

Межпроцессное взаимодействие

Евгений Иванович Клименков

osisp2019@gmail.com

Белорусский Государственный Университет
Информатики и Радиоэлектроники

2019

Оно же широко известно как Inter-Process Communication (IPC).
IPC подразумевает не только процессы, но и потоки.

Включает в себя:

- Синхронизацию
- Обмен данными

Data race происходит, когда в программе есть как минимум два обращения к памяти, такие что для всех справедливо следующее:

- Они затрагивают одну ячейку памяти
- Осуществляются как минимум из двух разных потоков
- Включают операцию записи
- Не являются операциями синхронизации

Data race МОЖЕТ и ОБЫЧНО приводит к Race Condition (Состояние Гонки).

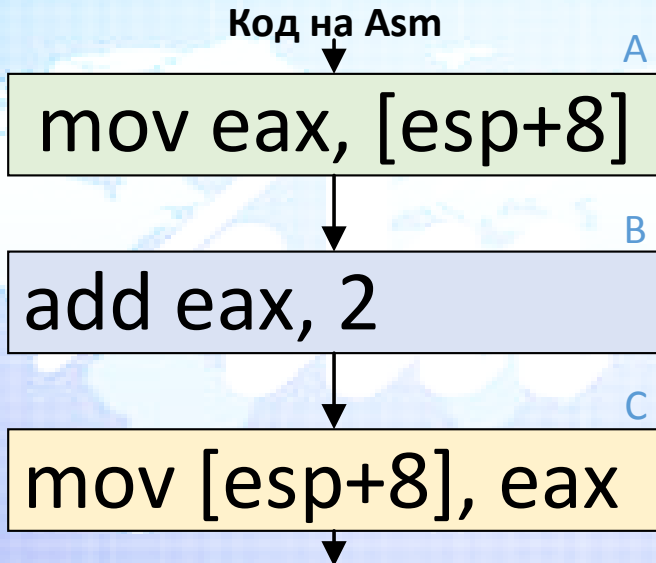
Race Condition – тип ошибок возникающих при функционировании (как следствие ошибки проектирования) параллельной системы, при которых корректность работы системы зависит от фактического порядка выполнения частей кода.

Код на С

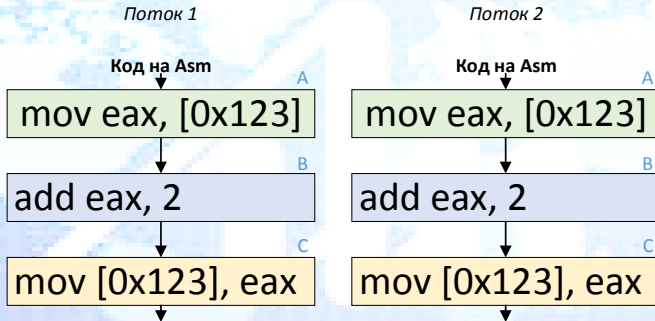
...

```
a += 2;
```

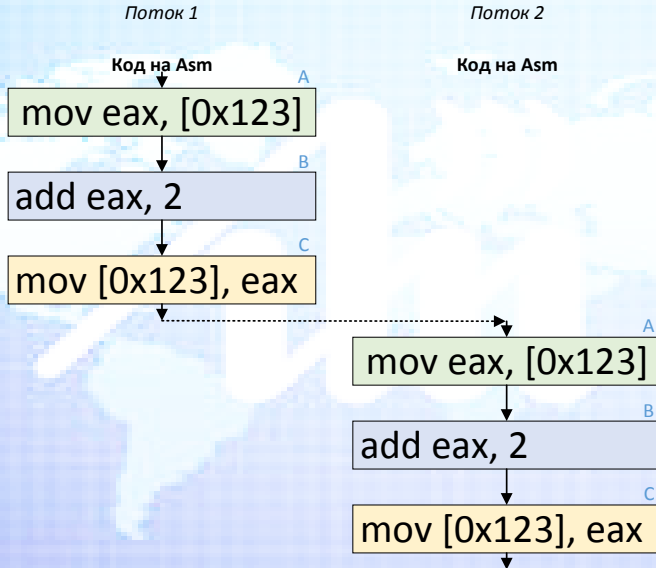
...



