

# Operating Systems

## Threads Synchronization

Me

February 27, 2016

- 1 Конкурентное исполнение. Состояние Гонки.
- 2 Взаимное исключение. Алгоритм Петерсона.
- 3 Честное взаимное исключение. Алгоритм пекарни.
- 4 Когерентность кешей и модели памяти.
- 5 Атомарность (CAS и LL/SC). Test-And-Set Lock. Queued Locks.
- 6 Атомарный снимок (seqlock).
- 7 Неблокирующая синхронизация.
- 8 Проблема ABA и безопасное освобождение памяти.
- 9 Differential Reference Counting. Hazzard Pointers. RCU.
- 10 Per-CPU и Thread Local данные.

# Конкурентное исполнение



Figure : Concurrent Execution

Конкурентное исполнение - исполняющиеся участки кода накладываются друг на друга произвольным образом

- вы не должны делать никаких предположений о порядке наложения;
- произвольный порядок ведет к произвольному доступу к общим ресурсам;

# Конкурентное исполнение

## Источники конкурентности

- много агентов исполняющих код:
  - Hyper Threading, SMP, NUMA (shared memory);
  - cluster nodes (shared storage);

# Конкурентное исполнение

## Источники конкурентности

- много агентов исполняющих код:
  - Hyper Threading, SMP, NUMA (shared memory);
  - cluster nodes (shared storage);
- прерывания - прерывают один код и запускают исполнение другого;

# Конкурентное исполнение

## Источники конкурентности

- много агентов исполняющих код:
  - Hyper Threading, SMP, NUMA (shared memory);
  - cluster nodes (shared storage);
- прерывания - прерывают один код и запускают исполнение другого;
- сигналы - userspace аналог прерываний.

# Конкурентное исполнение

## Состояние гонки

```
1 int cnt;  
2  
3 void foo(void)  
4 {  
5     ++cnt;  
6 }
```

```
1 extern int cnt;  
2  
3 void bar(void)  
4 {  
5     ++cnt;  
6 }
```

# Конкурентное исполнение

## Состояние гонки

```
1  .global cnt
2  cnt:
3  .int 0
4
5  foo:
6      mov cnt, %rax
7      inc %rax
8      mov %rax, cnt
9      ret
```

```
1  .extern cnt
2
3  bar:
4      mov cnt, %rax
5      inc %rax
6      mov %rax, cnt
7      ret
```



# Взаимное исключение

Взаимное исключение (Mutual Exclusion) - позволяет оградить критическую секцию, чтобы предотвратить конкуренции

- lock и unlock - ограничивают критическую секции в начале и конце соответственно;
- только один поток исполнения может находиться в критической секции
  - lock не вернет управление, до тех пор, пока поток в критической секции не сделает unlock;
- если в критической секции не находится поток, то из нескольких конкурирующих lock-ов, как минимум один будет успешным;

# Взаимное исключение

Мы можем считать, что

- поток выйдет из критической секции за конечное время
  - поток не зависнет и не упадет внутри критической секции;

# Взаимное исключение

Мы можем считать, что

- поток выйдет из критической секции за конечное время
  - поток не зависнет и не упадет внутри критической секции;

Мы не можем делать предположений о

- скорости работы потока
  - время нахождения в критической секции конечно, но ограничение сверху нам не известно;
- взаимной скорости работы потоков
  - мы не можем считать, что один поток быстрее/медленнее другого или что их скорости равны

# Реализация взаимного исключения

## Глобальный флаг

```
1 extern int claim1;  
2 int claim0;  
3  
4 void lock0()  
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

Следующее расписание приводит к deadlock-у:

- 1 Thread 0, line 6;
- 2 Thread 1, line 6;
- 3 Thread 0, line 7 (Thread 0 завис на этой строке);
- 4 Thread 1, line 7 (Thread 1 завис на этой строке);

# Реализация взаимного исключения

## Глобальный порядок

