей и писателей

- ▶ Не все запросы к разделяемым данным одинаковы
 - ▶ есть запросы, которые модифицируют данные;

Разделение на читателей и писателей

- ▶ Не все запросы к разделяемым данным одинаковы
 - ▶ есть запросы, которые модифицируют данные;
 - ▶ есть запросы, которые только читают данные.

Разделение на читателей и писателей

```
struct rwlock {
    atomic_uint ticket;
    atomic_uint write;
    atomic_uint read;
};

void read_lock(struct rwlock *lock)
{
    const unsigned ticket = atomic_fetch_add(&lock->
        ↪ ticket, 1);

    while (atomic_load(&lock->read) != ticket);
    atomic_store(&lock->read, ticket + 1);
}

void read_unlock(struct rwlock *lock)
{
    atomic_fetch_add(&lock->write, 1);
}
```

Разделение на читателей и писателей

```
struct rwlock {
    atomic_uint ticket;
    atomic_uint write;
    atomic_uint read;
};

void write_lock(struct rwlock *lock)
{
    const unsigned ticket = atomic_fetch_add(&lock->
        ↪ ticket, 1);

    while (atomic_load(&lock->write) != ticket);
}

void write_unlock(struct rwlock *lock)
{
    atomic_fetch_add(&lock->read, 1);
    atomic_fetch_add(&lock->write, 1);
}
```

Стратегии ожидания

- ▶ До сих пор функция `lock` всегда просто ждала в цикле

Стратегии ожидания

- ▶ До сих пор функция lock всегда просто ждала в цикле
 - ▶ такая стратегия называется активным ожиданием;

Стратегии ожидания

- ▶ До сих пор функция lock всегда просто ждала в цикле
 - ▶ такая стратегия называется активным ожиданием;
 - ▶ блокировки, использующие активное ожидание, часто называются spinlock-ами;
 - ▶ они "крутятся" в цикле.

Активное ожидание

- ▶ Активное ожидание хорошо работает если:

Активное ожидание

- ▶ Активное ожидание хорошо работает если:
 - ▶ потоки не держат блокировку очень долго;

Активное ожидание

- ▶ Активное ожидание хорошо работает если:
 - ▶ потоки не держат блокировку очень долго;
 - ▶ блокировка не находится под сильной нагрузкой;

Активное ожидание

- ▶ Активное ожидание хорошо работает если:
 - ▶ потоки не держат блокировку очень долго;
 - ▶ блокировка не находится под сильной нагрузкой;
 - ▶ т. е. если активное ожидание длится недолго.

Альтернативы активному ожиданию

- ▶ Как можно ожидать не активно?

Альтернативы активному ожиданию

- ▶ Как можно ожидать не активно?
 - ▶ можно добровольно отдать CPU (переключиться на другой поток);

Альтернативы активному ожиданию

- ▶ Как можно ожидать не активно?
 - ▶ можно добровольно отдать CPU (переключиться на другой поток);
 - ▶ можно пометить поток как неактивный, чтобы планировщик не давал ему время на CPU, пока блокировка не будет отпущена.

Задача Producer-a и Consumer-a

- ▶ Рассмотрим следующую задачу:

Задача Producer-a и Consumer-a

- ▶ Рассмотрим следующую задачу:
 - ▶ Producer - поток/потoki, который генерирует данные;

Задача Producer-a и Consumer-a

- ▶ Рассмотрим следующую задачу:
 - ▶ Producer - поток/потоки, который генерирует данные;
 - ▶ Consumer - поток/потоки, который потребляет данные;

Задача Producer-а и Consumer-а

- ▶ Рассмотрим следующую задачу:
 - ▶ Producer - поток/потоки, который генерирует данные;
 - ▶ Consumer - поток/потоки, который потребляет данные;
 - ▶ что если Producer и Consumer работают с разной скоростью?

Переменная состояния

- ▶ Переменная состояния (condition variable) - объект и несколько методов для работы с ним

Переменная состояния

- ▶ Переменная состояния (condition variable) - объект и несколько методов для работы с ним
 - ▶ wait - ожидает, пока кто-нибудь не просигналит;

Переменная состояния

- ▶ Переменная состояния (condition variable) - объект и несколько методов для работы с ним
 - ▶ wait - ожидает, пока кто-нибудь не просигналит;
 - ▶ notify_one - просигналить одному из ожидающих;

Переменная состояния

- ▶ Переменная состояния (condition variable) - объект и несколько методов для работы с ним
 - ▶ wait - ожидает, пока кто-нибудь не просигналит;
 - ▶ notify_one - просигналить одному из ожидающих;
 - ▶ notify_all - просигналить всем ожидающим.

