Файберизуем, не привлекая внимания санитаров

АНТОН МАЛАХОВ

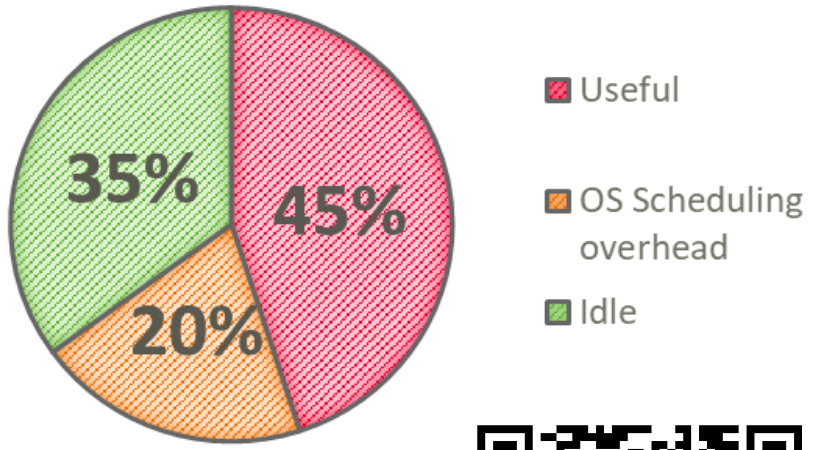
ДМИТРИЙ ОЛЬШАНСКИЙ

АНТОН ПОТАПОВ

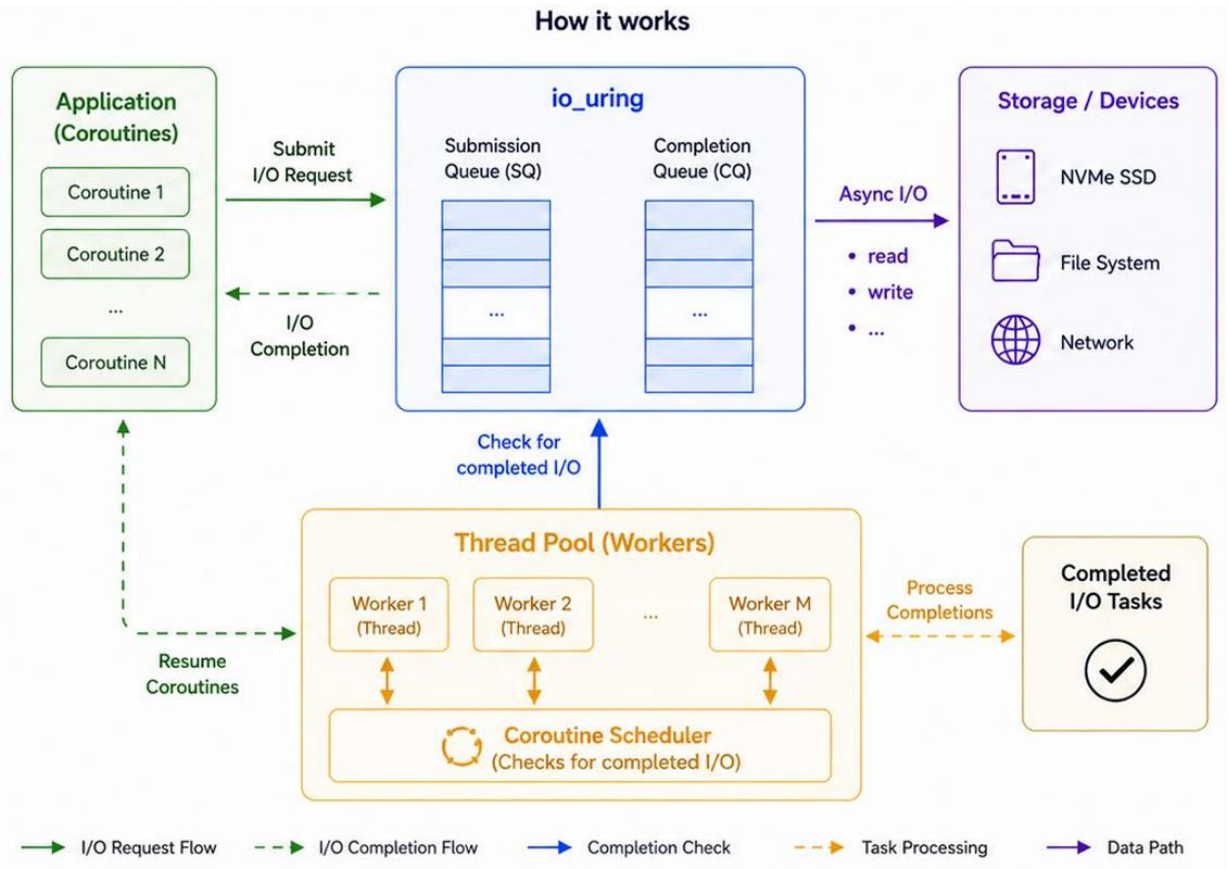
C++ RUSSIA 2026

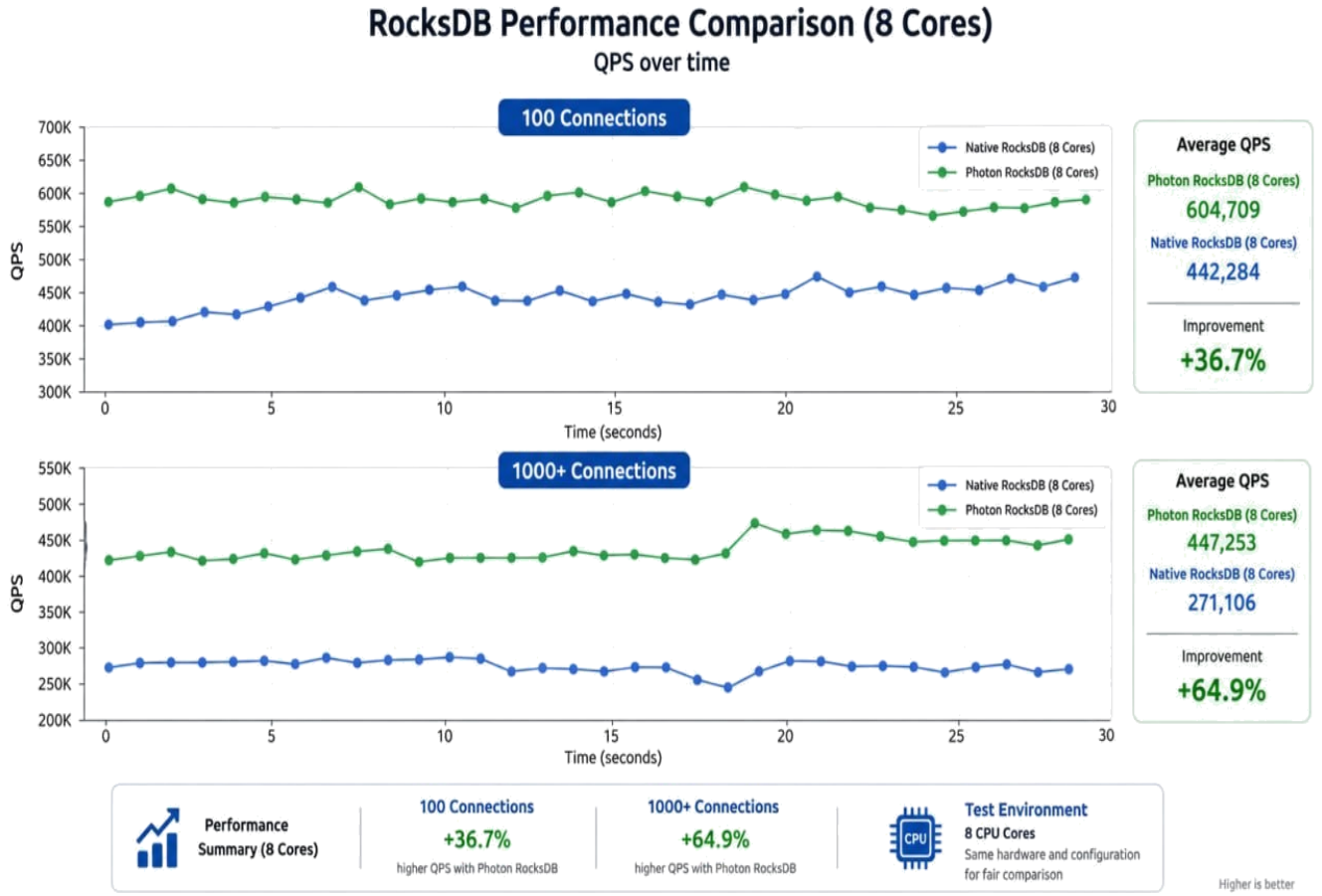
Опять файберы!

- Задача – ускорить приложение
  - Опять MySQL!
- Многопоточная композабельность!
  - Global Thread Pool с количеством рабочих потоков не более количества CPU
  - Устраняем oversubscription и оверхеды ОС – идём по пути SPDK, DPDK
  - «**Как впихнуть невпихуемое, или Давайте параллелить дружно!»** C++ Russia 2024
- Stackfull корутины со стилингом
  - Boost.Fibers, а не Folly или C++ корутины
  - «**Файберы: чудо-технология или нано-костыль, или Как ускорить легаси-код, (почти) не меняя его**» C++ Russia 2025
- Асинхронность!
  - Нельзя блокировать рабочий поток – его ждут много файберов
  - Начинали с Boost.Asio
- Придётся переписать всё?!



Такое уже было!