```
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 unlock0(void)
9  {
10     turn = 0;
11 }
```

# Реализация взаимного исключения

## Глобальный порядок

Расписание приводящее к проблемам:

- 1 Thread 1, line 5 (Thread 1 завис на этой строке);
- 2 Thread 0 - умер (решил не заходить в критическую секцию);

# Реализация взаимного исключения

## Глобальный порядок

Расписание приводящее к проблемам:

- 1 Thread 1, line 5 (Thread 1 завис на этой строке);
- 2 Thread 0 - умер (решил не заходить в критическую секцию);

Да, оно довольно короткое...



# Реализация взаимного исключения

Соберем все в кучу

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

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

# Реализация взаимного исключения

## Алгоритм Петтерсона

### *Доказательство взаимного исключения:*

- пусть сразу два потока находятся в критической секции:
  - turn принимает одно из двух значений: 0 или 1; для определенности пусть это будет 0, т. е. последним в turn записывал поток 1;
  - claim0 и claim1 оба равны true;
- к моменту проверки условия цикла потоком 1 имеем:
  - turn равен 0;
  - claim0 равен true;
  - но в этом случае поток 1 должен зависнуть в цикле до изменения turn или claim0 - противоречие;

# Реализация взаимного исключения

## Алгоритм Петтерсона

*Доказательство наличия прогресса:* пусть поток 0 пытается войти в свободную критическую секцию:

- поток 0 в 10 строке видит  $\text{claim1} == \text{true}$ :
  - поток 0 видит  $\text{turn} == 0$ , поток 0 входит в критическую секцию;
  - поток 0 видит  $\text{turn} == 1$ , возможны два случая:
    - поток 1 выполнил строку 8, поток 0 перезаписал  $\text{turn}$  - поток 1 входит в критическую секцию;
    - поток 1 собирается выполнить строку 8 - поток 0 войдет в критическую секцию, после того как поток 1 выполнит строку 8;

# Реализация взаимного исключения

## Алгоритм Петтерсона

*Доказательство наличия прогресса:* пусть поток 0 пытается войти в свободную критическую секцию:

- поток 0 в 10 строке видит  $\text{claim1} == \text{true}$ :
  - поток 0 видит  $\text{turn} == 0$ , поток 0 входит в критическую секцию;
  - поток 0 видит  $\text{turn} == 1$ , возможны два случая:
    - поток 1 выполнил строку 8, поток 0 перезаписал  $\text{turn}$  - поток 1 входит в критическую секцию;
    - поток 1 собирается выполнить строку 8 - поток 0 войдет в критическую секцию, после того как поток 1 выполнит строку 8;
- поток 0 видит  $\text{claim1} == \text{false}$  - поток 0 входит в критическую секцию;

# Реализация взаимного исключения

## Алгоритм Петтерсона для N потоков

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

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Рассмотрим пример на 3 потоках. Начальное состояние:

- $\text{flag}[3] = \{0, 0, 0\};$
- $\text{turn}[2] = \{0, 0\};$

Поток 0 пытается войти в критическую секцию ( $\text{count} = 0$ ):

- $\text{flag}[3] = \{1, 0, 0\};$
- $\text{turn}[2] = \{0, 0\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ( $\text{count} = 0$ ):

- $\text{flag}[3] = \{1, 1, 0\};$
- $\text{turn}[2] = \{1, 0\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ( $\text{count} = 0$ ):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{2, 0\};$



# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ( $\text{count} = 1$ ):

- $\text{flag}[3] = \{1, 2, 1\};$
- $\text{turn}[2] = \{2, 1\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 1 вошел в критическую секцию... И вышел из критической секции:

- $\text{flag}[3] = \{1, 0, 1\};$
- $\text{turn}[2] = \{2, 1\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ( $\text{count} = 0$ ):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{1, 1\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ( $\text{count} = 1$ ):

- $\text{flag}[3] = \{1, 1, 2\};$
- $\text{turn}[2] = \{1, 2\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 2 вошел в критическую секцию... И вышел из критической секции:

- $\text{flag}[3] = \{1, 1, 0\};$
- $\text{turn}[2] = \{1, 2\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ( $\text{count} = 0$ ):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{2, 2\};$

# Реализация взаимного исключения

## Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ( $\text{count} = 1$ ):

- $\text{flag}[3] = \{1, 2, 1\};$
- $\text{turn}[2] = \{2, 1\};$

Мы уже были в этом состоянии! А поток 0 так и не получил управление!

# Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;



# Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;

Как выбирать это число?

- посмотреть на числа всех потоков и прибавить 1 к наибольшему;
- что если два потока выбирают число одновременно?

# Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;

Как выбирать это число?

- посмотреть на числа всех потоков и прибавить 1 к наибольшему;
- что если два потока выбирают число одновременно?

Как использовать выбранное число?

- если число наименьшее среди всех потоков выбравших число, то входим в критическую секцию;

# Реализация взаимного исключения

## Алгоритм Пекарни (Л. Лэмпорт)

```
1  int choosing[N];
2  int number[N];
3
4  int max(void)
5  {
6      int rc = 0;
7
8      for (int i = 0; i != N; ++i) {
9          const int n = number[i];
10
11          if (n > rc)
12              rc = n;
13      }
14
15      return rc;
16  }
17
18  int less(int id0, int n0,
19          int id1, int n1)
20  {
21      if (n0 < n1)
22          return true;
23      if (n0 == n1 && id0 < id1)
24          return true;
25      return false;
26  }
```

```
1  void lock(int i)
2  {
3      choosing[i] = true;
4      number[i] = max() + 1;
5      choosing[i] = false;
6
7      for (int j = 0; j != N; ++j) {
8          while (choosing[j]);
9          while (less(j, number[j], i,
10                  ↪ number[i]));
11      }
12
13  void unlock(int i)
14  {
15      number[i] = 0;
16  }
```