# Operating Systems

## Race conditions and mutual exclustion

Me

October 13, 2016

# Разделяемая память

- Несколько потоков могут обрабатывать общие данные
  - не редко несколько потоков могут решить общую задачу быстрее.
- Неатомарные чтения/обновления общих данных могут приводить к проблемам:
  - можно получить неконсистентное состояние разделяемой памяти;
  - не предсказуемый результат работы.

# Атомарность

- Операция атомарна, если она может только выполниться полностью или не выполниться вовсе:
  - либо все эффекты операции видны, либо не видны никакие;
  - наблюдатель не может увидеть только часть результата операции;
  - наблюдатель не может заметить, что операция была приостановлена.
- Не все что выглядит как одна операция является атомарным!

# Пример

```
1  int main() {
2      static volatile int i = 42;
3
4      ++i;
5      return 0;
6  }
```

```
1  main:
2    mov 0x200c4a(%rip),%eax
3    add $0x1,%eax
4    mov %eax,0x200c41(%rip)
5    xor %eax,%eax
6    retq
```

- Обычная операция инкремента транслируется в три ассемблерные инструкции:
  - чтение, обновление (инкремент) и запись;
  - что если инкремент *одной переменной* пытаются сделать сразу несколько потоков?

# Состояние гонки по данным

```
1   i: .int 42
2
3   thread0:
4     mov i, %eax
5     add $1, %eax
6     mov %eax, i
7     retq
8
9   thread1:
10    mov i, %eax
11    add $1, %eax
12    mov %eax, i
13    retq
```

- thread0 и thread1 вызываются из разных потоков;
- каждую инструкцию считаем атомарной;
- какое значение будет в $i$ после завершения?

# Хороший вариант исполнения

```
1  thread0:
2    mov i, %eax
3    add $1, %eax
4    mov %eax, i
5
6
7
8    retq
9  _
```

```
1  thread1:
2
3
4
5    mov i, %eax
6    add $1, %eax
7    mov %eax, i
8
9    retq
```

# Плохой вариант исполнения

```
1  thread0:
2    mov i, %eax
3
4    add $1, %eax
5    mov %eax, i
6
7
8    retq
9  _
```

```
1  thread1:
2
3    mov i, %eax
4
5
6    add $1, %eax
7    mov %eax, i
8
9    retq
```

# Состояние гонки

- Ситуация, когда результат работы зависит от порядка в котором выполняются атомарные инструкции называется состоянием гонки
  - состояние гонки может приводить к случайным ошибкам;
  - порядок, в котором атомарные инструкции разных потоков будут выполнены зависит от целой кучи обстоятельств:
    - от входных данных;
    - от скорости CPU;
    - от количества работы на CPU (от прерываний, других потоков и др.).
- *Пока* атомарными инструкциями для нас будут только чтение и запись переменных
  - размер переменной при этом ограничен.

# Критическая секция

- Участок кода, в котором происходит обращение к общим данным и для которого важен порядок наложения, будем называть критической секцией
  - чтобы избежать состояния гонки достаточно чтобы в критической секции не могло находиться более одного потока одновременно;
  - другими словами, если критическая секция атомарна, то проблем нет.

# Взаимное исключение 1/4

- Взаимное исключение (Mutual Exclusion, aka mutex) - примитив, который позволяет оградить критическую секцию и сделать ее атомарной
  - пара функций lock и unlock
    - вызываем lock перед тем как войти в критическую секцию;
    - вызываем unlock после выхода из критической секции.

# Взаимное исключение 2/4

- Не более одного потока может находиться между lock и unlock единовременно (и, соответственно, в критической секции)
  - если в критической секции находится другой поток, то lock не вернет управление, до тех пор пока поток в критической секции не вызовет unlock;
- если в критической секции не находится ни один поток, то из нескольких конкурирующих вызовов lock из разных потоков как минимум один окажется успешным
  - в противном случае мы могли бы просто сделать бесконечный цикл в lock - это избавляет от гонки, но создает несколько других проблем.

# Взаимное исключение 3/4

- Корректность гарантируется только для потоков использующих mutex
  - т. е. все потоки должны использовать lock/unlock, чтобы все было хорошо;
  - т. е. приходится полагаться, на то, что потоки не делают гадостей.
- Кроме того можем считать, что ни один поток не "сидит" в критической секции бесконечно долго
  - т. е. если поток вызвал lock, то он точно когда-нибудь вызовет unlock;
  - не сильное ограничение, учитывая, что мы и так считаем потоки хорошими;
  - приходится следить, чтобы поток не "упал" в критической секции.

