Инструмент для верификации конкурентных структур данных LTest

Илья Кокорин
kokorin.ilya.1998@gmail.com

Кирилл Гарманов
garmanow.kirill@gmail.com

Кто мы такие?

Старший разработчик в команде баз данных

vk.com/rpc

t.me/ilyambda

Разработчик в команде core infrastructure

vk.com/svilex

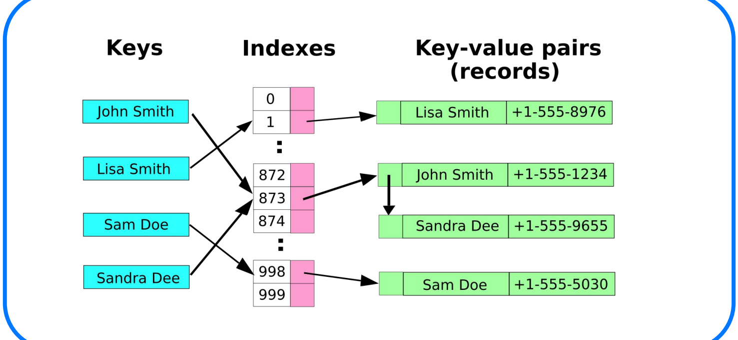
t.me/lim123123123

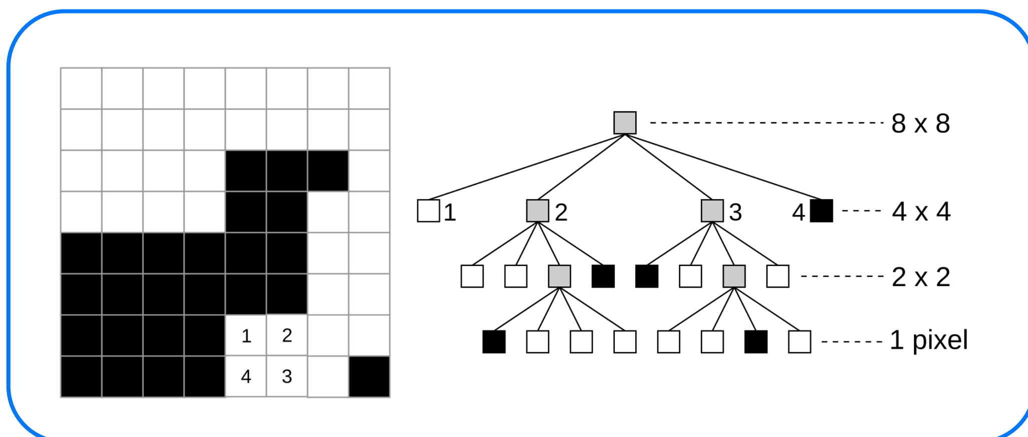
Действие первое:
LTest in a nutshell

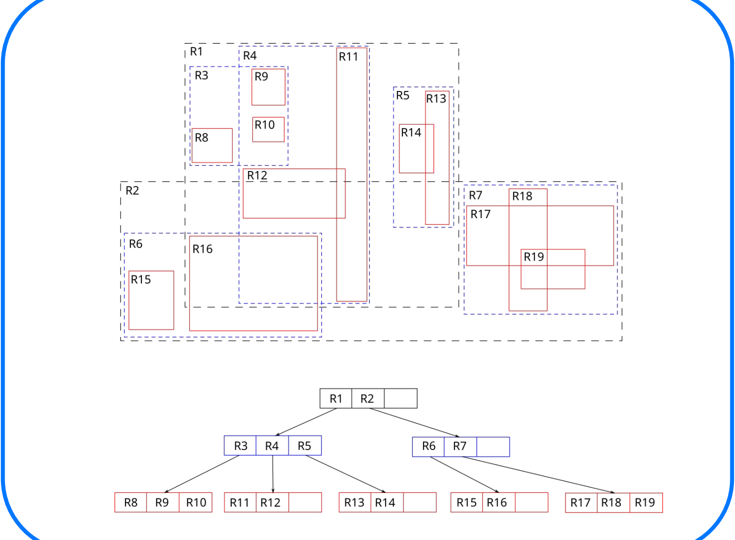
- Что такое конкурентные структуры данных
- Как их тестировать
- Какие проблемы возникают при тестировании
- Как LTest решает эти проблемы
- Как LTest устроен внутри (вкратце!)