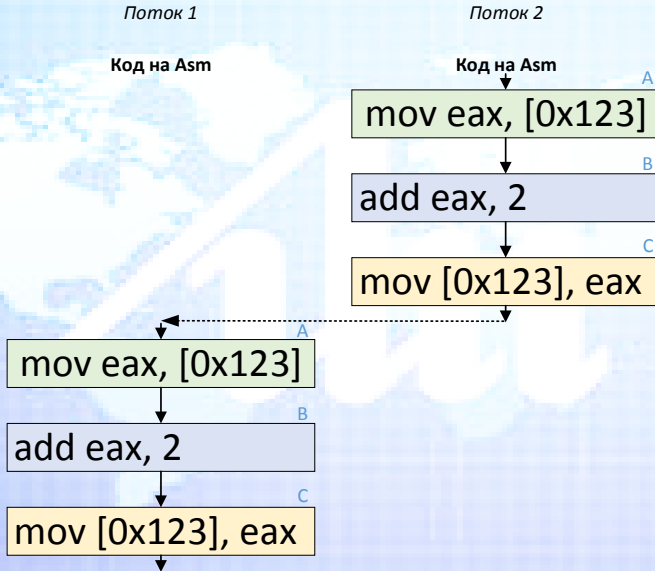
Race Condition: Пример



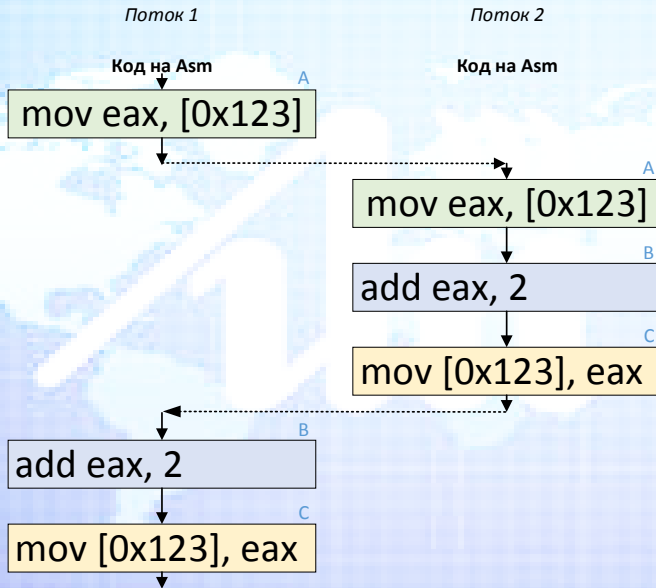
Race Condition: Пример

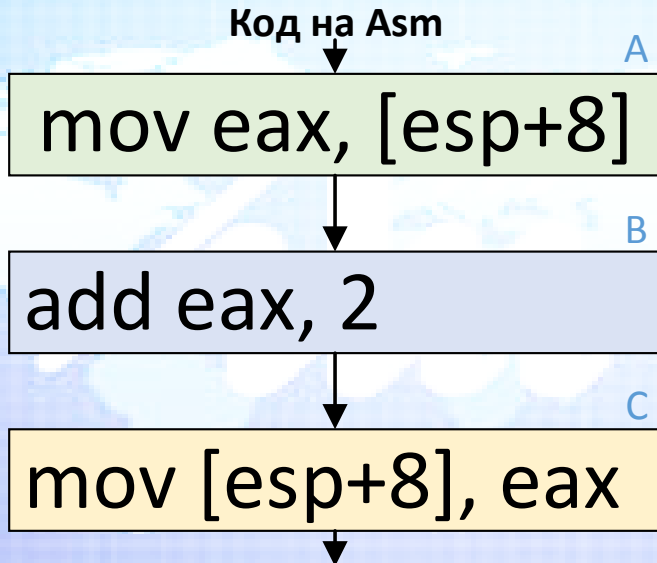


Race Condition: Пример



Race Condition: Пример





Код на Asm



```
xadd [0x123], 2
```



A

Инструкции процессора которые включают в себя одновременно чтение и запись в память называются Read-Modify-Write.