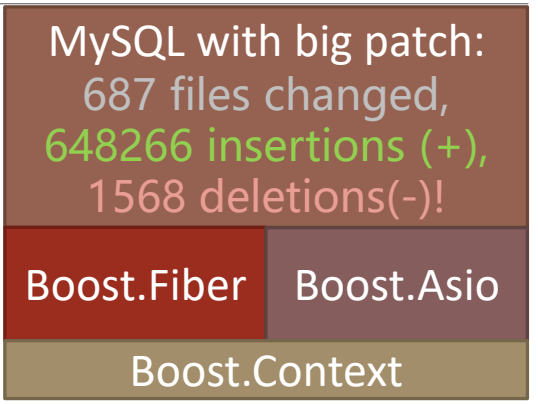


RocksDB: 200 lines of code to rewrite the 600'000 lines RocksDB into a coroutine program #11017

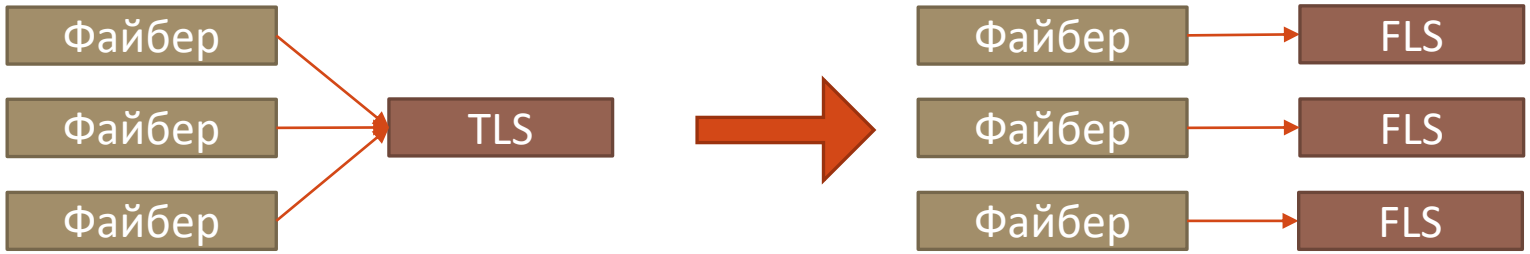
Наше решение

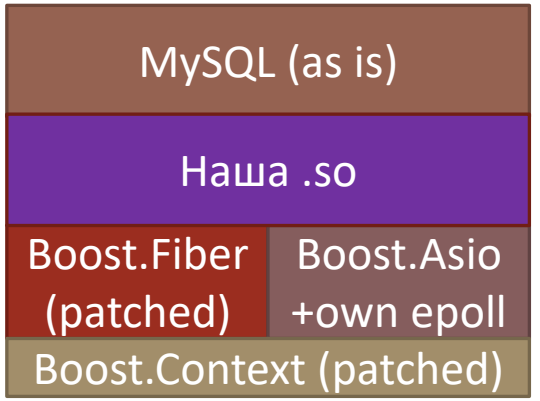
- Задача – ускорить MySQL не меняя MySQL
  - Но мы не компилятор пишем!
  - Мы - команда Parallel Runtimes

MySQL with big patch:
687 files changed,
648266 insertions (+),
1568 deletions(-)!



- Решение – вынести всё в свою библиотеку
  - Заменять стандартные символы во время загрузки
  - Заменять блокирующие вызовы асинхронными
  - Заменять потоки и синхронизацию на файберы
  - Прозрачно заменять TLS на FLS во время исполнения





Модели TLS

Модель	Типичное применение	Доступ
Local Exec (LE)	Статически линкуемые исполняемые файлы	**TP** + фиксированное_смещение
Initial Exec (IE)	Общие библиотеки, загружаемые при старте	**TP** + GOT[пересчитанное_смещение]
Local Dynamic (LD)	Динамически загружаемые модули с известным размером TLS	**TP** + DTV[индекс] + смещение_символа
Global Dynamic (GD)	Полностью динамические плагины (dlopen/dlclose)	**TP** + DTV[индекс] + смещение_символа

Достаточно поменять один регистр при переключении контекста!

Архитектура	Регистр **TP**	Метод доступа
x86_64 (Linux)	базовый FS	RDFSBASE / WRFSBASE (расширение FSGSBASE)
x86_64 (Windows)	базовый GS	TEB через сегмент GS
ARM64	TPIDR_EL0	инструкции MRS / MSR
RISC-V	CSR tp	чтение/запись через CSR

Наше решение: подмены libc+pthread

pthread mutex_t: pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock

pthread_cond_t: pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_clockwait

pthread rwlock_t: pthread_rwlock_init, pthread_rwlock_destroy, pthread_rwlock_rdlock, pthread_rwlock_tryrdlock, pthread_rwlock_wrlock, pthread_rwlock_trywrlock, pthread_rwlock_unlock

pthread_t: pthread_create, pthread_exit, pthread_join, pthread_detach, pthread_self, pthread_equal, pthread_cancel, pthread_kill, pthread_setaffinity_np, pthread_getaffinity_np, setpriority

Sleep functions: sched_yield, nanosleep, usleep, sleep

IO functions: recv, send, accept, accept4, poll, ppoll, pread, pread64, pwrite, pwrite64, fsync, fdatasync, select, read, write, AIO*, epoll... syscall(FUTEX)

Как бороться со spinwait-ами

Варианты

- Не бороться, хороший spinwait зовёт yield/sleep → пока так
- Compile-time → так и было
- Binary patching → в планах
- Preemption на сигналах → в планах

А теперь – в детали!

АНТОН ПОТАПОВ

Пишем POSIX Threads (fibers) на C++

«Подмена» перегрузкой и дефайнами,
или

«Поехали, потом заведешь!» (C)

Пишем POSIX Threads (fibers) на C++

```
//Что-то похожее на :
struct posix_like_mutex_t { boost::fibers::mutex mtx; };
#define pthread_mutex_t  posix_like_mutex_t

int pthread_mutex_init(posix_like_mutex_t *, const pthread_mutexattr_t *)
{ return 0;}
int pthread_mutex_destroy(posix_like_mutex_t *)
{ return 0;}
int pthread_mutex_lock   (posix_like_mutex_t * p)
{ p->mtx.lock(); return 0; }
```

Пишем POSIX Threads (fibers) на C++

Подмена символов перегрузкой и дефайнами - кто хуже?

- Отображение pthread → boost::fiber
  - Профит! ☺

- I/O = boost.asio + `boost::fibers::condition_variable_any`
- Патч меньше чем PhotonLibOS в RocksDB

Пишем POSIX Threads (fibers) на C++

(ELF) Loader symbol interposition

- Таблица символов
- «Ленивое» заполнение
- Сначала в исполняемом файле,
- потом «Первый подходящий» из библиотек
- поэтому работает LD_PRELOAD

Пишем POSIX Threads (fibers) на C++

(ELF) Loader **symbol** interposition,

Структуры – не символы
Подменить структуры в бинарном коде нельзя ☹

Что делать ? ☺

POSIX Thread (fibers) mutex на C++

```
pthread_mutex_t my_mutex;
int status = pthread_mutex_init(&my_mutex, NULL);
pthread_mutex_lock(&my_mutex);
pthread_mutex_unlock(&my_mutex);
pthread_mutex_destroy(&my_mutex);
```

Пишем POSIX Threads (fibers) на C++

Повторное использование pthread_*** структур ☺, на примере pthread_mutex_t

«In most POSIX-compliant systems, including Linux (glibc), pthread_mutex_t is defined as an *opaque union*. This design ensures binary compatibility across different versions while allowing the internal structure to change without breaking user applications» Stack Overflow

“Use the source, Luke” (C)

Повторное использование pthread_mutex_t

```
typedef union {
    struct __pthread_mutex_s {
        int __lock;              // Current lock state (0 for unlocked)
        unsigned int __count;    // Used for recursive locks
        int __owner;             // Thread ID (TID) of the current owner
        int __kind;              // Type: Normal, Recursive, Error-Checking
        unsigned int __nusers;   // Number of users/waiters
        // ... implementation-specific extension fields (e.g., futex wait queues)
    } __data;
    char __size[__SIZEOF_PTHREAD_MUTEX_T]; // Pad to maintain consistent size
    long int __align;                      // Ensures proper memory alignment
} pthread_mutex_t;
```

Повторное использование pthread_mutex_t

```
typedef union {
    struct __pthread_mutex_s {
        int __lock;              // Current lock state (0 for unlocked)
        unsigned int __count;    // Used for recursive locks
        int __owner;             // Thread ID (TID) of the current owner
        int __kind;              // Type: Normal, Recursive, Error-Checking
        unsigned int __nusers;   // Number of users/waiters
        // ... implementation-specific extension fields (e.g., futex wait queues)
    } __data;
    char __size[__SIZEOF_PTHREAD_MUTEX_T]; // Pad to maintain consistent size
    long int __align;                      // Ensures proper memory alignment
} pthread_mutex_t;
```

Повторное использование pthread_mutex_t

“Незаметно” подменим :

```
struct alignas(pthread_mutex_t) pthread_mutex_1{
    const constexpr size_t size = offsetof(__pthread_mutex_s, __kind);
    std::array<std::byte_t, size>   payload;
    std::atomic<int>                __kind; // init_state
};
```

Повторное использование pthread_mutex_t

И еще разок :

```
struct alignas(pthread_mutex_t)  pthread_mutex_t_1 {
    union {
        boost::fibers::mutex mtx;
        std::byte __pad[offsetof(__pthread_mutex_s, __kind)];
    };
    std::atomic<int> init_state; // used to be __kind
};
```

Повторное использование pthread_mutex_t

```
enum { initializing = 16, initialized = 32, destroyed = 64, };
```

```
extern "C" EXPORT int pthread_mutex_init(pthread_mutex_t *__restrict __mutex, const
pthread_mutexattr_t * )
{
    auto & location = * (reinterpret_cast<pthread_mutex_t_1*>(__mutex));
    std::construct_at(& location.mtx);
    location.init_state.store(initialized, std::memory_order_release);

    return 0;
}
```

Повторное использование pthread_mutex_t

```
enum { initializing = 16, initialized = 32, destroyed = 64, };
```

```
extern "C" EXPORT int pthread_mutex_destroy(pthread_mutex_t *__mutex)
 {
    auto & location = * (reinterpret_cast<pthread_mutex_t_1*>(__mutex));
    if (initialized == location.init_state.load(std::memory_order_acquire)){
        std::destroy_at(& location.mtx);
    }

    return 0;
}
```

POSIX Thread (fibers) mutex на C++

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&my_mutex);
pthread_mutex_unlock(&my_mutex);
pthread_mutex_destroy(&my_mutex);
```

POSIX Thread (fibers) mutex на C++