Последовательные структуры данных

Множество разновидностей

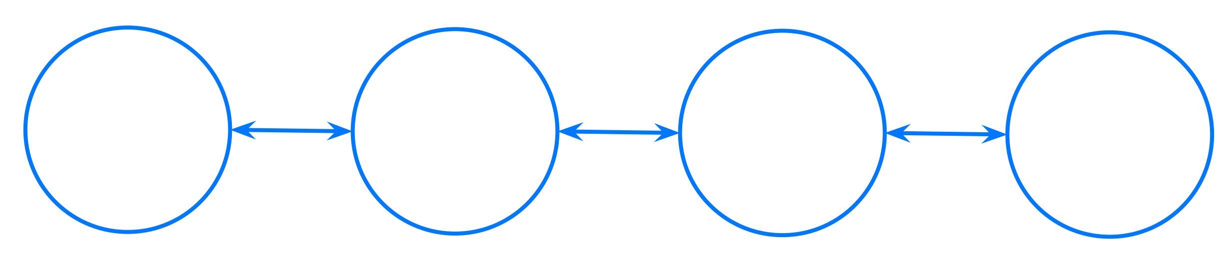


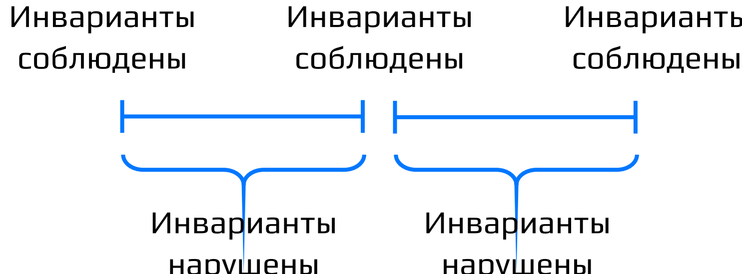




Последовательные структуры данных

- Следующая операция начинается только после конца предыдущей

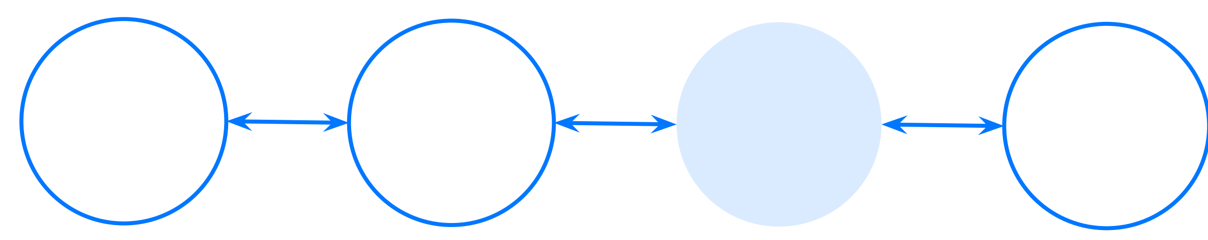


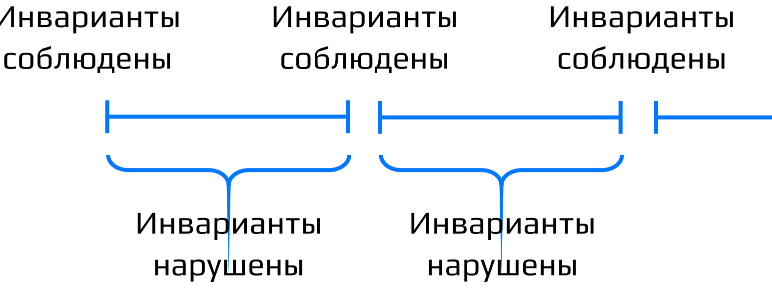


A.next == &B ↔ B.prev == &A

Последовательные структуры данных

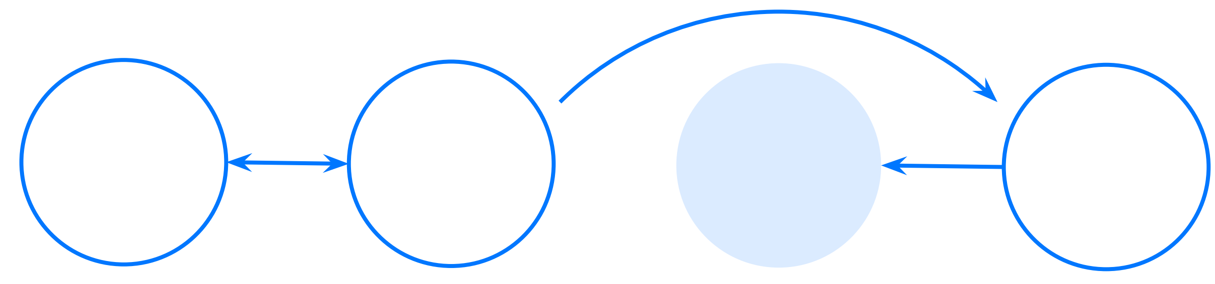
- В начале операции соблюдены все инварианты структуры данных