Классические примеры:

- SWAP (Swap) ==> xchg
C atomic_exchange(volatile C* obj, C desired);
- CAS (Compare-And-Swap) ==> cmpxchg
_Bool atomic_compare_exchange(volatile C* obj, C* expected, C desired);
- FAA (Fetch-And-Add) ==> xadd
C atomic_fetch_add(volatile C* obj, C arg);
- TAS (Test-And-Set) ==> bts, btc
_Bool atomic_flag_test_and_set(volatile atomic_flag* obj);

Инструкции процессора которые включают в себя одновременно чтение и запись в память называются Read-Modify-Write.

Классические примеры:

- SWAP (Swap) ==> xchg
C atomic_exchange(volatile C* obj, C desired);
- CAS (Compare-And-Swap) ==> cmpxchg
_Bool atomic_compare_exchange(volatile C* obj, C* expected, C desired);
- FAA (Fetch-And-Add) ==> xadd
C atomic_fetch_add(volatile C* obj, C arg);
- TAS (Test-And-Set) ==> bts, btc
_Bool atomic_flag_test_and_set(volatile atomic_flag* obj);

Инструкции процессора которые включают в себя одновременно чтение и запись в память называются Read-Modify-Write.

Неклассический пример:

- LL/SC (Load-Link/Store-Conditional)
- Транзакционная память

LL/SC - Это две связанные инструкции, которые любит RISC. Грубо, но элегантно.

Транзакционная память может рассматриваться как расширение LL/SC.

Пару слов о **volatile**...

volatile – это подсказка агрессивно-оптимизирующему **компилятору**, **но не процессору!!!**, о том что значение переменной может измениться непредусмотренным кодом программы образом. То есть **volatile** не имеет никакого отношения к синхронизации!

Пример:

```
for (; *tsc < 10000; )
```

где *tsc* – адрес переменной которая меняется обработчиком прерывания или другим потоком.

Что может сделать компилятор?

```
mov eax, [tsc]  
NEXT_ITERATION:  
cmp eax, 10000  
jb NEXT_ITERATION
```

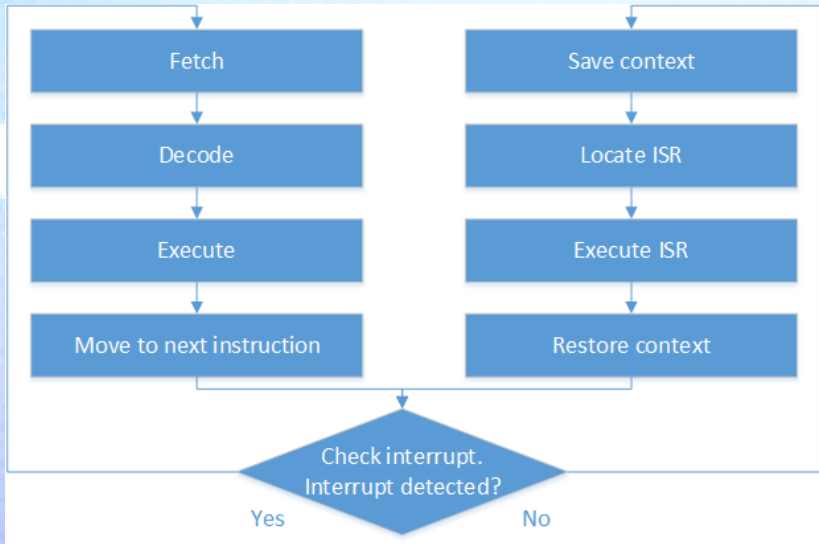
что эквивалентно:

```
mov eax, [tsc]  
cmp eax, 10000  
jae EXIT  
NEXT_ITERATION:  
jmp NEXT_ITERATION EXIT:
```

Если же `tsc` был объявлен как `volatile`, то:

```
NEXT_ITERATION:  
cmp [tsc], 10000  
jb NEXT_ITERATION
```