```
extern "C" EXPORT int pthread_mutex_lock(pthread_mutex_t *__mutex) {
    auto & location = * (reinterpret_cast<pthread_mutex_t_1*>(__mutex));
    initialize_once(location);
    location.mtx.lock();

    return 0;
}
```

POSIX Thread (fibers) mutex на C++

```
void initialize_once(auto& location){
    auto state = location.init_state.load(std::memory_order_relaxed);
    assert(destroyed != state);
    if (initialized == state) [[likely]] {
        //кажется, неизбежно
        std::atomic_thread_fence(std::memory_order_acquire);
        return;
    } else {
        initialize_out_of_line(location);
    }
}
```

POSIX Thread (fibers) mutex на C++

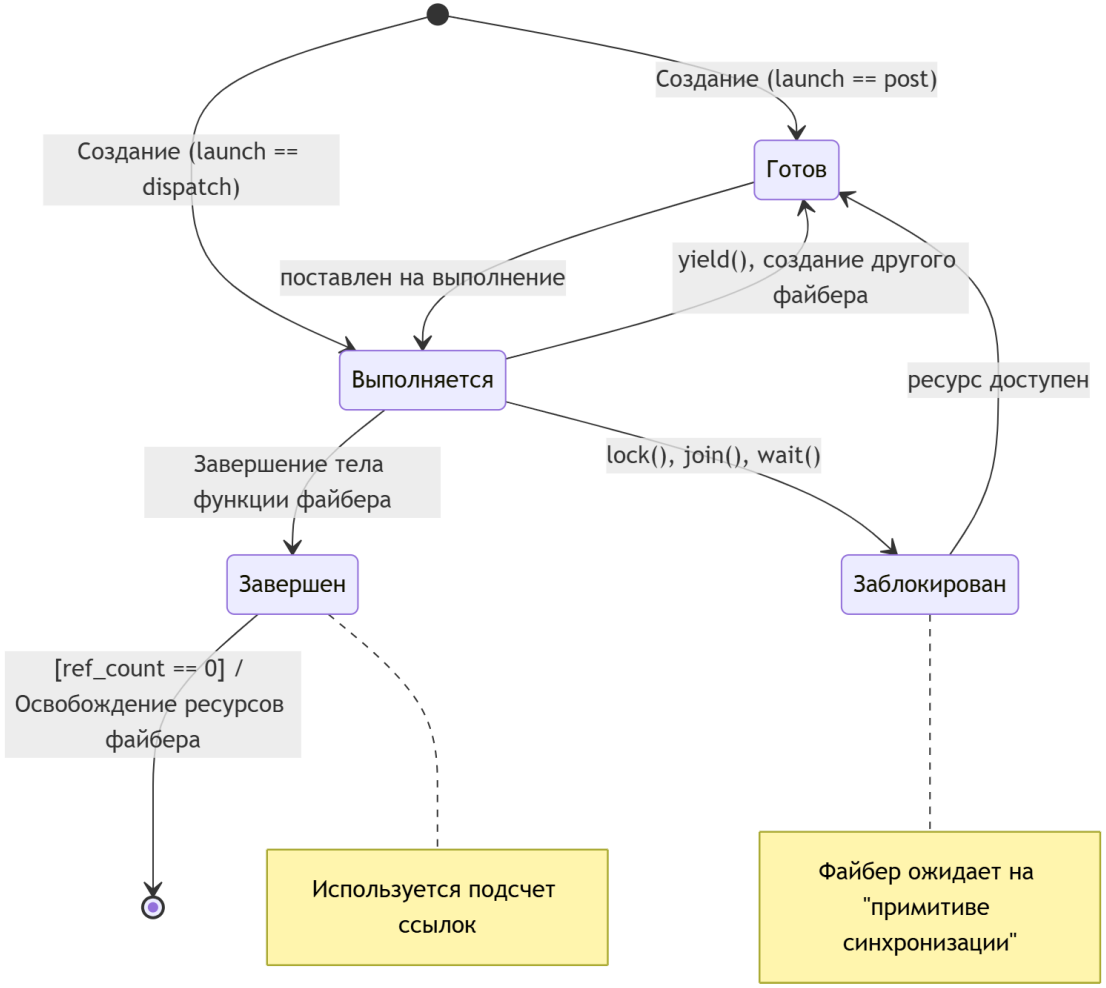
```
void initialize_out_of_line(auto& location){
    auto& init_state = location.init_state;
    do {
        auto state = init_state.load(std::memory_order_relaxed);
        if (initialized == state){
            std::atomic_thread_fence(std::memory_order_acquire);
            return;
        } else if (initializing == state) {
            pause(); //this_thread::yield();
        } else if (init_state.compare_exchange_weak(state, initializing)) {
            std::construct_at(& location.mtx);
            init_state.store(initialized, std::memory_order_release);
        }
    } while (true);
}
```

планировщик boost.fiber

Кооперативная многозадачность

- точки переключения, файбер :
  - Явно вызывает this_fiber::yield().
  - Блокируется на мьютексе, канале (channel) или условной переменной (condition_variable).
  - (опционально) Стартует новый файбер
  - из неожиданного: вызывает mutex::unlock (или как поймать себя за хвост в планровщике)

Диаграмма состояний файбера boost.fiber



планировщик boost.fiber

Очереди (Ready/Waiting Queues):

- Ready Queue: файберы, готовые к исполнению.
- Waiting Queue: файберы, ожидающие внешнего события или таймера. Планировщик периодически проверяет их статус (например, истечение времени ожидания).
- у каждого примитива синхронизации своя очередь
- boost.intrusive

Свой планировщик на поток OS

- thread_local переменная

планировщик boost.fiber

Планировщик = сумма из:

- Fiber Manager, диспетчер
- Алгоритм выбора следующего Файбера

планировщик boost.fiber

Алгоритм выбора следующего Файбера

- точка кастомизации, интерфейс algo::algorithm
- «мозг» планировщика
- стратегия, которая говорит диспетчеру, что именно делать в каждый момент времени

Алгоритм выбора следующего Файбера

Алгоритм выбора следующего Файбера

- Управление очередью готовых задач (Ready Queue)
  - Где и как лежат готовые файберы.
  - Кто идет следующим (Round Robin, Priority, Work Stealing).
- Реализация стратегии ожидания (Idle Strategy)
  - Что делать, если делать нечего
  - Как эффективно «усыпить» поток ОС и как его быстро «разбудить».
- Координация потоков (Multi-threading)
  - Если «рабочих» потоков (OS) несколько – как распределить файберы между ними

Алгоритм выбора следующего Файбера