Переменная состояния

```
struct lock;  
void lock(struct lock *lock);  
void unlock(struct lock *lock);  
  
struct condition;  
void wait(struct condition *cv, struct lock *lock);  
void notify_one(struct condition *cv);  
void notify_all(struct condition *cv);
```


Producer

```
struct condition cv;
struct lock mtx;
int value;
bool valid_value;
bool done;

void produce(int x)
{
    lock(&mtx);
    while (valid_value)
        wait(&cv, &mtx);
    value = x;
    valid_value = true;
    notify_one(&cv);
    unlock(&mtx);
}

void finish(void)
{
    lock(&mtx);
    done = true;
    notify_all(&cv);
    unlock(&mtx);
}
```

Consumer

```
int consume(int *x)
{
    int ret = 0;
    lock(&mtx);

    while (!valid_value && !done)
        wait(&cv, &mtx);

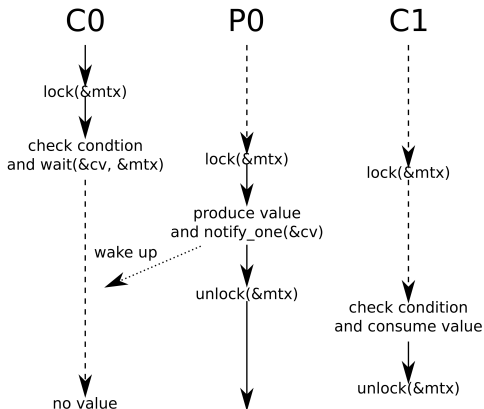
    if (valid_value) {
        *x = value;
        valid_value = false;
        notify_one(&cv);
        ret = 1;
    }
    unlock(&mtx);
    return ret;
}
```

Зачем нам lock?

```
1
2
3  /* lock(&mtx); */
4  done = true;
5  notify_all(&cv);
6  /* unlock(&mtx); */
7
8
9
```

```
1  /* lock(&mtx); */
2  while (... && !done)
3
4
5
6          wait(&cv, &mtx);
7  ...
8
9  /* unlock(&mtx); */
```

Зачем нам цикл?



Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда wait возвращает управление, даже если никто не сигнализировал

Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда wait возвращает управление, даже если никто не сигнализировал
 - ▶ многие реализации переменной состояния подвержены:

Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда wait возвращает управление, даже если никто не сигнализировал
 - ▶ многие реализации переменной состояния подвержены:
 - ▶ C++;

Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда wait возвращает управление, даже если никто не сигнализировал
 - ▶ многие реализации переменной состояния подвержены:
 - ▶ C++;
 - ▶ Java;

Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда wait возвращает управление, даже если никто не сигнализировал
 - ▶ многие реализации переменной состояния подвержены:
 - ▶ C++;
 - ▶ Java;
 - ▶ POSIX Threads...

Deadlock

- ▶ Deadlock - ситуация, при которой потоки не могут работать, потому что ждут друг друга:

Deadlock

- ▶ Deadlock - ситуация, при которой потоки не могут работать, потому что ждут друг друга:
 - ▶ deadlock потоком исполнения и обработчиком прерывания;

Deadlock

- ▶ Deadlock - ситуация, при которой потоки не могут работать, потому что ждут друг друга:
 - ▶ deadlock потоком исполнения и обработчиком прерывания;
 - ▶ поток А ждет, пока поток В что-то сделает (например, отпустит блокировку);

Deadlock

- ▶ Deadlock - ситуация, при которой потоки не могут работать, потому что ждут друг друга:
 - ▶ deadlock потоком исполнения и обработчиком прерывания;
 - ▶ поток А ждет, пока поток В что-то сделает (например, отпустит блокировку);
 - ▶ а поток В ничего не делает, потому что ждет, пока поток А что-то сделает (например, отпустит блокировку).