Concurrency



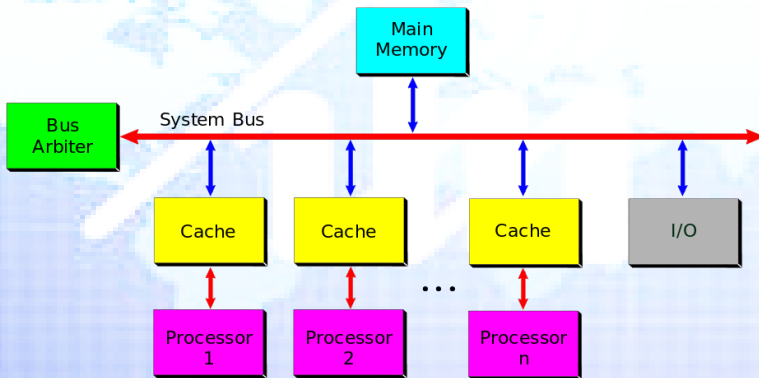
Выполнение любой инструкции процессора (практически любой) в отсутствии физического параллелизма является атомарным!

Синхронизация:

- RMW в одной инструкции
- Временное отключение прерываний: cli/sti

Однако...

SMP - Symmetric Multiprocessor System



By Ferruccio Zulian - M8an, Italy

Шина памяти (протокол когерентности кэшей) поддерживает только две операции:

- Read
- Write

и работает с блоками равными размеру строки кэша...

xadd “под капотом”:

```
xadd [0x123], 2
```

CHEAP

```
lock xadd [0x123], 2
```

EXPENSIVE

Итого, атомарными являются:

- Чтение памяти (`mov eax, [ecx]`)
- Запись в память (`mov [ecx], eax`)

но!! тогда и только тогда, когда весь доступ к памяти приходится на ОДНУ строку кэша! Для иных случаев процессор под-

держивает специальные атомарные транзакции на шине памяти.

Префикс **lock** примененный к инструкции инициирует атомарную транзакцию на время выполнения инструкции.

```
xadd [0x123], 2
```

Concurrency-SAFE
NOT safe for parallelism

```
lock xadd [0x123], 2
```

Concurrency-SAFE
Parallelism-SAFE

```
xadd [0x123], 2
```

CHEAP

```
lock xadd [0x123], 2
```

EXPENSIVE

Spinlock – метод синхронизации, при котором поток, пытающийся получить ресурс, ожидает его в цикле постоянно производя попытки, захвата ресурса.

Поток остается активным в процессе ожидания, НО не выполняет полезную работу – *busy waiting*.

Spinlock

```
unit32_t slock;
```

```
lock_spinlock:
```

```
mov eax, 1
```

```
xchg eax, [slock]
```

```
test eax, eax
```

```
jnz lock_spinlock
```

```
ret
```

```
unlock_spinlock:
```

```
mov eax, 0
```

```
xchg eax, [slock]
```

```
ret
```

Spinlock

```
alignas(4) unit32_t slock;
```

```
lock_spinlock:
```

```
mov eax, 1
```

```
try_again:
```

```
cmp [slock], 0
```

```
jne try_again
```

```
xchg eax, [slock]
```

```
test eax, eax
```

```
jnz lock_spinlock
```

```
ret
```

```
unlock_spinlock:
```

```
mov [slock], 0
```

```
ret
```

Non-Blocking Algorithm – такой алгоритм, в котором отказ или зависание одного из потоков не может привести к отказу или приостановке других потоков.