```
namespace boost::fibers::algo {
struct algorithm {
    virtual ~algorithm();
    virtual void awakened( context *) noexcept = 0;
    virtual context * pick_next() noexcept = 0;
    virtual bool has_ready_fibers() const noexcept = 0;
    //for slides only :)
    using time_point = std::chrono::steady_clock::time_point;
    virtual void suspend_until(time_point const&) noexcept = 0;
    virtual void notify() noexcept = 0;
};
}
```

Алгоритм выбора следующего Файбера

Алгоритм выбора следующего Файбера

- Управление очередью готовых задач (Ready Queue)
  - Где и как лежат готовые файберы - awakened(context * ctx):
    Алгоритм должен сохранить указатель на файбер, который стал готов к выполнению. Здесь вы реализуете политику: добавить его в конец (FIFO), в начало (LIFO) или в приоритетную очередь.
  - Кто идет следующим : pick_next():
    Диспетчер вызывает этот метод, когда текущий файбер приостановился. Алгоритм должен вернуть самый «подходящий» файбер из своего хранилища. Если вы пишете Priority Scheduler, здесь вы выбираете файбер с наивысшим приоритетом.

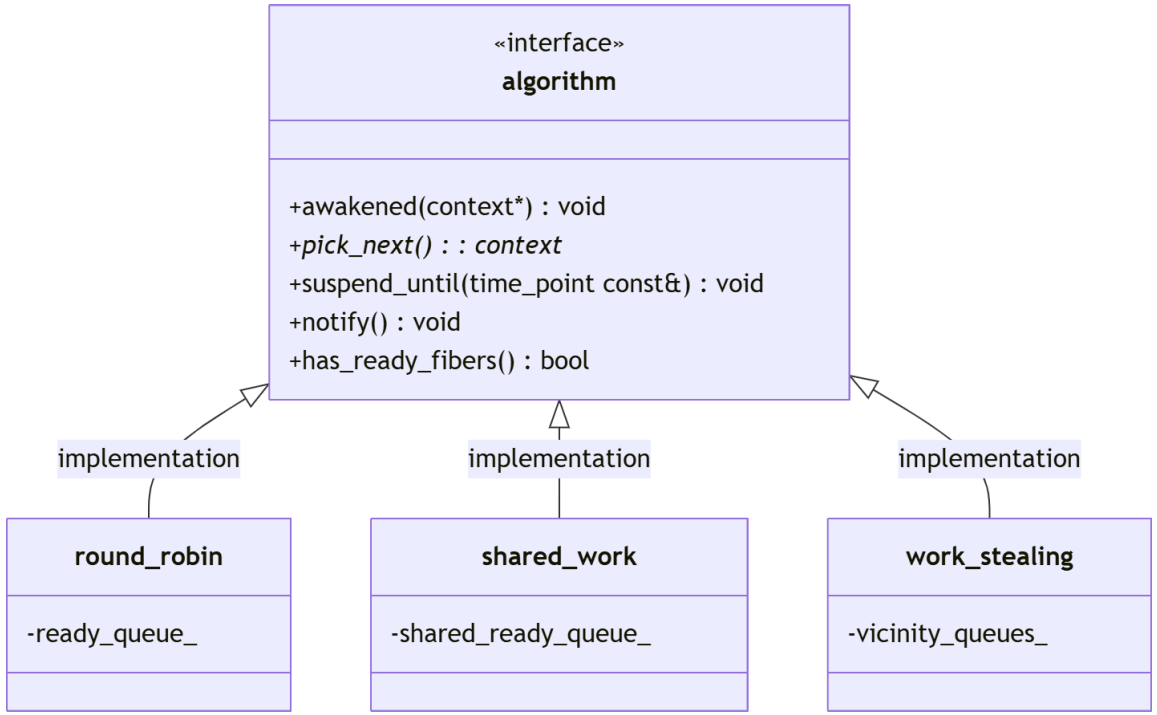
Алгоритм выбора следующего Файбера

Алгоритм выбора следующего Файбера

- Реализация стратегии ожидания (Idle Strategy)
  - Как эффективно «усыпить» поток ОС и как его быстро «разбудить».
  - *has_ready_fibers()*: Быстрый ответ диспетчеру, есть ли вообще смысл что-то планировать.
  - **suspend_until(time_point)**:
    Самая важная задача для экономии ресурсов. Алгоритм должен остановить выполнение текущего потока ОС до момента time_point (когда сработает таймер файбера)
    ИЛИ до того момента, как его «пнет» метод **notify()**
  - **notify()**:
    Этот метод вызывается, когда файбер в другом потоке стал готов (например, через channel).
    Задача алгоритма — немедленно «разбудить» свой поток ОС, если тот уснул в suspend_until, чтобы он как можно скорее перенес файбер из “удаленной очереди” в локальную
    - единственный метод, обязанный быть потоко-безопасным (вызываемый из другого потока)

Алгоритм выбора следующего Файбера

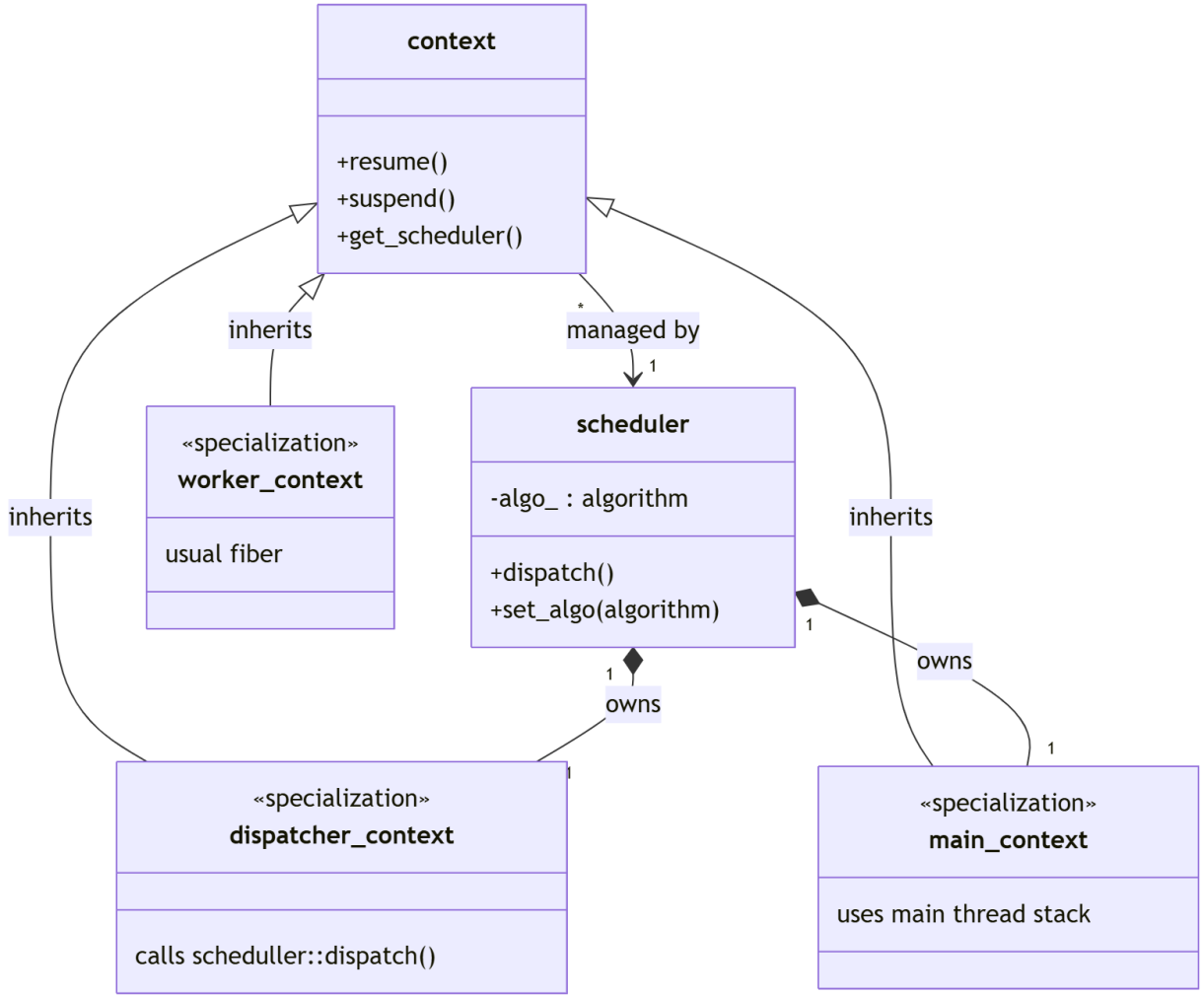
- Координация потоков (Multi-threading)
  - Кража задач (Work Stealing):
    - **pick_next()** содержит логику обращения к очередям других потоков, чтобы забрать у них лишнюю работу.
  - Work Sharing
  - NUMA
  - Round Robin («карусель» на одном потоке)



boost.fiber context-ы

contex – это файбер

- worker_context
- main_context
- dispatcher_context
- первые два исполняют пользовательский код
- последние два жестко привязаны к потоку OS



планировщик boost.fiber

Fiber Manager (dispatcher context)

- class scheduler
  - хозяин файберов
  - передает управление между файберами
- dispatcher context
  - «еще один файбер»
  - ~~«завхоз»~~ диспетчер, не делает полезной работы ☺
  - планируется на исполнение на общих правах
  - _всегда_ готов к исполнению
    - **всегда не пустая очередь готовых файберов**

планировщик boost.fiber

dispatcher context, диспетчер

- не просто «еще один файбер», а «режиссер» за кулисами
- не делает полезной работы ☺
- Обработка служебных очередей файберов:
  - ожидающих с таймаутом
  - «удаленно разбуженных» файберов
- инициирует «засыпание» рабочего потока (Idle management)
  - Если готовых файберов нет, он вызывает suspend_until(), чтобы эффективно усыпить поток ОС и не сжигать CPU
- Управление жизненным циклом (сборка мусора) файберов

Диспетчер планировщика boost.fiber

dispatcher context, Управление жизненным циклом (сборка мусора) файберов

- Файбер не может полностью уничтожить себя сам, так как в момент завершения он всё еще использует свой стек.
  - Когда рабочий файбер завершается, он помечается как terminated.
  - dispatcher_context подбирает такие контексты, освобождает память стека и удаляет объект контекста.

Диспетчер планировщика boost.fiber

dispatcher context, Обработка «удаленно разбуженных» файберов

- В многопоточной среде файбер может быть разбужен из другого потока ОС (например, данные пришли в channel). Чтобы избежать дорогостоящей синхронизации, Boost.Fiber использует двухступенчатую схему
  - Другой поток кладет файбер в Remote Ready Queue.
  - dispatcher_context при каждой итерации проверяет эту очередь и переносит файберы в Local Ready Queue.

А теперь – хардкор!

ДМИТРИЙ ОЛЬШАНСКИЙ

TLS ещё глубже – взгляд на ассемблер

```
thread_local int x;
int read_tls() {
    return x;
}
```

Более сложные схемы тоже опираются на один регистр

```
read_tls():
        mov     eax, dword ptr fs:[x@TPOFF]
        ret

read_tls():
        mrs     x0, tpidr_el0
        add     x0, x0, #:tprel_hi12:.LANCHOR0, lsl #12
        add     x0, x0, #:tprel_lo12_nc:.LANCHOR0
        ldr     w0, [x0]
        ret
        .set    .LANCHOR0,. + 0
x:
        .zero   4
```

https://godbolt.org/

Переключение контекста

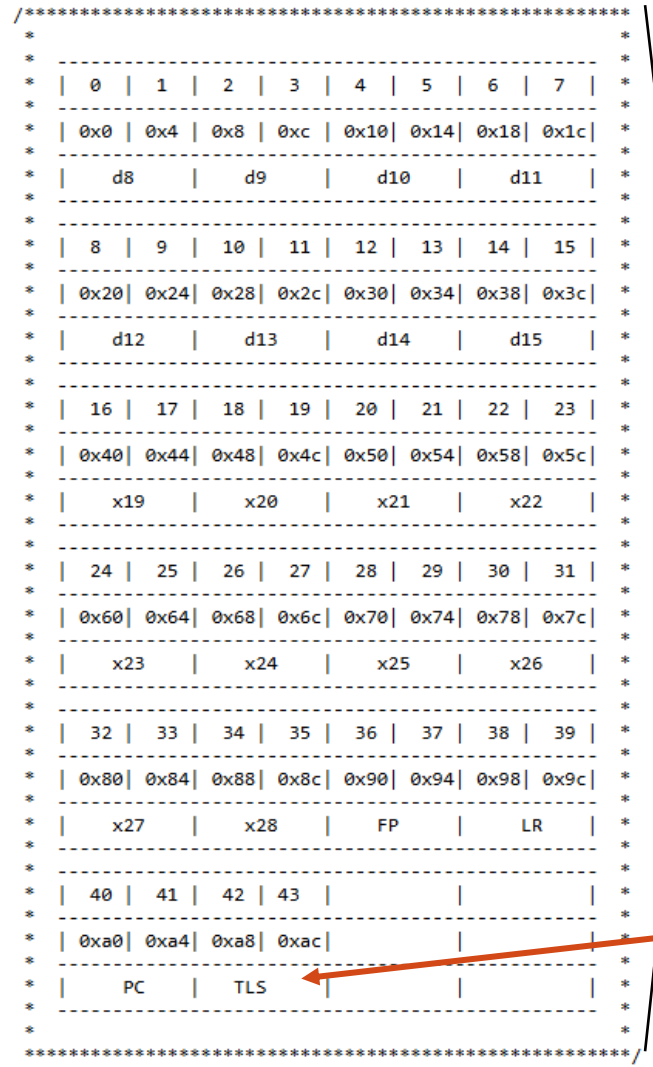


Таблица расположения сохраняемых регистров для ARM64

Наш регистр для TLS – оффсет 0xa8

Патч для функции переключения TLS для файберов

```
# save LR as PC
str  x30, [sp, #0xa0]

# save TLS
mrs  x9, tpidr_el0
str  x9, [sp, #0xa8]

# store RSP (pointing to context-data) in X0
mov  x4, sp

# restore RSP (pointing to context-data) from X1
mov  sp, x0

# restore TLS
ldr x9, [sp, #0xa8]
msr tpidr_el0, x9

# load d8 - d15
ldp  d8,  d9,  [sp, #0x00]
```

Переключение контекста #2

Теперь мы можем сохранять указатель TLS потока при переключении, однако

1. Нам нужен полностью сконструированный TLS для каждого файбера (далее “FLS”)
2. Нам нужен механизм для хранения того, чтобы было в TLS для каждого диспетчера

Идея – достать полноценный TLS можно создав поток «донор»

1. Чтобы не нагружать систему, он просто ждет завершения файбера по condvar
2. Бонусом – файберный стек вполне можно разместить на стеке донора
3. Для original TLS выделить область FLS и копировать ее содержимое при каждом переключении

Поток донор

```
std::atomic<bool> created_context{false};
donor_record donor_record_; // на самом деле часть this
const auto tls_values = tls::capture_current_values(); // сохраняем original TLS
donor_record_->donor_thread = thread([&created_context, this, tls_values]{
        tls_values.copy_to_current_tls(); // переносим original TLS в текущий поток
        make_fcontext(); // здесь создается файберный контекст с TLS донора
        created_context = true;
        std::unique_lock lk {donor_record_->mtx};
        while (!donor_record_->terminated) { // донор ждет завершения файбера
              donor_record_->cv.wait(lk);
        }
});
While (!created_context){ this_thread::yield(); } // примитивное ожидание создания контекста файбера
```

```
struct donor_record {
    mutex mtx;
    condition_variable cv;
    bool terminated{false};
    thread donor_thread;
};
```

Подводные камни

```
std::atomic<bool> created_context{false};

donor_record donor_record_; // на самом деле часть this

const auto tls_values = tls::capture_current_values(); // сохраняем real TLS

donor_record_->donor_thread = thread([&created_context, this, tls_values]{

          tls_values.copy_to_current_tls(); // переносим real TLS в текущий поток

          make_fcontext(); // здесь создается файберный контекст с TLS донора

          created_context = true;

          std::unique_lock lk {donor_record_->mtx};

          while(!donor_record_->terminated) { // донор ждет завершения файбера

                donor_record_->cv.wait(lk);

          } // здесь начнется вызов TLS деструкторов на потоке, который ничего не знает про файберы

});

while(!created_context){ this_thread::yield();} // примитивное ожидание создания контекста файбера
```

```
struct donor_record {
    mutex mtx;
    condition_variable cv;
    bool terminated{false};
    thread donor_thread;
};
```

Режим «Зомби»!

Надо дать отработать TLS деструкторам для потоков с завершенным файбером

Все блокировки и примитивы синхронизации файберные

Включаем режим «зомби» - мертвые файберы снова живут!