# Взаимное исключение 4/4

- Мы не можем делать предположений
  - о скорости работы потоков
    - мы знаем, что время нахождения в критической секции конечно, но конкретное ограничение сверху нам не известно и мы не можем на него полагаться;
  - о взаимной скорости работы потоков
    - мы не можем считать, что один поток быстрее другого, или что их скорости работы;
  - другими словами, мы не можем делать предположений о том, как инструкции из разных потоков "накладываются" друг на друга.

# Реализация взаимного исключения

## Глобальный флаг 1/2

```
1  extern int claim1;
2  int claim0;
3
4  void lock0(void)
5  {
6    claim0 = true;
7    while (claim1);
8  }
9
10 void unlock0(void)
11 {
12   claim0 = false
13 }
```

```
1  extern int claim0;
2  int claim1;
3
4  void lock1(void)
5  {
6    claim1 = true;
7    while (claim0);
8  }
9
10 void unlock1(void)
11 {
12   claim1 = false;
13 }
```

# Реализация взаимного исключения

## Глобальный флаг 2/2

```
1 void lock0(void)
2 {
3   claim0 = true;
4
5   while (claim1);
6 }
```

```
1 void lock1(void)
2 {
3
4   claim1 = true;
5   while (claim0);
6 }
```

- Оба потока могут установить свои флаги до того, как другой поток успеет их проверить
  - оба потока будут ждать в цикле бесконечно - такая ситуация называется deadlock-ом.

# Реализация взаимного исключения

## Глобальный порядок 1/2

```
 1  int turn;
 2
 3  void lock0(void)
 4  {
 5    while (turn != 0);
 6  }
 7
 8  void unlock0(void)
 9  {
10    turn = 1;
11  }
```

```
 1  extern int turn;
 2
 3  void lock1(void)
 4  {
 5    while (turn != 1);
 6  }
 7
 8  void unlock1(void)
 9  {
10    turn = 0;
11  }
```

# Реализация взаимного исключения

## Глобальный порядок 2/2

```
 1  int turn;
 2
 3  void lock0(void)
 4  {
 5    while (turn != 0);
 6  }
 7
 8  void unlock0(void)
 9  {
10    turn = 1;
11  }
```

```
 1  extern int turn;
 2
 3  void lock1(void)
 4  {
 5    while (turn != 1);
 6  }
 7
 8  void unlock1(void)
 9  {
10    turn = 0;
11  }
```

- Что если lock0 никогда не будет вызван?
  - тогда и unlock0 никогда не будет вызван;
  - turn никогда не будет равен 1;
  - lock1 никогда не вернет управление.

# Реализация взаимного исключения

## Глоабльный порядок и глоабльный флаг 1/2

```
1  int claim0;
2  extern int claim1;
3  int turn;
4
5  void lock0(void)
6  {
7    claim0 = true;
8    turn = 1;
9
10   while (claim1 && turn != 0);
11 }
12
13 void unlock0(void)
14 {
15   claim0 = flase;
16 }
```

```
1  extern int claim0;
2  int claim1;
3  extern int turn;
4
5  void lock1(void)
6  {
7    claim1 = true;
8    turn = 0;
9
10   while (claim0 && turn != 1);
11 }
12
13 void unlock1(void)
14 {
15   claim1 = flase;
16 }
```

# Реализация взаимного исключения

## Глоабльный порядок и глоабльный флаг 2/3

```
1  void lock0(void)
2  {
3    claim0 = true;
4    turn = 1;
5
6    while (claim1 && turn != 0);
7  }
```

```
1  void lock1(void)
2  {
3    claim1 = true;
4    turn = 0;
5
6    while (claim0 && turn != 1);
7  }
```

- Если в lock0 мы прочитали claim1 и он оказался равен false, то
  - другой поток не вызывал lock1 вообще и не присваивал $turn = 0$, т. е. поток вызвавший lock0 - единственный кандидат;
  - другой поток не сможет войти до тех пор пока мы не вызовем unlock0, т. к. claim0 равен true и другой поток сам присвоит $turn = 0$ в lock1.