Пример

```
1 struct lock a;  
2  
3 void thread0(void)  
4 {  
5     lock(&a);  
6     lock(&b);  
7  
8     /* do something  
        ↪ */  
9  
10    unlock(&b);  
11    unlock(&b);  
12 }
```

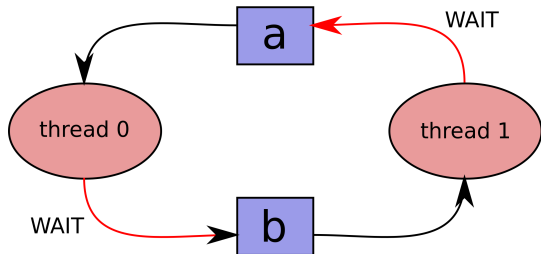
```
1 struct lock b;  
2  
3 void thread1(void)  
4 {  
5     lock(&b);  
6     lock(&a);  
7  
8     /* do something  
        ↪ else */  
9  
10    unlock(&a);  
11    unlock(&b);  
12 }
```

Пример

```
1    lock(&a);  
2  
3    lock(&b);  
4
```

```
1  
2    lock(&b);  
3  
4    lock(&a);
```

Wait-for graph



Deadlock

- ▶ Как и с состоянием гонки, deadlock не поддается тестированию

Deadlock

- ▶ Как и с состоянием гонки, deadlock не поддается тестированию
 - ▶ появление зависит от многих факторов;

Deadlock

- ▶ Как и с состоянием гонки, deadlock не поддается тестированию
 - ▶ появление зависит от многих факторов;
 - ▶ входные данные, решения планировщика, прерывания, производительность оборудования ...

Предотвращение deadlock-ов

- ▶ Мы хотим избежать появления цикла в wait-for графе

Предотвращение deadlock-ов

- ▶ Мы хотим избежать появления цикла в wait-for графе
 - ▶ простой случай - все блокировки известны заранее;

Предотвращение deadlock-ов

- ▶ Мы хотим избежать появления цикла в wait-for графе
 - ▶ простой случай - все блокировки известны заранее;
 - ▶ упорядочим все блокировки (например, по адресу);

Предотвращение deadlock-ов

- ▶ Мы хотим избежать появления цикла в wait-for графе
 - ▶ простой случай - все блокировки известны заранее;
 - ▶ упорядочим все блокировки (например, по адресу);
 - ▶ захватываем блокировки только по порядку.

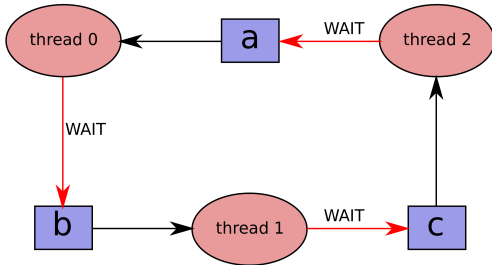
Пример

```
1 void thread0 ()
2 {
3     lock(&a);
4     lock(&b);
5     ...
6     unlock(&b);
7     unlock(&a);
8 }
```

```
1 void thread1 ()
2 {
3     lock(&b);
4     lock(&c);
5     ...
6     unlock(&c);
7     unlock(&b);
8 }
```

```
1 void thread2 ()
2 {
3     lock(&c);
4     lock(&a);
5     ...
6     unlock(&a);
7     unlock(&c);
8 }
```


Пример



Пример

- ▶ Отсортируем блокировки a, b и c по алфавиту:

Пример

- ▶ Отсортируем блокировки a, b и c по алфавиту:
 - ▶ каждый поток должен захватывать блокировки только согласно порядку;

Пример

- ▶ Отсортируем блокировки a, b и c по алфавиту:
 - ▶ каждый поток должен захватывать блокировки только согласно порядку;
 - ▶ например, поток 2 хочет захватить блокировки c и a:

Пример

- ▶ Отсортируем блокировки a, b и c по алфавиту:
 - ▶ каждый поток должен захватывать блокировки только согласно порядку;
 - ▶ например, поток 2 хочет захватить блокировки c и a:
 - ▶ так как a в алфавитие раньше c, то сначала хватаем a,

Пример

- ▶ Отсортируем блокировки a, b и c по алфавиту:
 - ▶ каждый поток должен захватывать блокировки только согласно порядку;
 - ▶ например, поток 2 хочет захватить блокировки c и a:
 - ▶ так как a в алфавитие раньше c, то сначала хватаем a,
 - ▶ потом хватаем c.

Пример

```
1 void thread0 ()
2 {
3     lock(&a);
4     lock(&b);
5     ...
6     unlock(&b);
7     unlock(&a);
8 }
```

```
1 void thread1 ()
2 {
3     lock(&b);
4     lock(&c);
5     ...
6     unlock(&c);
7     unlock(&b);
8 }
```

```
1 void thread2 ()
2 {
3     lock(&a);
4     lock(&c);
5     ...
6     unlock(&c);
7     unlock(&a);
8 }
```

Предотвращение deadlock-ов

- ▶ Сложный случай - все блокировки не известны заранее:

Предотвращение deadlock-ов

- ▶ Сложный случай - все блокировки не известны заранее:
 - ▶ для этого случая придумано много различных вариантов;

Предотвращение deadlock-ов

- ▶ Сложный случай - все блокировки не известны заранее:
 - ▶ для этого случая придумано много различных вариантов;
 - ▶ мы рассмотрим подход, который называется Wait-Die.

