

# Операционные Системы

## Синхронизация потоков

January 5, 2019

# 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
  - ▶ порядок зависит от слишком многих факторов;
  - ▶ решения планировщика, влияние других потоков, прерывания...
  - ▶ могут быть трудно воспроизводимы - не поддаются тестированию.

# 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
  - ▶ lock - метод захвата блокировки;
  - ▶ unlock - метод освобождения блокировки.

# 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

```
1      struct lock l;  
2      int counter;  
3  
4      int add_one(void)  
5      {  
6          int res;  
7  
8          lock(&l);  
9          res = ++counter;  
10         unlock(&l);  
11     }
```

# 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

```
1      struct lock {
2      };
3
4      void lock(struct lock *unused)
5      {
6          (void) unused;
7          while (1);
8      }
9
10     void unlock(struct lock *unused)
11     {
12         (void) unused;
13     }
```

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



# Альтернатива

```
1      struct lock {
2          atomic_int last;
3      };
4
5      void lock_init(struct lock *lock)
6      {
7          atomic_store(&lock->last, 0);
8      }
9
10     void lock(struct lock *lock)
11     {
12         while (atomic_load(&lock->last) == threadId());
13     }
14
15     void unlock(struct lock *lock)
16     {
17         atomic_store(&lock->last, threadId());
18     }
```

# Свойство взаимного исключения

- ▶ Для приведенного алгоритма взаимное исключение гарантируется
  - ▶ lock может вернуть управление только потоку с идентификатором, не равным `lock->last`;
  - ▶ только поток с `threadId() != lock->last` может изменить значение `lock->last`.

# 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

```
1      struct lock {
2          atomic_int flag[2];
3      };
4
5      void lock_init(struct lock *lock)
6      {
7          atomic_store(&lock->flag[0], 0);
8          atomic_store(&lock->flag[1], 0);
9      }
10
11     void lock(struct lock *lock)
12     {
13         const int me = threadId();
14         const int other = 1 - me;
15
16         atomic_store(&lock->flag[me], 1);
17         while (!atomic_load(&lock->flag[other]));
18     }
19
20     void unlock(struct lock *lock)
21     {
22         const int me = threadId();
23
24         atomic_store(&lock->flag[me], 0);
25     }
```

# Корректность

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

# Алгоритм Петерсона для 2-х потоков

```
1      struct lock {
2          atomic_int last;
3          atomic_int flag[2];
4      };
5
6      void lock(struct lock *lock)
7      {
8          const int me = threadId();
9          const int other = 1 - me;
10
11          atomic_store(&lock->flag[me], 1);
12          atomic_store(&lock->last, me);
13
14          while (atomic_load(lock->flag[other])
15                 && atomic_load(&lock->last) == me);
16      }
17
18      void unlock(struct lock *lock)
19      {
20          const int me = threadId();
21
22          atomic_store(&lock->flag[me], 0);
23      }
```

# 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:
  - ▶ `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?
  - ▶ очевидно, никак.

- ▶ 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.`

- ▶ 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

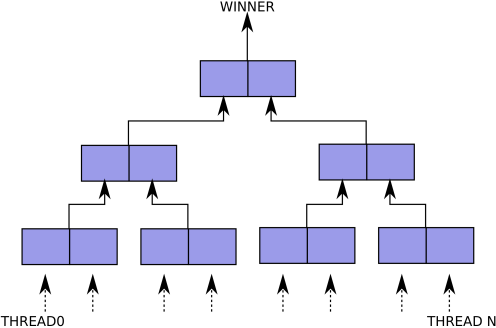
- ▶ Во втором случае ( $\text{lock} \rightarrow \text{flag}[1] == 1$ )
  - ▶ оба потока изъявили намерение войти в критическую секцию;
  - ▶ нужно показать, что хотя бы один из них рано или поздно войдет в критическую секцию (или уже там).

- ▶ 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
  - ▶ организуем турнир для N потоков;
  - ▶ потоки конкурируют друг с другом на "выбывание".

# N потоков



# Алгоритм Петерсона для N потоков

```
1      struct lock_one {
2          atomic_int last;
3          atomic_int flag[N];
4      };
5
6      int flags_clear(const struct lock_one *lock)
7      {
8          const int me = threadId();
9
10         for (int i = 0; i != N; ++i) {
11             if (i != me && atomic_load(&lock->flag[i]))
12                 return 0;
13         }
14         return 1;
15     }
16
17     void lock_one(struct lock_one *lock)
18     {
19         const int me = threadId();
20
21         atomic_store(&lock->flag[me], 1);
22         atomic_store(&lock->last, me);
23
24         while (!flags_clear(lock)
25             && atomic_load(&lock->last) == me);
26     }
27
28     void unlock_one(struct lock_one *lock)
29     {
30         const int me = threadId();
31
32         atomic_store(&lock->flag[me], 0);
33     }
```