# Реализация взаимного исключения

## Глоабльный порядок и глоабльный флаг 3/3

```
1  void lock0(void)
2  {
3    claim0 = true;
4    turn = 1;
5
6    while (claim1 && turn != 0);
7  }
```

```
1  void lock1(void)
2  {
3    claim1 = true;
4    turn = 0;
5
6    while (claim0 && turn != 1);
7  }
```

- Если в lock0 мы прочитали claim1 и он оказался равен true, то
  - lock1 либо еще не присвоил в turn значение 0
    - значит он скоро сделает это, и мы будем ждать пока lock1 не выполнит $turn = 0$, после чего сможем выйти из цикла;
  - либо сделал это до того, как lock0 присвоил $turn = 1$
    - значит claim0 и claim1 оба равны true и turn равен 1, значит lock1 рано или поздно выйдет из цикла, в то время как lock0 будет ждать в цикле.

# Алгоритм Петерсона

```
1  void lock0(void)
2  {
3    turn = 1;
4    claim0 = true;
5
6    while (claim1 && turn != 0);
7  }
```

```
1  void lock1(void)
2  {
3    turn = 0;
4    claim1 = true;
5
6    while (claim0 && turn != 1);
7  }
```

- В алгоритме очень важен порядок присваиваний в claim0/claim1 и turn:
  - допустим lock0 присвоил $turn = 1$, затем в дело вступил lock1;
  - lock1 присвоил $turn = 0$ и затем $claim1 = true$, после чего прочитал значение claim0 и увидел там false - цикл в lock1 завершился;
  - lock0 продолжил там где остановился, т. е. присвоил $claim0 = true$, далее lock0 проверяет условие цикла и видит turn равным 0, т. к. lock1 записал туда последним, и тоже выходит из цикла.

# Взаимное исключение для N потоков

- ВОПРОС: как умея делать взаимное исключение для двух потоков, сделать то же самое для N потоков?

# Алгоритм Петерсона для N потоков 1/4

```
1  int claim[N];
2  int turn[N - 1];
3
4  void lock(int thread_id)
5  {
6    for (int level = 0; level < N - 1; ++level) {
7      claim[thread_id] = level + 1;
8      turn[level] = thread_id;
9
10     while (1) {
11       int found = false;
12       for (int thread = 0; !found && thread != N; ++thread) {
13         if (thread == thread_id) continue;
14         found = claim[thread] > level;
15       }
16       if (!found) break;
17       if (turn[level] != thread_id) break;
18     }
19   }
20 }
21
22 void unlock(int thread_id)
23 {
24   claim[thread_id] = 0;
25 }
```

# Алгоритм Петерсона для N потоков 2/4

- у нас есть $N - 1$ глобалных флага в массиве turn;
- и $N$ переменных намерения в массиве claim - по одной для каждого потока;
- изначально все перемнные claim равны 0.

# Алгоритм Петерсона для N потоков 3/4

- Выбор потока входящего в критическую секцию то же самое, что и выбор потоков, которые будут ждать:
  - если у нас три потока, то нужно отсеять два и оставить один;
  - отсеивание будет происходить в два этапа: на нулевом этапе отсеивается один из потоков, на первом второй;
  - переменная claim для потока хранит номер этапа (+ 1), до которого он добрался;
  - каждая переменная turn соответствует своему этапу и хранит идентификатор последнего потока добравшегося до этого этапа.

# Алгоритм Петерсона для N потоков 4/4

- Поток *может* перейти к следующему этапу если выполняется хотя бы одно из условий:
  - нет потоков на том же этапе или впереди него, т. е. поток уже победил - цикл проверяющий значения claim всех потоков отвечает за это условие;
  - поток не последний поток на данном этапе, т. е. пришел новый поток и записал в соответствующий turn свой идентификатор - проверка turn в код отвечает за это условие;

# Порядок входа в критическую секцию

- Если сразу несколько потоков конкурируют за lock, то в каком порядке они их получат?
  - нам бы хотелось, чтобы каждый из кандидатов завершил lock рано или поздно;
  - но не хотелось бы, чтобы ожидание завершения lock было не ограниченным;
  - еще больше не хотелось бы ждать, пока другие входят и выходят из критической секцию.