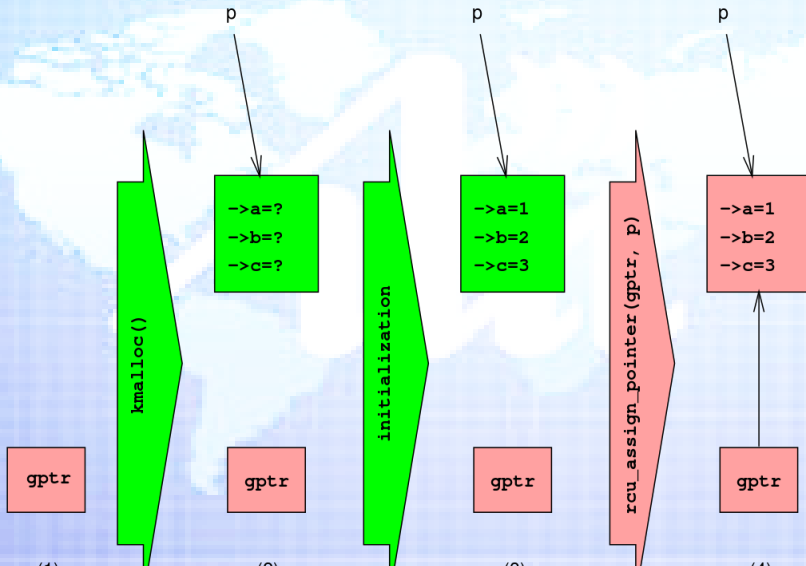
То есть в Non-Blocking Algorithms гарантируется прогресс системы и не требуется участие операционной системы.

- 1 Obstruction-freedom. Гарантируется что в любой момент времени один поток, выполняемый изолированно в течение ограниченного числа шагов, завершит свою работу.
- 2 Lock-freedom. Гарантируется что как минимум один поток в системе всегда прогрессирует. Гарантируется прогресс всей системы.
- 3 Wait-freedom. Гарантируется что любая операция выполняемая потоком завершится в течении ограниченного числа шагов. Гарантируется прогресс каждого потока в системе.

- 1 Obstruction-freedom. Гарантируется что в любой момент времени один поток, выполняемый изолированно в течение ограниченного числа шагов, завершит свою работу.
- 2 Lock-freedom. Гарантируется что как минимум один поток в системе всегда прогрессирует. Гарантируется прогресс всей системы.
- 3 Wait-freedom. Гарантируется что любая операция выполняемая потоком завершится в течении ограниченного числа шагов. Гарантируется прогресс каждого потока в системе.

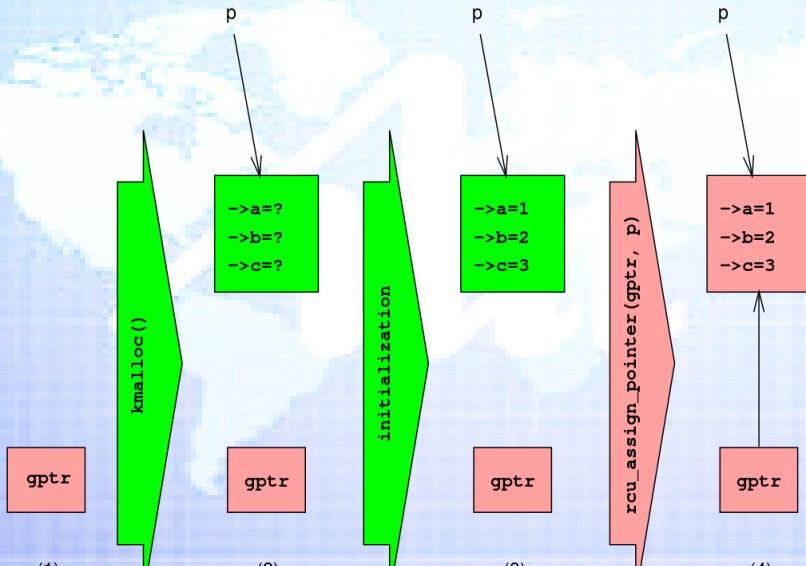
Read-Copy-Update

RCU используется при интенсивном чтении и спорадической записи.