Изменим интерфейс

```
struct wdlock_ctx {
    unsigned long long timestamp;
    struct wdlock *next;
};

struct wdlock {
    ...
};

/* Grab unique "timestamp" */
void wdlock_ctx_init(struct wdlock_ctx *ctx);

/* This function may fail */
int wdlock_lock(struct wdlock *lock, struct wdlock_ctx
    ↪ *ctx);

/* Unlocks all of the locks */
void wdlock_unlock(struct wdlock_ctx *ctx);
```

Как использовать Wait-Die подход?

```
void thread(void)
{
    struct wdlck_ctx ctx;

    wdlck_ctx_init(&ctx);

    while (1) {
        ...
        if (!wdlock_lock(&lock1, &ctx)) {
            wdlck_unlock(&ctx);
            continue;
        }
        ...
        if (!wdlock_lock(&lock2, &ctx)) {
            wdlck_unlock(&ctx);
            continue;
        }
        ...
    }
    /* Acquired all required locks successfully,
       can do something. */
    wdlck_unlock(&ctx);
}
```

"Контекст"

- ▶ Wait-Die контекст состоит из:

"Контекст"

- ▶ Wait-Die контекст состоит из:
 - ▶ списка захваченных блокировок;

"Контекст"

- ▶ Wait-Die контекст состоит из:
 - ▶ списка захваченных блокировок;
 - ▶ уникального "timestamp".

"Контекст"

```
struct wdlck_ctx {
    unsigned long long timestamp;
    struct wdlck *next;
};

void wdlck_ctx_init(struct wdlck_ctx *ctx)
{
    static atomic_ullong timestamp;

    ctx->timestamp = atomic_fetch_add(&timestamp, 1) +
        ↪ 1;
    ctx->next = NULL;
}
```


Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов

Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов
 - ▶ храним в каждой блокировке timestamp из `wdlock_ctx`, который использовали при захвате блокировки;

Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов
 - ▶ храним в каждой блокировке timestamp из `wdlock_ctx`, который использовали при захвате блокировки;
 - ▶ при попытке захватить блокировку возможно несколько вариантов:

Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов
 - ▶ храним в каждой блокировке timestamp из `wdlock_ctx`, который использовали при захвате блокировки;
 - ▶ при попытке захватить блокировку возможно несколько вариантов:
 - ▶ если блокировка свободна, то пытаемся ее захватить - как обычно;

Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов
 - ▶ храним в каждой блокировке timestamp из `wdlock_ctx`, который использовали при захвате блокировки;
 - ▶ при попытке захватить блокировку возможно несколько вариантов:
 - ▶ если блокировка свободна, то пытаемся ее захватить - как обычно;
 - ▶ если блокировка занята, то нужно сравнить timestamp-ы.

Магия timestamp

- ▶ Если блокировка захвачена, то нужно сравнить наш timestamp с сохраненным в блокировке:

Магия timestamp

- ▶ Если блокировка захвачена, то нужно сравнить наш timestamp с сохраненным в блокировке:
 - ▶ если наш timestamp меньше, чем timestamp блокировки, то ждем;

Магия timestamp

- ▶ Если блокировка захвачена, то нужно сравнить наш timestamp с сохраненным в блокировке:
 - ▶ если наш timestamp меньше, чем timestamp блокировки, то ждем;
 - ▶ в противном случае не ждем, а возвращаем признак неудачи (умираем).

Корректность

- ▶ Поток ждет на блокировке, если timestamp блокировки больше, чем timestamp потока

Корректность

- ▶ Поток ждет на блокировке, если timestamp блокировки больше, чем timestamp потока
 - ▶ deadlock соответствует циклу в Wait-For графе;

Корректность

- ▶ Поток ждет на блокировке, если timestamp блокировки больше, чем timestamp потока
 - ▶ deadlock соответствует циклу в Wait-For графе;
 - ▶ при использовании Wait-Die timestamp-ы блокировок на любом пути в графе строго возрастают;

Корректность

- ▶ Поток ждет на блокировке, если timestamp блокировки больше, чем timestamp потока
 - ▶ deadlock соответствует циклу в Wait-For графе;
 - ▶ при использовании Wait-Die timestamp-ы блокировок на любом пути в графе строго возрастают;
 - ▶ следовательно, цикла в Wait-For графе быть не может.

Wait-Die граф