# Честность алгоритма Петерсона, пример

1. Изначально все равно 0:
   $claim[0] == claim[1] == claim[2] == 0$ и $turn[0] == turn[1] == 0$;
2. поток 0 вызывает lock:
   $claim[0] == 1$, $claim[1] == claim[2] == 0$ и $turn0 == turn1 == 0$;
3. поток 1 вызывает lock:
   $claim[0] == claim[1] == 1$, $claim[2] == 0$, $turn[0] == 1$ и $turn[1] == 0$;
4. поток 2 вызывает lock:
   $claim[0] == claim[1] == claim[2] == 1$, $turn[0] == 2$ и $turn[1] == 0$;
5. поток 1 не последний вошел на этап 0, а значит он может двинуться дальше:
   $claim[0] == claim[2] == 1$, $claim[1] == 2$, $turn[0] == 2$ и $turn[1] == 1$;
6. впереди потока 1 или на том же этапе других потоков нет - поток 1 выиграл, он входит в критическую секцию, а затем выходит из нее:
   $claim[0] == claim[2] == 1$, $claim[1] == 0$, $turn[0] == 2$ и $turn[1] == 1$;
7. поток 1 опять вызывает lock:
   $claim[0] == claim[1] == claim[2] == 1$, $turn[0] == 1$ и $turn[1] == 1$;
8. поток 2 не последний на этапе 0 - он может перейти к следующему этапу:
   $claim[0] == claim[1] == 1$, $claim[2] == 2$, $turn[0] == 1$ и $turn[1] == 2$;
9. впереди потока 2 или на том же этапе никоого нет - поток 2 выиграл, он входит в критическую секцию, а затем выходит из нее:
   $claim[0] == claim[1] == 1$, $claim[2] == 0$, $turn[0] == 1$ и $turn[1] == 2$;
10. поток 2 опять вызывает lock:
    $claim[0] == claim[1] == claim[2] == 1$, $turn[0] == 2$ и $turn[1] == 2$;
11. поток 1 не последний вошел на этап 0, а значит он может двинуться дальше:
    $claim[0] == claim[2] == 1$, $claim[1] == 2$, $turn[0] == 2$ и $turn[1] == 1$.

# Честность взаимного исключения 1/2

- Разделим lock на две части:
  1. вход (будем обозначать его как $D$) - эта часть всегда завершается за известное заранее конечное количество шагов;
  2. ожидание (будем обозначать как $W$) - может потребовать неограниченное количество шагов:
     - зависит от того, как долго другой поток сидит в критической секции;
     - зависит от того, в каком порядке накладываются инструкции из разных потоков;
  3. после ожидания идет критическая секция (будем обозначть ее как $CS$).

# Честность взаимного исключения 2/2

- Свойство $r$-ограниченного ожидания:
  - для некоторый потоков $A$ и $B$, из того что $D_A^j$ предшествует $D_B^k$ ($j$-ый вход потока $A$ предшествует $k$-ому входу потока $B$), следует, что $CS_A^j$ предшествует $CS_B^{k+r}$;
  - другими словами, после того как мы завершили свой $D$, то после этого не более чем $r$ потоков войдут в критическую секцию (*ВНИМАНИЕ: это очень-очень-очень грубое упрощение!*).
- Для алгоритма Петерсона для N потоков не существует такого разделения на $D$ и $W$ чтобы он удовлетворял свойству $r$-ограниченного ожидания для любого $r$
  - в частном случае двух потоков алгоритм Петерсона удовлетворяет свойству 1-ограниченного ожидания.

# Взаимное исключение с очередью

- Пусть все потоки, которые хотят войти в критическую секцию выстраиваются в очередь:
  - будем выдавать каждому пришедшему наименьший из номеров больший всех номеров в очереди;
  - обслуживать будем в порядке возрастания номеров.
- Как найти и занять нужный номер?
  - мы должны посмотреть на номера в очереди и выбрать себе подходящий;
  - но это действие звучит очень не атомарно;
  - будем вместо номера использовать пару: номер и идентификатор потока;
  - даже если выбранный номер не уникален, пара будет уникальной.