```
// пример – функция переключения на другой доступный файбер
void context::yield() noexcept {
    if (terminated_) { // текущий файбер на этом потоке уже умер
        this_thread::yield(); // ждем как обычный поток
        return;
    }
    // идем в планировщик и выбираем следующий файбер
    get_scheduler()->yield(context::active());
}
```

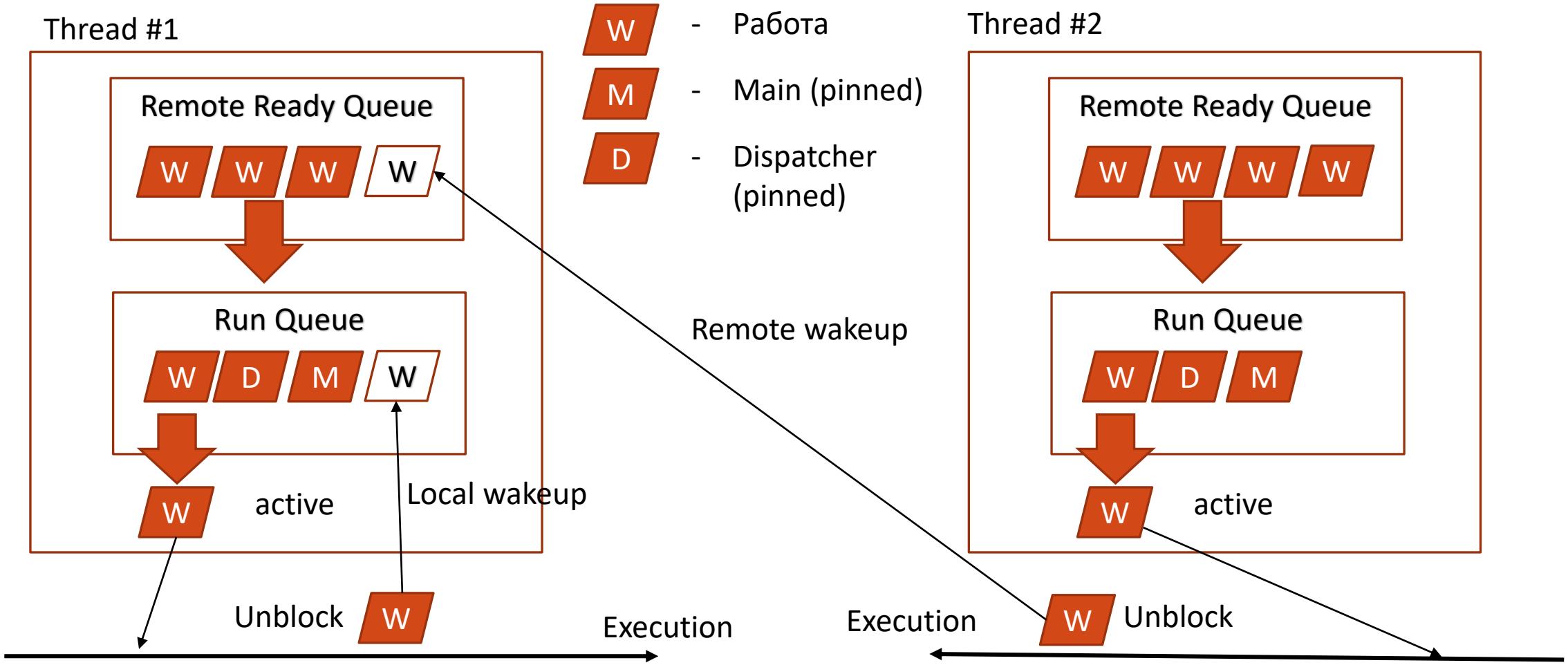
Режим «Зомби»!

А что с локами и другими примитивами?

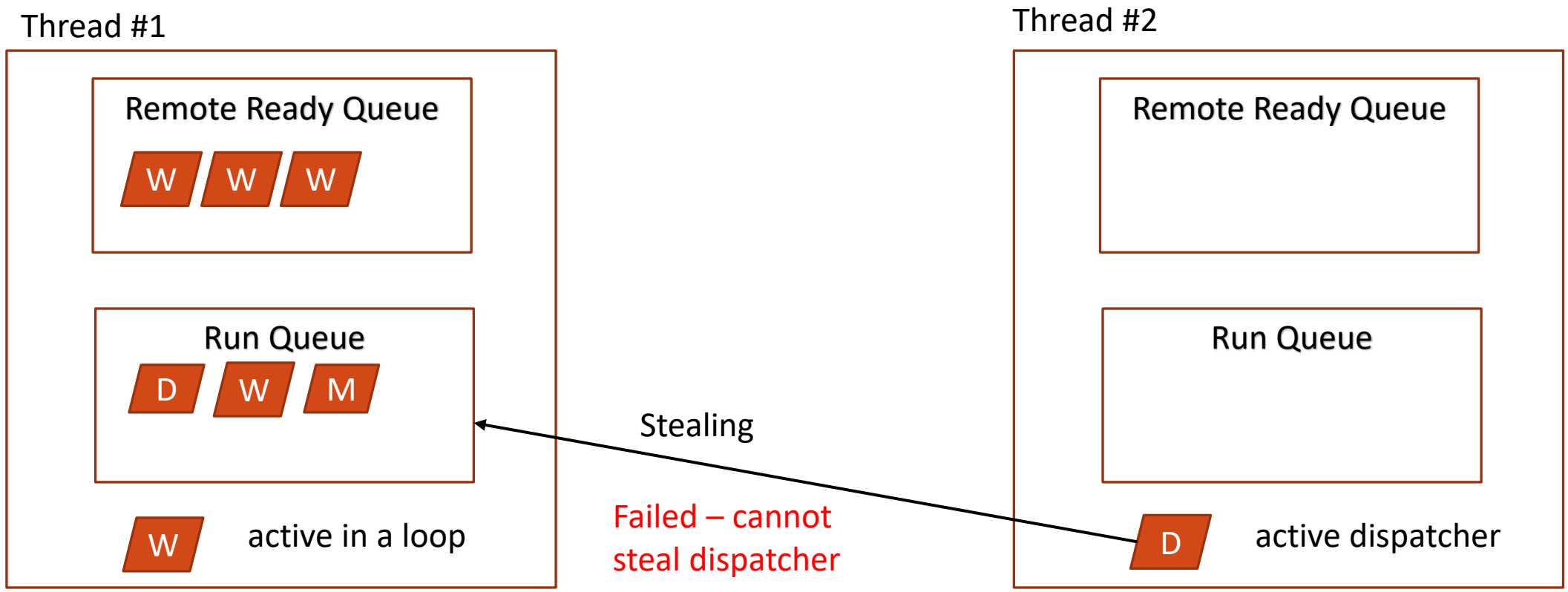
В основном spin-loop с yield