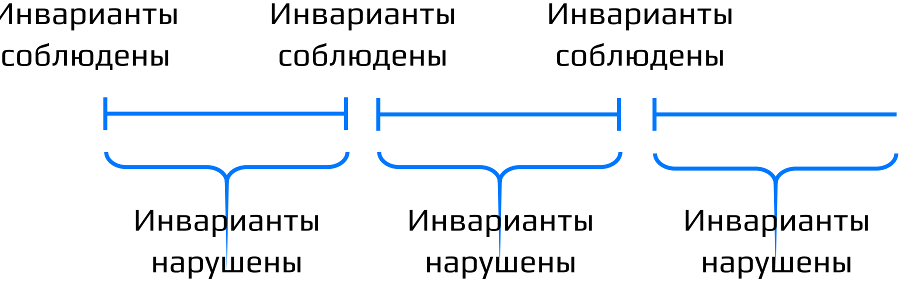




Последовательные структуры данных

- В ходе операции инварианты структуры данных могут нарушаться





A.next->prev != &A

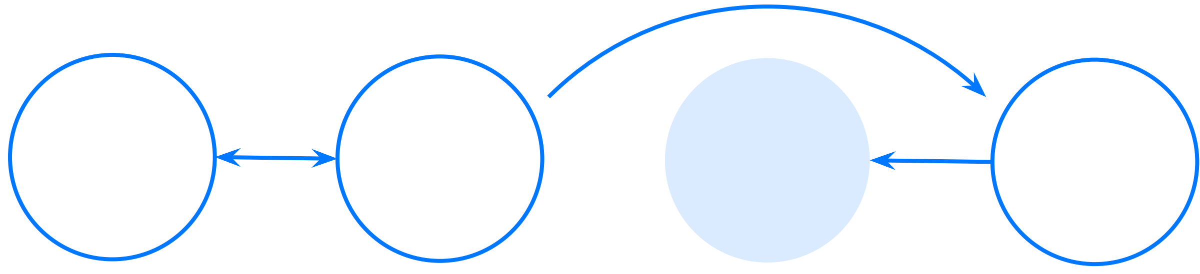
Последовательные структуры данных

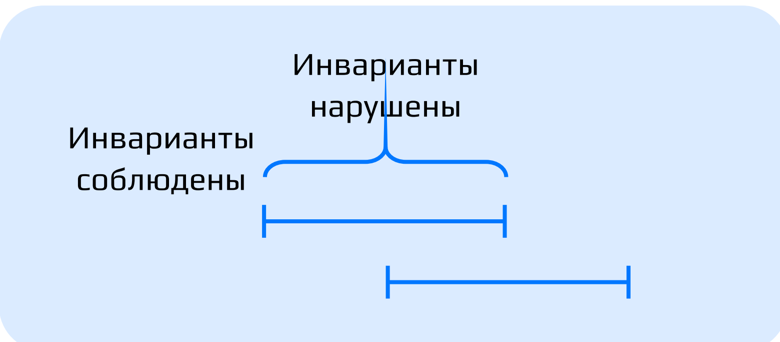
- К концу операции все инварианты структуры данных восстановлены, можно начинать следующую операцию

Инварианты соблюдены
Инварианты соблюдены
Инварианты соблюдены
Инварианты соблюдены

Инварианты нарушены
Инварианты нарушены
Инварианты нарушены

Конкурентные структуры данных





- Операция начинается до завершения предыдущей
- В начале операции структура данных может находиться в неконсистентном состоянии

Конкурентные структуры данных это сложно

```
template<typename T>
struct stack_t {
  void push(T value) {
    auto* new_node = new node_t<T>{};
    new_node->data = std::move(value),
    new_node->next = head;
    head = new_node;
  }

  std::optional<T> pop() noexcept {
    if (head == nullptr) return
std::nullopt;
    T result{std::move(head->data)};
    node_t<T>* const new_head =
head->next;
    delete head;
    head = new_head;
    return result;
  }

  std::optional<T> peek() const {
    return head != nullptr
      ? std::optional{head->data}
      : std::nullopt;
  }
private:
  node_t<T>* head{nullptr};
};
```

Treiber R. K. et al.
Systems programming:
Coping with parallelism

```
template<typename T>
struct concurrent_stack_t {
  void push(T value) {
    auto* const new_node = new node_t<T>{};
    new_node->data = std::move(value);
    while (true) {
      node_t<T>* cur_head = head.load();
      new_node->next = cur_head;
      if (head.compare_exchange_strong(