# Алгоритм Петерсона для N потоков

```
1      struct lock {
2          struct lock_one lock[N - 1];
3      };
4
5      void lock(struct lock *lock)
6      {
7          for (int i = 0; i != N - 1; ++i)
8              lock_one(&lock->lock[i]);
9      }
10
11     void unlock(struct lock *lock)
12     {
13         for (int i = N - 2; i >= 0; --i)
14             unlock_one(&lock->lock[i]);
15     }
```

# Алгоритм Петерсона для N потоков

```
1      struct lock {
2          atomic_int level[N];
3          atomic_int last[N - 1];
4      };
5
6      void lock(struct lock *lock)
7      {
8          const int me = threadId();
9
10         for (int i = 0; i != N - 1; ++i) {
11             atomic_store(&lock->level[me], i + 1);
12             atomic_store(&lock->last[i], me);
13
14             while (!flags_clear(lock, i)
15                    && atomic_load(&lock->last[i]) == me);
16         }
17     }
18
19     void unlock(struct lock *lock)
20     {
21         const int me = threadId();
22
23         atomic_store(&lock->level[me], 0);
24     }
```

# Честность

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

# 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
  - ▶ прочитать значение в регистре;
  - ▶ преобразовать некоторым образом прочитанное значение;
  - ▶ записать преобразованное значение назад.

# Атомарный Read/Modify/Write регистр

```
1  int atomic_rmw(int *reg, int (*f)(int))
2  {
3      const int old = *reg;
4      const int new = f(old);
5
6      *reg = new;
7      return old;
8  }
```

# 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)
  - ▶ xchg;
  - ▶ lock add, lock sub, lock or, lock and, lock xor;
  - ▶ lock cmpxchg.

# 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

```
1  #define LOCKED 1
2  #define UNLOCKED 0
3
4  struct lock {
5      atomic_int locked;
6  };
7
8  void lock(struct lock *lock)
9  {
10     while (atomic_exchange(&lock->locked, LOCKED) != UNLOCKED);
11 }
12
13 void unlock(struct lock *lock)
14 {
15     atomic_store(&lock->locked, UNLOCKED);
16 }
```

# 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

```
1      struct lock {
2          atomic_uint ticket;
3          atomic_uint next;
4      };
5
6      void lock(struct lock *lock)
7      {
8          const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
9
10         while (atomic_load(&lock->next) != ticket);
11     }
12
13     void unlock(struct lock *lock)
14     {
15         atomic_fetch_add(&lock->next, 1);
16     }
```

# 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?
  - ▶ 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:
  - ▶ 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
  - ▶ если прерывания отключены, то deadlock между обработчиком прерывания и прерванным потоком не может возникнуть.

# 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
  - ▶ есть запросы, которые модифицируют данные;
  - ▶ есть запросы, которые только читают данные.

# Разделение на читателей и писателей

```
1  struct rwlock {
2      atomic_uint ticket;
3      atomic_uint write;
4      atomic_uint read;
5  };
6
7  void read_lock(struct rwlock *lock)
8  {
9      const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
10
11      while (atomic_load(&lock->read) != ticket);
12      atomic_store(&lock->read, ticket + 1);
13  }
14
15  void read_unlock(struct rwlock *lock)
16  {
17      atomic_fetch_add(&lock->write, 1);
18  }
```

# Разделение на читателей и писателей

```
1  struct rwlock {
2      atomic_uint ticket;
3      atomic_uint write;
4      atomic_uint read;
5  };
6
7  void write_lock(struct rwlock *lock)
8  {
9      const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
10
11      while (atomic_load(&lock->write) != ticket);
12  }
13
14  void write_unlock(struct rwlock *lock)
15  {
16      atomic_fetch_add(&lock->read, 1);
17      atomic_fetch_add(&lock->write, 1);
18  }
```

# 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:
  - ▶ 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?
  - ▶ 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:
  - ▶ 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
  - ▶ wait - ожидает, пока кто-нибудь не просигналит;
  - ▶ notify\_one - просигналить одному из ожидающих;
  - ▶ notify\_all - просигналить всем ожидающим.