```
// прозрачно использует yield
void mutex::lock() {
    auto active_ctx = boost::fibers::context::active();
    while (true) {
        if (splk_.try_lock()) {
            owner_ = active_ctx;
            return;
        }
        active_ctx->yield();
    }
}
```

Особенности планировщика



Work stealing Boost.Fiber

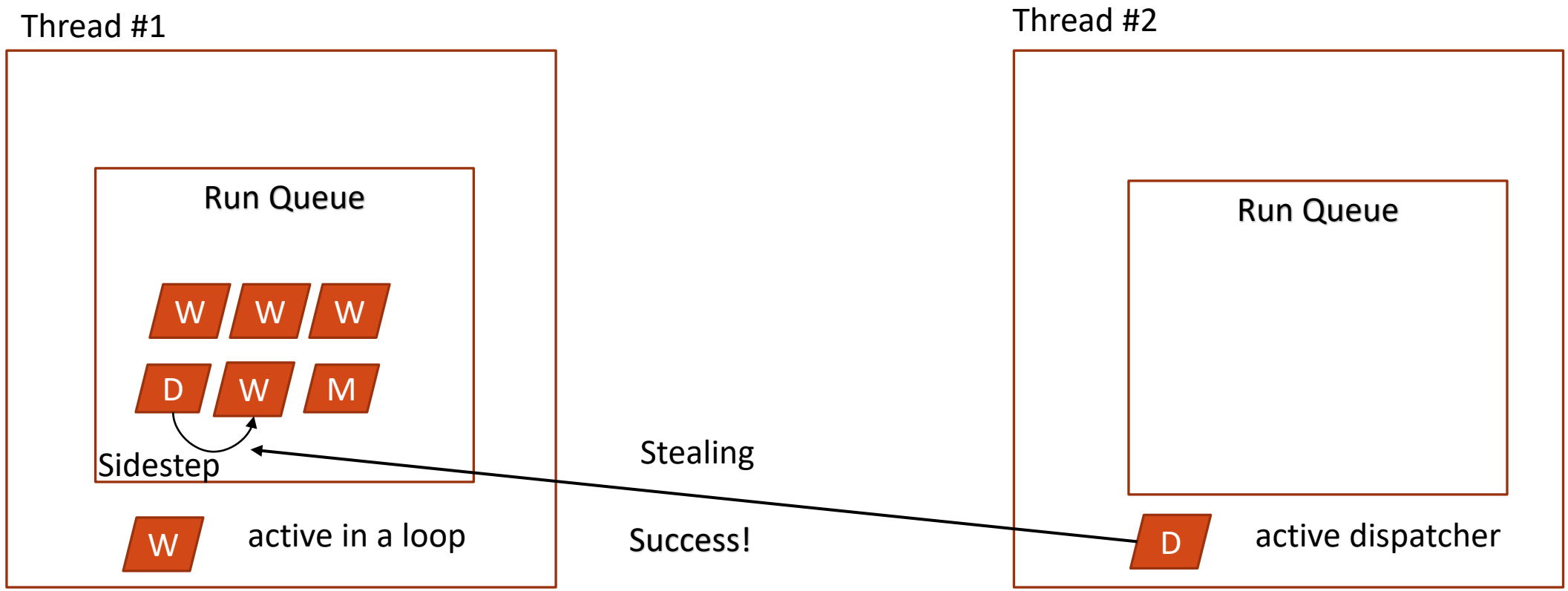


Работа есть, но мы не можем до нее добраться:

- stealing затыкается встретив pinned контекст, например dispatcher

- remote ready не спускается в run queue пока поток чем-то занят

Work stealing - наши хаки



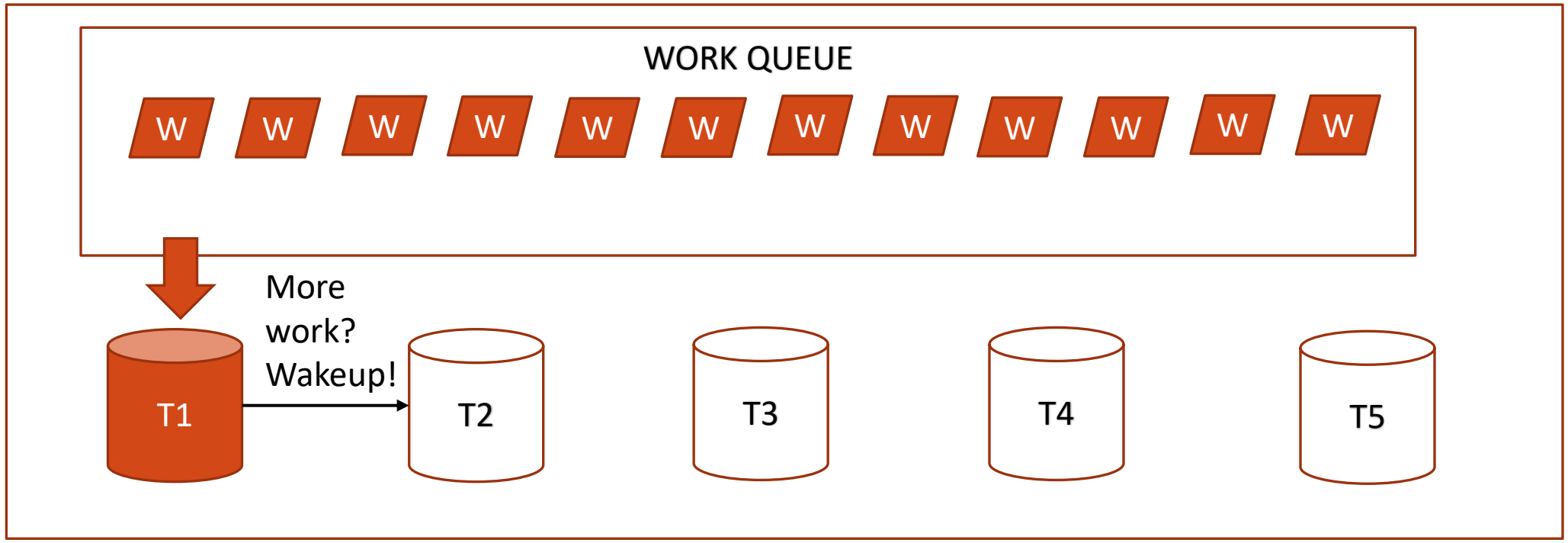
Решение – убить remote ready queue, усложнив синхронизацию

И наконец обходить pinned контекст на случай, если за ним есть работа

Пример – thread pool в Folly

Thread pool в Folly можно кастомизовать политиками

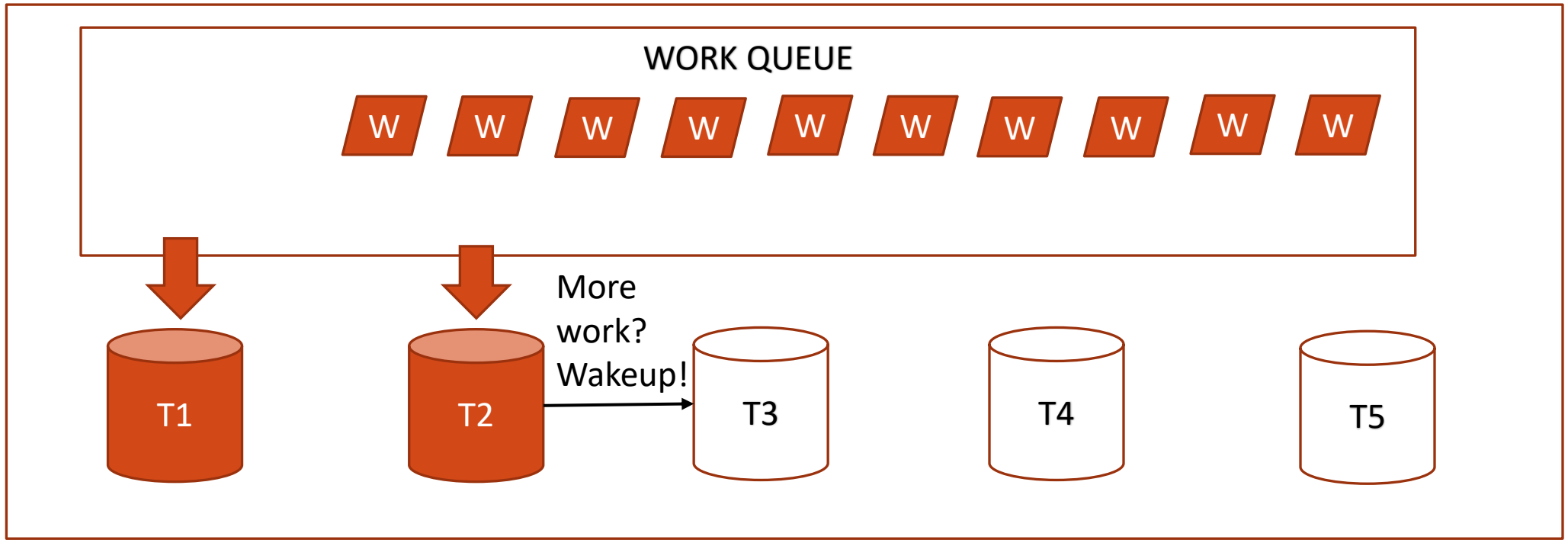
По умолчанию ThrottledSemaphore – ограниченное пробуждение «по цепочке»



Пример – thread pool в Folly

T1 пробудил T2

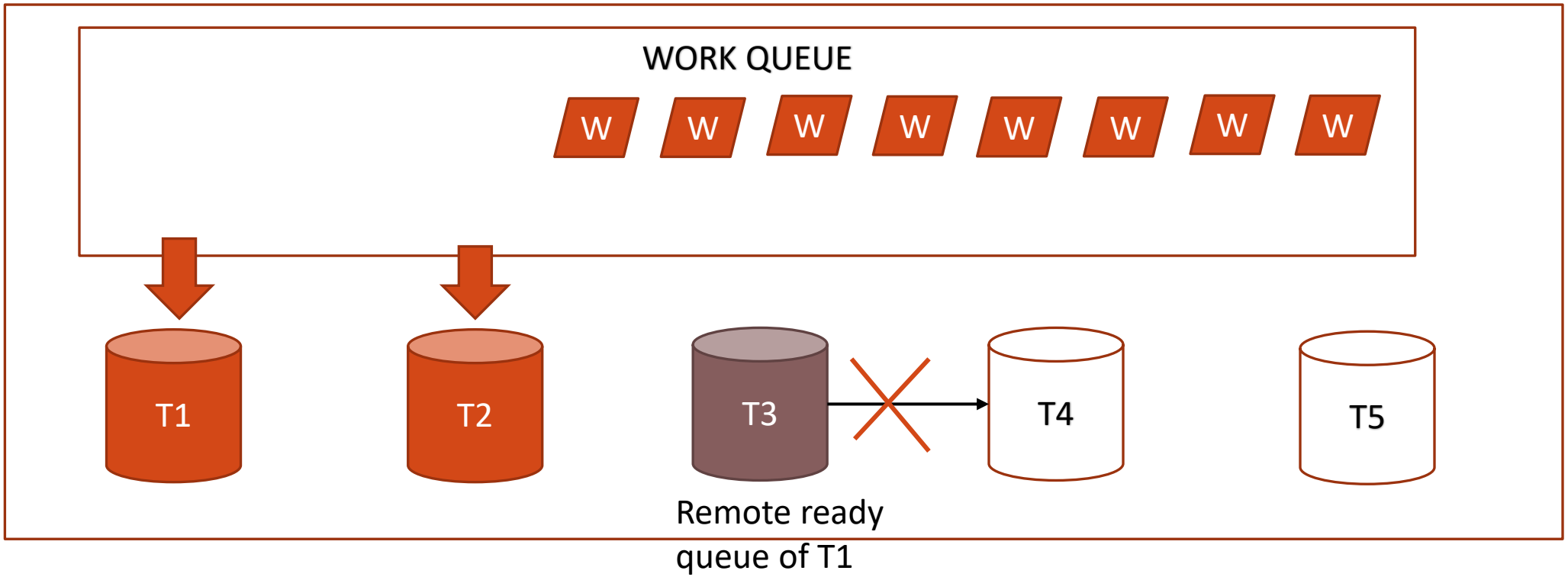
Работа все еще есть - теперь T2 пробуждает по цепочке T3



Пример – thread pool в Folly

T3 оказался в том же диспетчере, где живет T1

T1 работает в цикле пока есть работа, очередь remote ready не проверяется



Немного о скрытых блокировках

Вот мы пересадили приложение на наш рантайм

Ничего не крашится – хорошо

Тредпуллы не забивают нам очереди, work stealing работает

Весь pthread успешно подменен, FUTEX тоже

Но где же наша производительность???

iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more

offcputime спешит на помощь

Утилита из BCC offcputime показывет время проведенное потоком без исполнения на CPU

Удобно выявлять странные ожидания и локи

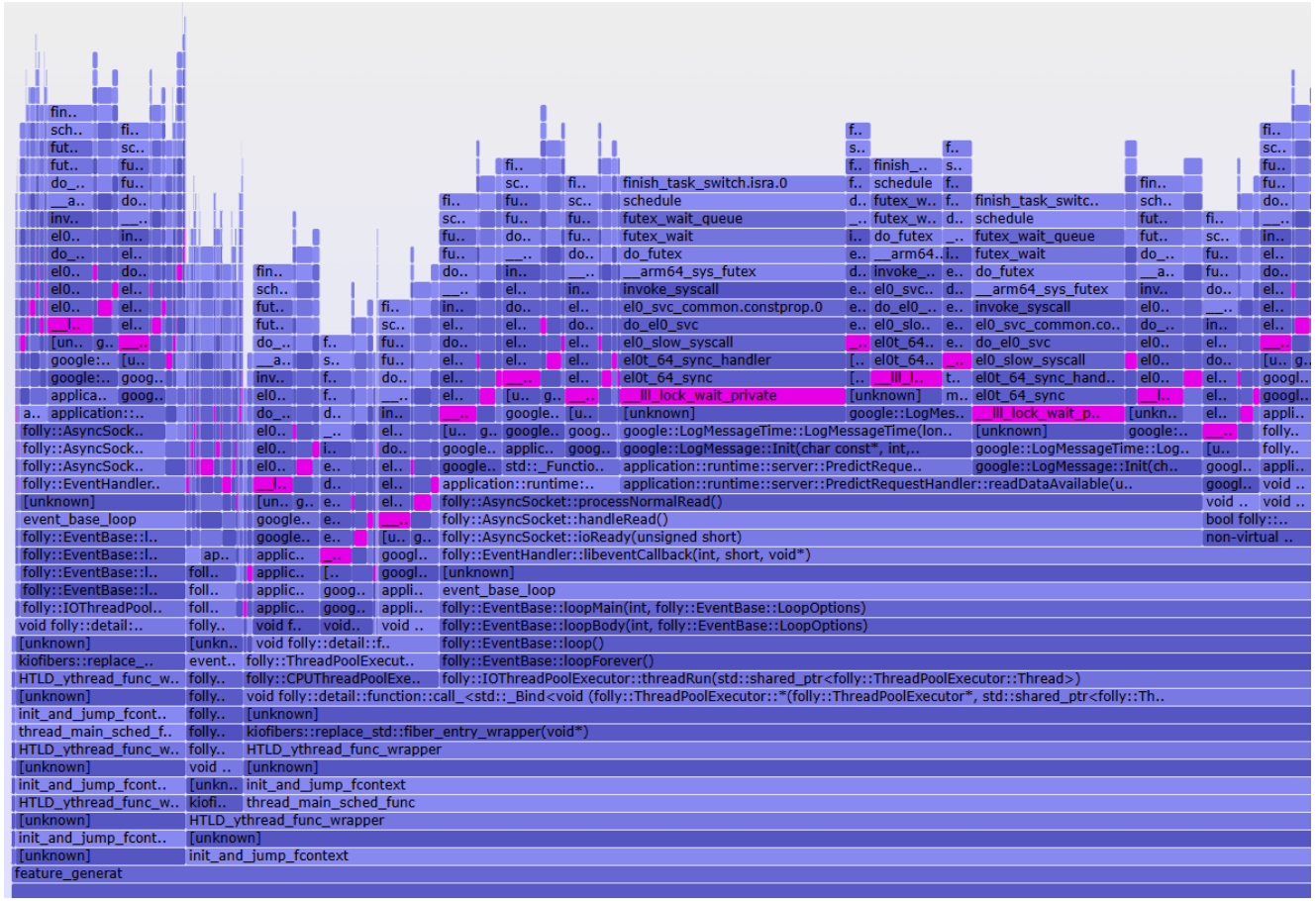
Пример справа реальный бенчмарк с нашим рантаймом

__lll_lock_wait_private – внутренняя блокировка libc

Шеф, все пропало?

... но стеки все подозрительно похожие

Подсвечен __lll_lock_wait_private



iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more

Логирование?