                cur_head, new_node))
        return;
    }
  }
  std::optional<T> pop() {
    while (true) {
      node_t<T>* cur_head = head.load();
      if (cur_head == nullptr) return std::nullopt;
      if (head.compare_exchange_strong(
          cur_head, cur_head->next)) {
        T result{cur_head->data};
        retire(cur_head);
        return result;
      }
    }
  }

  std::optional<T> peek() const {
    const auto* const cur_head = head.load();
    return cur_head != nullptr
      ? std::optional{cur_head->data} : std::nullopt;
  }
private:
  std::atomic<node_t<T>*> head{nullptr};
};
```

```
void push(T value) {
  auto* new_node = new node_t<T>{};
  new_node->data = std::move(value),
  new_node->next = head;
  head = new_node;
}
```

- Появляются атомарные регистры
- Нужно думать над memory order
- Появляются CAS loops

```
void push(T value) {
  auto* const new_node = new node_t<T>{};
  new_node->data = std::move(value);
  while (true) {
    node_t<T>* cur_head = head.load();
    new_node->next = cur_head;
    if (head.compare_exchange_strong(
        cur_head, new_node))
      return;
  }
}
```

Конкурентные структуры данных это сложно

```
std::optional<T> pop() noexcept {
  if (head == nullptr)
    return std::nullopt;
  T result{std::move(head->data)};
  node_t<T>* const new_head =
     head->next;
  delete head;
  head = new_head;
  return result;
}
```

```
std::optional<T> pop() {
  while (true) {
    node_t<T>* cur_head = head.load();
    if (cur_head == nullptr)
      return std::nullopt;
    if (head.compare_exchange_strong(
        cur_head, cur_head->next)) {
      T result{cur_head->data};
      retire(cur_head);
      return result;
    }
  }
}
```