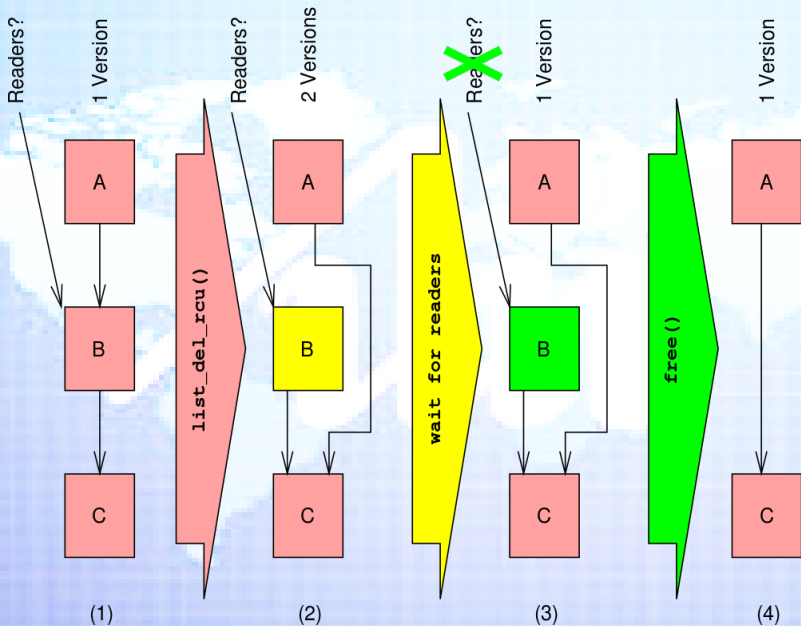


Read-Copy-Update

RCU используется при интенсивном чтении и спорадической записи.



Read-Copy-Update



Pessimistic вариант RCU. Обычно реализуется на основе семафора

Значение из памяти читается два раза.
Оба раза мы получаем одно и тоже значение.
Однако, это не означает, что защищаемая структура данных не изменилась. :-)

ABA problem

```
void Push(Obj* obj_ptr) {  
    while(1) {  
        Obj* next_ptr = top_ptr;  
        obj_ptr->next = next_ptr;  
        // If the top node is still next, then assume no one has changed the stack.  
        // (That statement is not always true because of the ABA problem)  
        // Atomically replace top with obj.  
        if (top_ptr.compare_exchange_weak(next_ptr, obj_ptr)) {  
            return;  
        }  
        // The stack has changed, start over.  
    }  
}
```

Все проблемы программирования связанные с многопоточностью можно разделить на следующие классы:

- 1 Abnormal termination.
- 2 Starvation.
- 3 Race condition.
- 4 Deadlock.

Abnormal termination

На уровне потоков: Exit Gracefully or Fail Fast!

На уровне процессов: Be Prepared, Check Exit Codes and Have Rollback Plan.

Основная проблема: разрушение гарантий целостности данных.

Предпосылки:

- 1 Задачи выполняются с разным приоритетом.
- 2 Высокий уровень конкуренции.
- 3 Coarse-grained locking.
- 4 Отсутствие контроля за параметрами критической секции.
- 5 Отсутствие справедливых политик

Решения:

- 1 Используйте повышенный приоритет выполнения только там где это действительно нужно
- 2 Локализируйте данные
- 3 Fine-grained locking.
- 4 Сделайте код готовым к работе с неожиданными данными
- 5 Следите за политиками
- 6 Изолируйте ваши блокировки

Причина: конкурентное обращение с разделяемым ресурсом (память, устройство, файл и т.д.).

Варианты:

- 1 Нарушение атомарности
- 2 Нарушение порядка

Пример:

```
1  Thread 1::
2  if (thd->proc_info) {
3      ...
4      fputs(thd->proc_info, ...);
5      ...
6  }
7
8  Thread 2::
9  thd->proc_info = NULL;
```

Решение:

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6      ...
7      fputs(thd->proc_info, ...);
8      ...
9  }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```

Другое Решение:

```
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1::
4 int local_proc_info = thd->proc_info;
5
6 if (local_proc_info) {
7     ...
8     fputs(local_proc_info, ...);
9     ...
10 }
11
12 Thread 2::
13 thd->proc_info = NULL;
```

Пример:

```
1  Thread 1::  
2  void init() {  
3      ...  
4      mThread = PR_CreateThread(mMain, ...);  
5      ...  
6  }  
7  
8  Thread 2::  
9  void mMain(...) {  
10     ...  
11     mState = mThread->State;  
12     ...  
13 }
```