```
void LogMessageTime::CalcGmtOffset() {
  std::tm gmt_struct;
  int isDst = 0;
  if ( FLAGS_log_utc_time ) {
    localtime_r(&timestamp_, &gmt_struct);
    isDst = gmt_struct.tm_isdst;
    gmt_struct = time_struct_;
  } else {
    isDst = time_struct_.tm_isdst;
    gmtime_r(&timestamp_, &gmt_struct);
  }

  time_t gmt_sec = mktime(&gmt_struct);
  const long hour_secs = 3600;
  // If the Daylight Saving Time(isDst) is active subtract an hour from the current timestamp.
  gmtoffset_ = static_cast<long int>(timestamp_ - gmt_sec + (isDst ? hour_secs : 0) ) ;
}
```

Google glog v0.6.1

Эти вычисления делаются для каждого **даже выключенного** лога

mktime вызывает tzset, который в свою очередь лок

tzset не чаще раза в секунду

```
boost::fibers::mutex mktime_mtx;
thread_local timespec last_call;
static time_t localtime_offset;

extern "C" time_t mktime (tm *tp)
{
    timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    if (nanos(ts) > nanos(last_call) + 1000000000) {
        if (mktime_mtx.try_lock()){
            tzset();
            mktime_mtx.unlock();
        }
        last_call = ts;
    }

    return __mktime_internal (tp, localtime_r, &localtime_offset);
}
```

Ускорение на 35% e2e!

И напоследок

АНТОН МАЛАХОВ

Worker Thread Local Storage (WTLS)

- TLS подменили, thread_local стали файбер-локальными
  - На каждый файбер – спящий поток-донор, чей стек и TLS используются от лица файбера
- Поломка планировщика boost_fiber: Нужна логика с thread_local переменной
- Как получить информацию о конкретном потоке? Взять трассу?
- Нужны тред-локальные кеши для оптимизации
  - Кеш ведет к учащению кеш-миссов. Дожили...
- В случае аллокатора с кешами не уменьшается расход памяти
  - А мог бы: потоков стало меньше, и кешей должно стать меньше

WTLS: Наивный интерфейс

```
thread_local T p = 0; // Файбер-локальная переменная.
WTLS_REGISTER(p)      // Сделать переменную worker_thread-локальной
```

- Теперь переменная копируется из одного TLS в другой, при переключении файберов.
- После небольшой отладки с int работает.
  - Главное не копировать из уничтоженного TLS.
- Большие объекты тоже по значению копировать?

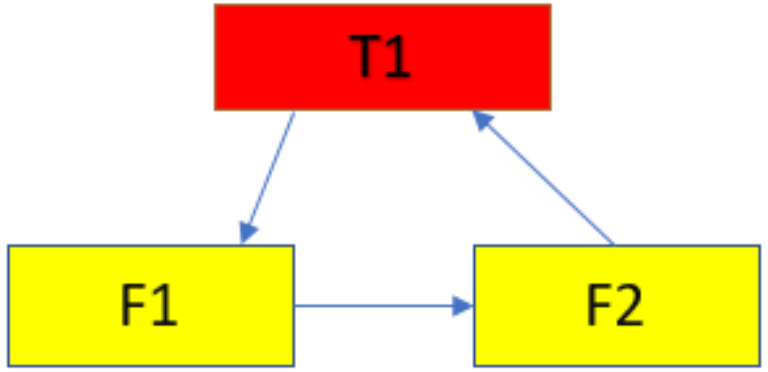
WTLS: Проблемы с указателями

```
thread_local cache_t cache;
thread_local cache_t *p;
WTLS_REGISTER(p)
void use_cache() {
    if (!p)
        p = &cache;
}
```

Кто первым зашел в use_cache()?

В чей TLS Все ходят?

Проблема времени жизни TLS.

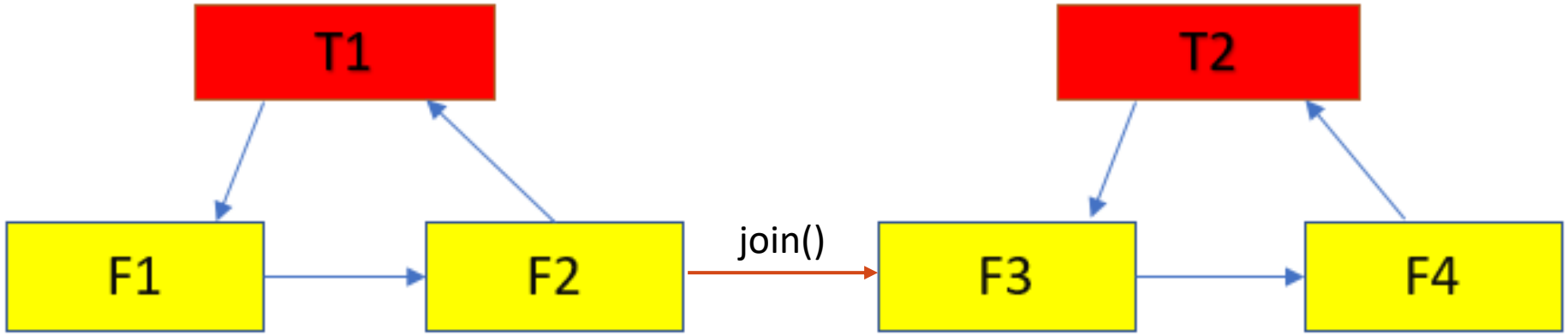


WTLS: Интерфейс чуть умнее

```
WTLS_REGISTER(p) // Сделать p worker_thread-локальной.
WTLS_ADDR_OF(p)  // Вернуть адрес p из TLS потока воркера.