# Переменная состояния

```
1      struct lock;  
2      void lock(struct lock *lock);  
3      void unlock(struct lock *lock);  
4  
5      struct condition;  
6      void wait(struct condition *cv, struct lock *lock);  
7      void notify_one(struct condition *cv);  
8      void notify_all(struct condition *cv);
```

# Producer

```
1      struct condition cv;
2      struct lock mtx;
3      int value;
4      bool valid_value;
5      bool done;
6
7      void produce(int x)
8      {
9          lock(&mtx);
10         while (valid_value)
11             wait(&cv, &mtx);
12         value = x;
13         valid_value = true;
14         notify_one(&cv);
15         unlock(&mtx);
16     }
17
18     void finish(void)
19     {
20         lock(&mtx);
21         done = true;
22         notify_all(&cv);
23         unlock(&mtx);
24     }
```

# Consumer

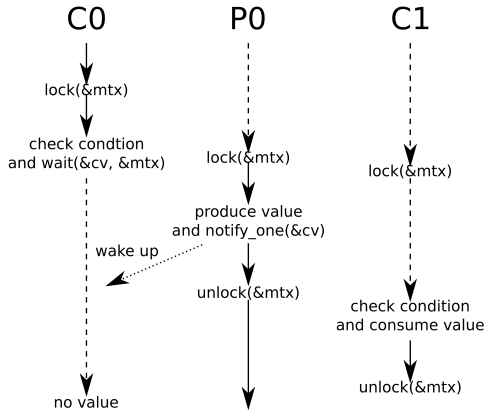
```
1      int consume(int *x)
2      {
3          int ret = 0;
4          lock(&mtx);
5
6          while (!valid_value && !done)
7              wait(&cv, &mtx);
8
9          if (valid_value) {
10             *x = value;
11             valid_value = false;
12             notify_one(&cv);
13             ret = 1;
14         }
15         unlock(&mtx);
16         return ret;
17     }
```

# 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 сигналил
  - ▶ многие реализации переменной состояния подвержены:
    - ▶ C++;
    - ▶ Java;
    - ▶ POSIX Threads...



# 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(&a);
11     unlock(&b);
12 }

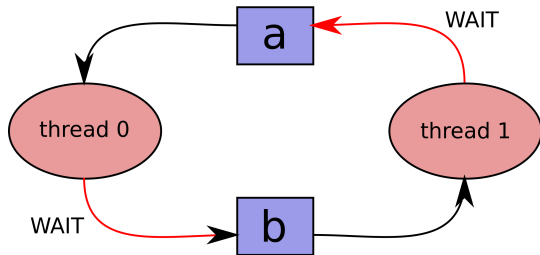
1  struct lock b;
2
3  void thread1(void)
4  {
5      lock(&b);
6      lock(&a);
7
8      /* do something else
9         ↪ */
10
11     unlock(&a);
12     unlock(&b);
13 }
```

# 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
  - ▶ появление зависит от многих факторов;
  - ▶ входные данные, решения планировщика, прерывания, производительность оборудования ...

# 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.

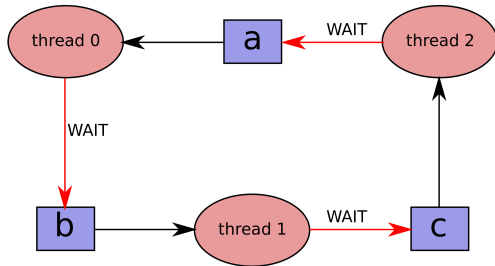
# Пример

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

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

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

# Пример





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

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

# 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

```
1      struct wdlock_ctx {
2          unsigned long timestamp;
3          struct wdlock *next;
4      };
5
6      struct wdlock {
7          ...
8      };
9
10     /* Grab unique "timestamp" */
11     void wdlock_ctx_init(struct wdlock_ctx *ctx);
12
13     /* This function may fail */
14     int wdlock_lock(struct wdlock *lock,
15                    struct wdlock_ctx *ctx);
16
17     /* Unlocks all of the locks */
18     void wdlock_unlock(struct wdlock_ctx *ctx);
```

# Как использовать Wait-Die подход?

```
1  void thread(void)
2  {
3      struct wdlck_ctx ctx;
4
5      wdlck_ctx_init(&ctx);
6
7      while (1) {
8          ...
9          if (!wdlock_lock(&lock1, &ctx)) {
10             wdlck_unlock(&ctx);
11             continue;
12         }
13         ...
14         if (!wdlock_lock(&lock2, &ctx)) {
15             wdlck_unlock(&ctx);
16             continue;
17         }
18         ...
19     }
20     /* Acquired all required locks successfully ,
21        can do something. */
22     wdlck_unlock(&ctx);
23 }
```

# "Контекст"

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

# "Контекст"

```
1      struct wdlock_ctx {
2          unsigned long timestamp;
3          struct wdlock *next;
4      };
5
6
7      void wdlock_ctx_init(struct wdlock_ctx *ctx)
8      {
9          static atomic_ullong timestamp;
10
11          ctx->timestamp = atomic_fetch_add(&timestamp, 1) + 1;
12          ctx->next = NULL;
13      }
```

# 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:
  - ▶ 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
  - ▶ 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 graph