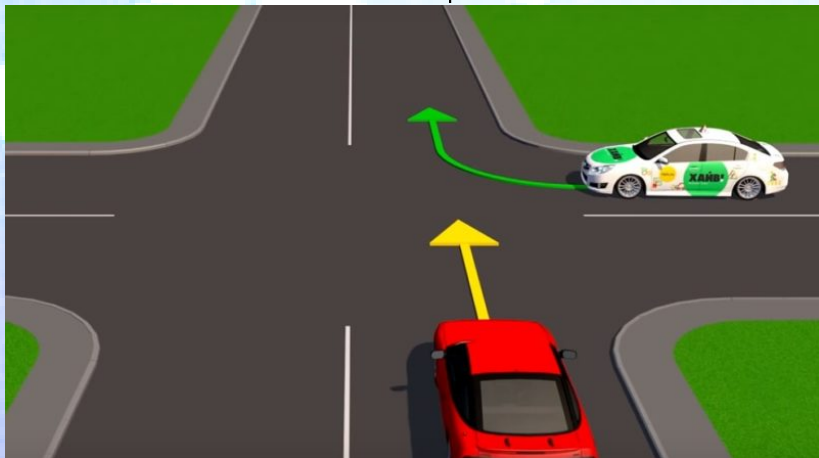

Решение:

```
1 HANDLE event = ::CreateEvent(NULL, FALSE, FALSE, NULL);
2
3 Thread 1::
4 void init() {
5     ...
6     mThread = PR_CreateThread(mMain, ...);
7     SetEvent(event);
8     ...
9 }
10
11 Thread 2::
12 void mMain(...) {
13     ...
14     WaitForSingleObject(event, ...);
15     mState = mThread->State;
16     ...
17 }
```

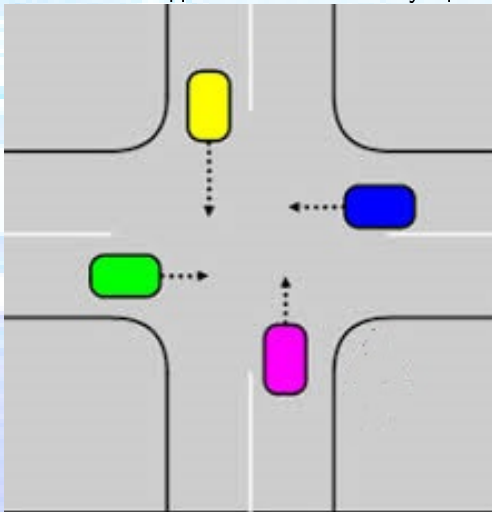
Другое Решение:

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3 int mtInit
4     = 0;
5 Thread 1::
6 void init() {
7     ...
8     mThread = PR_CreateThread(mMain, ...);
9
10    // signal that the thread has been created...
11    pthread_mutex_lock(&mtLock);
12    mtInit = 1;
13    pthread_cond_signal(&mtCond);
14    pthread_mutex_unlock(&mtLock);
15    ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20     ...
21     // wait for the thread to be initialized...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

Помеха справа:



Ok... Но что делать в такой ситуации?



“Всё, что может пойти не так,
пойдет не так”
Murphy's law

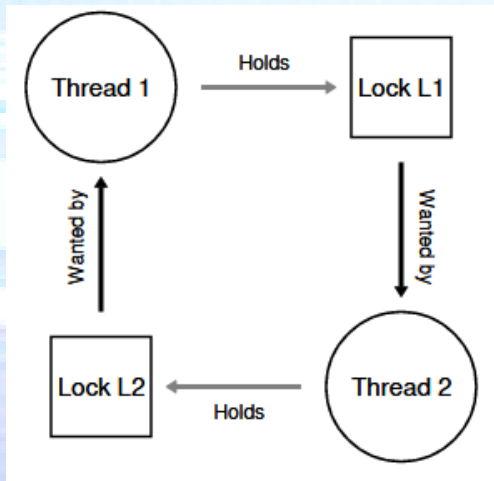
Deadlock



memecenter.com

Deadlock – ситуация в многозадачной среде, при которой несколько задач находятся в состоянии ожидания ресурсов, занятых друг другом, так что ни одна из них не может продолжать свое выполнение.