# Алгоритм пекарни

```
1  int select[N];
2  int ticket[N];
3
4  int max(void)
5  {
6    int rc = 0;
7    for (int i = 0; i != N; ++i)
8      if (ticket[i] > rc)
9        rc = ticket[i];
10   return rc;
11 }
12
13 void lock(int thread_id)
14 {
15   select[thread_id] = true;
16   ticket[thread_id] = max() + 1;
17   select[thread_id] = false;
18
19   for (int thread = 0; thread != N; ++thread) {
20     if (thread == thread_id)
21       continue;
22     while (select[thread]);
23     while (ticket[thread] && ((ticket[thread] < ticket[thread_id]) ||
24       (ticket[thread] == ticket[thread_id] && thread < thread_id)));
25   }
26 }
27
28 void unlock(int thread_id)
29 {
30   ticket[thread_id] = 0;
31 }
```

# Честность алгоритма пекарни

- вход алгоритма пекарни $D$ состоит из выбора номера для потока;
- если $D$ потока $A$ предшествует $D$ потока $B$, то возможны два случая:
  - номер выбранный потоком $B$ меньше или равен номеру выбранному потоком $A$, но это значит, что поток $B$ не видел номер выбранный $A$, т. е. $A$ его уже занулил, т. е. завершил критическую секцию;
  - номер выбранный потоком $B$ больше номера выбранного потоком $A$, а значит в критическую секцию он войдет позже.
- Другими словами алгоритм пекрани является 0-ограниченным
  - обратите внимание, что если $D_A$ не предшествует строго $D_B$ то их взаимный порядок может быть любым.

# Нижние границы

- Для реализации взаимного исключения для N потоков с использованием перемнных, которые можно атомарно только читать или писать (RW регистры), требуется не менее N таких RW регистров
  - дополнительно каждый поток может иметь внутреннее состояние, но так как оно не видимо другим потокам, то и принимать решения на его основе они не могут;
  - размер RW регистров *абсолютно не важен*, т. е. они вообще могут не иметь ограничения на хранимые данные.
- Покажем это на примере двух потоков
  - доказательство в общем виде тоже существует, но оно нам не очень интересно.

# Вспомогательные утверждения 1/2

- Состояние нашей системы описывается состоянием RW регистров и внутренним состоянием каждого потока участника
  - мы почти не можем делать предположений о внутреннем состоянии потока;
  - но мы знаем, что если поток не делал никаких действий, то его внутренее состояние не могло измениться.

# Вспомогательные утверждения 2/2

- Пусть  некоторое состояние, в котором некоторый поток $p$ не находится в критической секции и не пытается туда войти (не вызвал еще lock); рассмотрим последовательность шагов $a$ такую что:
  - в $a$ только $p$ что-то делает (планировщик отдал CPU потоку $p$);
  - в конце $a$ поток $p$ находится в критической секции.
- Тогда мы можем утверждать, что среди шагов $a$ есть как минимум одна запись потока $p$ в RW регистр.

# Доказательство 1/3

- Допустим противное, есть два потока $p_0$ и $p_1$, один общий RW регистр $x$, вся система находится в некотором начальном состояние $C_0$ и мы можем реализовать взаимное исключение.
- Рассмотрим последовательность шагов, в которой участвует только $p_0$, то из состояния $C_0$ система должна рано или поздно перейти в состояние $C$, в котором $p_0$ находится в критической секции
  - а значит где-то между $C_0$ и $C$ было состояние, из которого мы ушли сделав запись в RW регистр $x$, обозначим состояние как $C'$, последовательность шагов переводяшую систему из $C_0$ в $C'$ обозначим как $a'$;
  - последовательность шагов переводящую $C'$ в $C$ обозначим как $a$.

# Доказательство 2/3

- Если же мы дадим выполняться только $p_1$, то рано или поздно система должна прийти в состояние $C''$, в котором $p_1$ находится в критической секции
  - последовательность шагов переводящую систему из $C_0$ в $C''$ обозначим $a''$.

# Доказательство 3/3

- Мы можем "склеить" последовательности шагов $a'$, $a''$ и $a$:
  - последовательность шагов $a'$ не пишет в $x$, т. е. с точки зрения потока $p_1$ состояния $C_0$ и $C'$ не различимы;
  - значит от $C'$ мы можем отложить последовательность шагов $a''$ и перейти в состояние $C''$, в котором $p_1$ находится в критической секции;
  - внутреннее состояние потока $p_0$ в $C'$ и $C''$ идентично, так как он не делал шагов в $a''$, а значит от $C''$ мы модем отложить $a$;
  - первое действие в $a$ - запись в $x$, которая затирает все видимые результаты работы потока $p_1$, значит поток $p_0$ продолжит так как будто $p_1$ ничего не делал и войдет в критическую секцию в результате $a$.

# Q&A