thread_local cache_t cache;
thread_local cache_t *p = 0;
void ordinary_constructor() {
    if (!p)
        // По указателю ходим только в TLS воркера!
        p = WTLS_ADDR_OF(cache);
}
```

WTLS: Причины гонок



T1(F2): F3.join() -> F3.donor_thread.join()

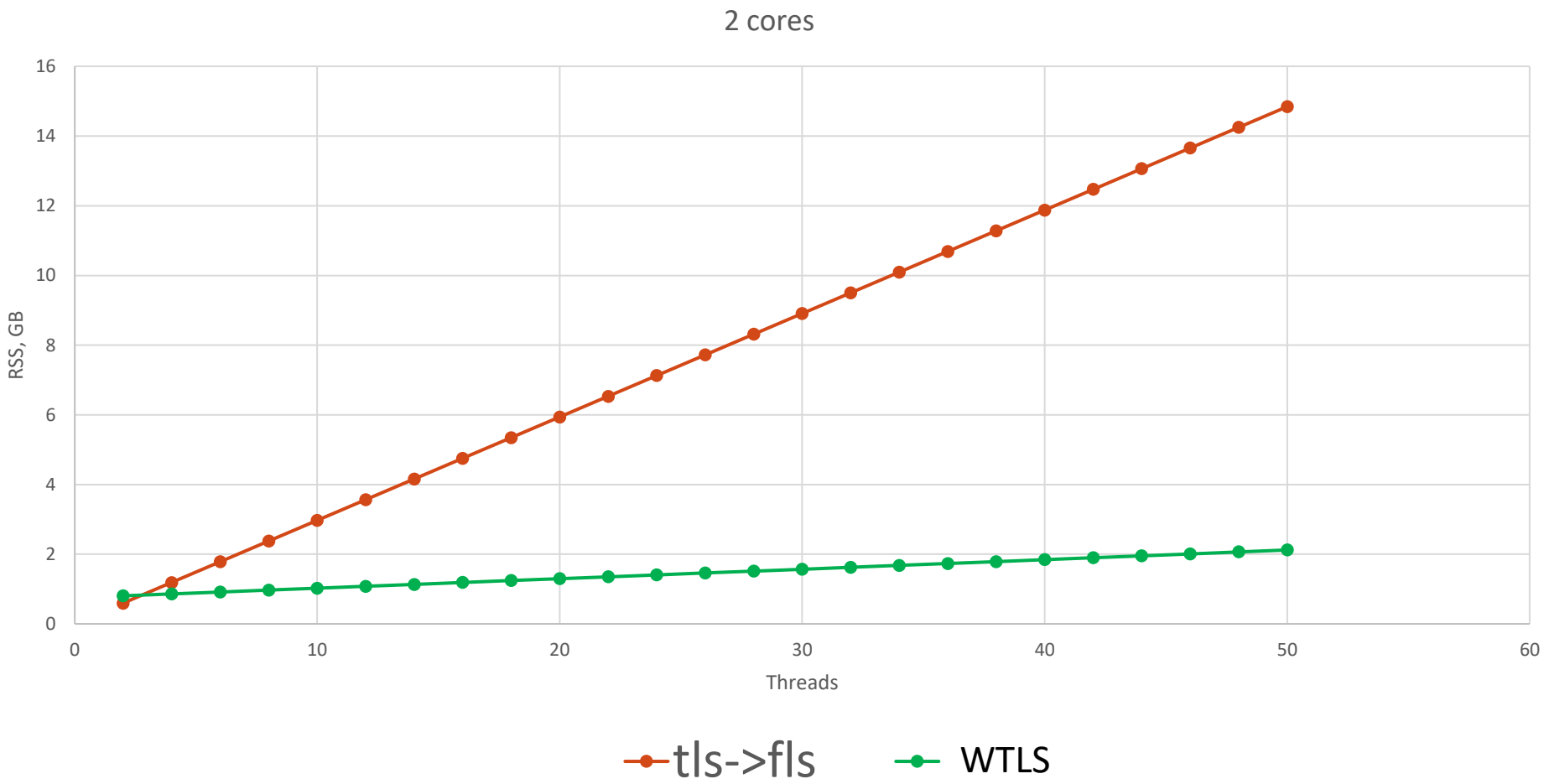
Умирающий поток лезет в TLS(T2) параллельно с потоком T2 от лица F4.

WTLS: Интерфейс финальный

```
WTLS_ADDR_OF(p)       // Вернуть адрес p из TLS потока воркера.
WTLS_REGISTER(p)      // Сделать p worker thread-локальной.
AT_FIBER_DESTROY(d)   // Вызвать d перед файбер-локальными
                      // деструкторами.

thread_local cache_t cache;
thread_local cache_t *p = 0;
WTLS_REGISTER(p)
void ordinary_constructor() { p = WTLS_ADDR_OF(cache); }
// Натравить указатель на бесхозный кеш
void at_destroy() { p = &cache; }
AT_FIBER_DESTROY(at_destroy);
```

WTLS: Синтетический бенчмарк



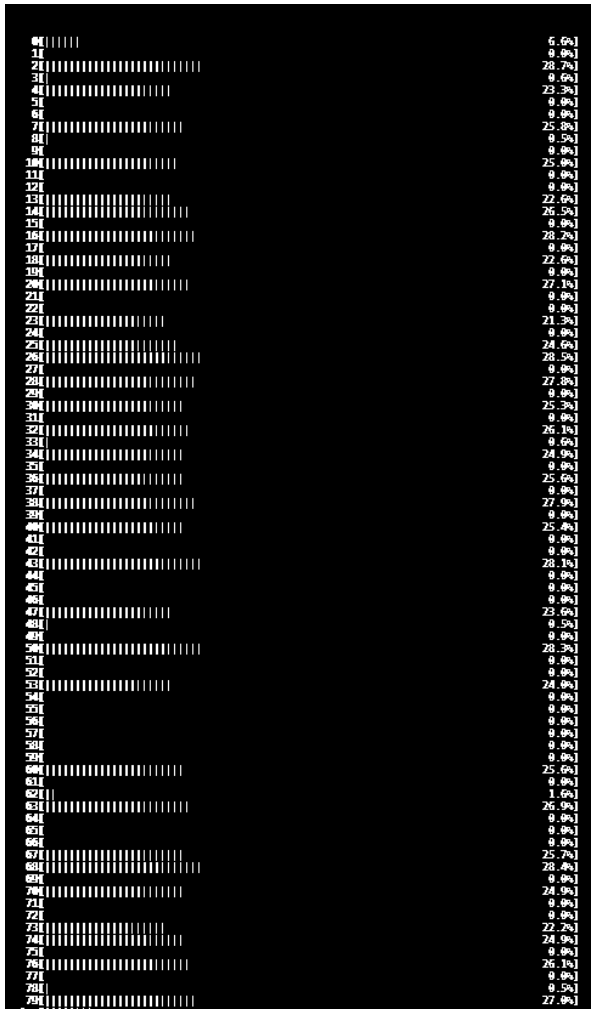
Всегда ли файберы хороши?

Выполняя один экземпляр mysqld без жёсткой привязки к ядрам, но с ограничением на 800% (--cpu=8)

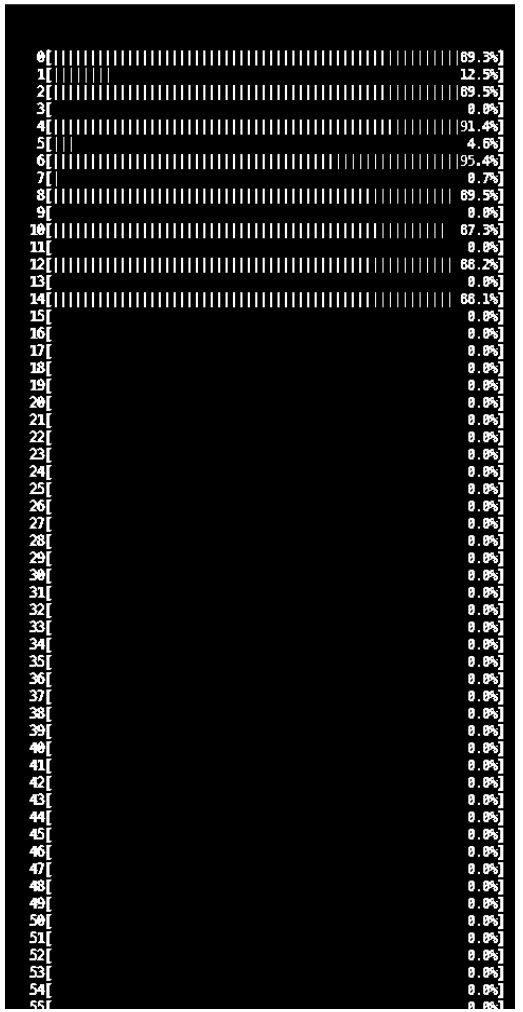
Это

- снижает эффект переподписки
- Увеличивает объём доступного потокам L1, L2 кеша
- Ломает красивые графики

Original pthreads

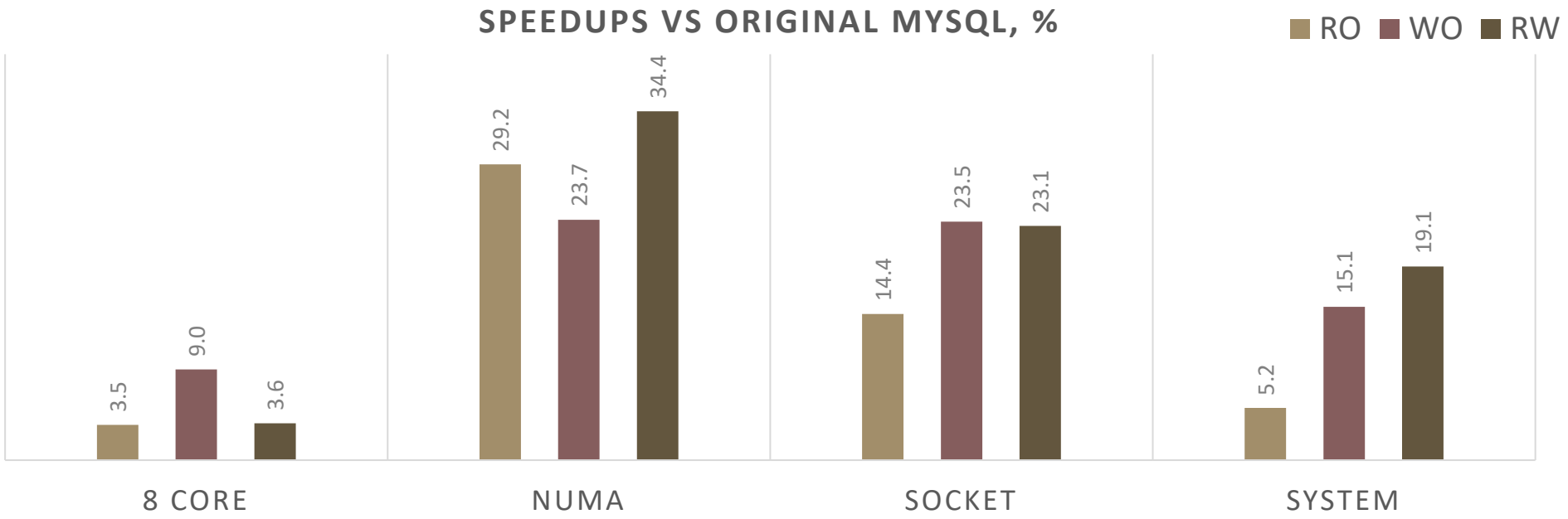


Fibers on 8 cores



Результаты нашей работы

- Задача “ускорить MySQL не меняя MySQL” – выполнена
  - Патч в MySQL полностью нивелирован
- Производительность при отсутствии ограничений CPU-time растёт



Всем Спасибо!

Уж сколько лет твердим мы миру, что ОС плоха, скедуллер плох,

Он потребляет много жиру, чтоб с оверхедом не заглох.

Настало время выдать миру, как же иначе делать всем,

Стыдливо пряча и скрывая свои заплатки насовсем,

Подменой праведной потоков и асинхронности, дружок,

Мы получили супербыстрый, универсальный хак-движок!