Deadlock

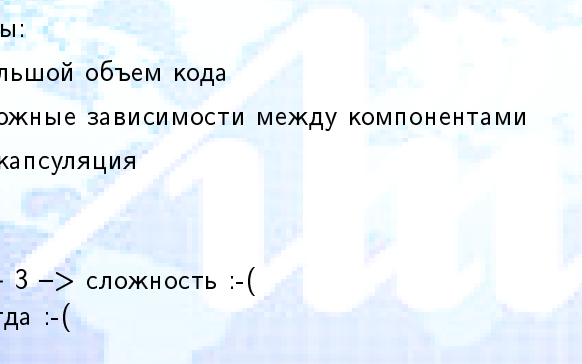


Условия возникновения (Coffman conditions):

- 1 Взаимное исключение с пересечением двух критических секций
- 2 Ожидание с удержанием
- 3 Отсутствие принудительного вытеснения ресурса
- 4 Наличие цикла в графе ожидания

Причины:

- 1 Большой объем кода
- 2 Сложные зависимости между компонентами
- 3 Инкапсуляция

$1 + 2 + 3 \rightarrow$ сложность :-(
как всегда :-(


Обнаружение:

- 1 Манифестируется в виде зависания программы или части программы
- 2 Источник ищется путем анализа call stack-ов потоков в режиме отладки
- 3 Анализ исходного кода

Анализ call stack-ов

Thread 1:

Mutex.Get()

MakeProgress() + 0x13

....

TakeAction() + 0x45

...

ThreadMain() + 0x44

Thread 2:

Mutex.Get()

TakeAction() + 0x28

...

MakeProgress() + 0x89

...

ThreadMain() + 0x50

Анализ исходного кода

```
ProgressBar pb;  
  
void TakeAction()          void MakeProgress()  
{                          {  
    ...                    ...  
    SomeAction();          EnforceAction();  
    ...                    ...  
    pb.lock();             pb.lock();  
    ...                    ...  
}                          }  
  
void SomeAction()          void EnforceAction()  
{                          {  
    ...                    ...  
    MakeProgress();        TakeAction();  
    ...                    ...  
}                          }
```

Предотвращение Deadlock-ов

- 1 Упорядочивание захватов блокировок
 - Явное определение весов блокировок
- 2 Все-и-Сразу
- 3 Оптимистичный подход
- 4 Развязывание критических секций

```
1  pthread_mutex_lock(prevention);    // begin lock acquisition
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

Оптимистичный подход:

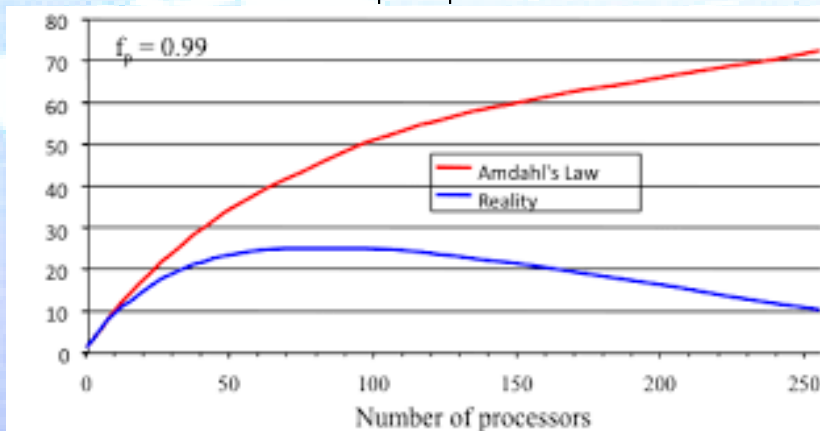
```
1  top:
2      pthread_mutex_lock(L1);
3      if (pthread_mutex_trylock(L2) != 0) {
4          pthread_mutex_unlock(L1);
5          goto top;
6      }
```

Speedup:

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

И еще немного о параллелизме

Speedup:



Speedup:

$$S_n = s + (1 - s)n$$