- Нельзя использовать `std::move`
- Усложняется memory reclamation

Fraser K. Practical lock-freedom

Michael M. M. Hazard pointers: Safe memory reclamation for lock-free objects

Конкурентные структуры данных это полезно

- Разделяемый между рабочими потоками веб-сервера кеш

CACHE

Querries

Конкурентные структуры данных это полезно

- Producer-consumer очереди для передачи заданий от потоков, обслуживающих пользовательские запросы, потокам, работающим с блочным устройством

Querries

Конкурентные структуры данных это полезно

Наш код выглядит так

```
NodeRef tail = tail_;
NodePtr next = GetNext(tail).load(std::memory_order_acquire);

if (tail == &stub_) {
    if (next == nullptr) {
        if (mode == PopMode::kWeak) return nullptr;
        if (tail == head_->load(std::memory_order_acquire))
            return nullptr;
        next = BlockThreadUntilNotNull(GetNext(tail));
    }
    GetNext(stub_).store(nullptr, std::memory_order_relaxed);
    tail_ = *next;
    tail = *next;
    next = GetNext(tail).load(std::memory_order_acquire);
}

if (next != nullptr) {
    tail_ = *next;
    GetNext(tail).store(nullptr, std::memory_order_relaxed);
    return tail;
}

NodeRef head = head_->load(std::memory_order_acquire);
if (head == tail &&
    head_->compare_exchange_strong(head, stub_,
                                   std::memory_order_release,
                                   std::memory_order_relaxed)) {
    tail_ = stub_;
    return tail;
}

if (mode == PopMode::kWeak) return nullptr;
next = BlockThreadUntilNotNull(GetNext(tail));
tail_ = *next;
GetNext(tail).store(nullptr, std::memory_order_relaxed);
return tail;
```

Чтобы продуктовый код выглядел так

```
#include <userver/easy.hpp>
#include "schemas/key_value.hpp"

int main(int argc, char* argv[]) {
    using namespace userver;
    easy::HttpWith<easy::PgDep>(argc, argv).Get(
      "/kv", [](formats::json::Value request_json,
                const easy::PgDep& dep) {
      auto key =
        request_json.As<schemas::KeyRequest>().key;
      auto res = dep.pg().Execute(
          storages::postgres::ClusterHostType::kSlave,
          "SELECT value FROM key_value_table WHERE key=$1",
          key
      );

      schemas::KeyValue response{
        key, res[0][0].As<std::string>()};
      return formats::json::ValueBuilder{response}.ExtractValue();
  });
}
```

[Никита Костливцев, Устройство coroutine scheduler для современного рантайма]

(Почти) Herlihy & Wing Queue

```
struct queue_t {
  explicit queue_t(const uint32_t queue_size) noexcept
     : data_{queue_size} {
    for (int32_t i = 0; i < queue_size; ++i) {
      data_[i] = std::nullopt;
    }
  }

  void Push(const int32_t value) noexcept { /* ... */ }
  std::optional<int32_t> Pop() noexcept { /* ... */ }
private:
  std::vector<std::atomic<std::optional<int32_t>>> data_;
  std::atomic_int32_t index_{0};
};
```

- FIFO-очередь на массиве ограниченного размера
- Массив элементов и индекс следующей вставки

Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

(Почти) Herlihy & Wing Queue: Вставка

```
void Push(
    const int32_t value) noexcept {
  const int32_t i =
    index_.fetch_add(1);
  data_[i] = value;
}
```

∅	24	42	∅	∅	∅

- Для вставки нужно атомарно прочитать и увеличить на единицу значение индекса в массиве

(Почти) Herlihy & Wing Queue: Вставка

```
void Push(
    const int32_t value) noexcept {
  const int32_t i =
    index_.fetch_add(1);
  data_[i] = value;
}
```

∅	24	42	∅	∅	∅

- Для вставки нужно атомарно прочитать и увеличить на единицу значение индекса в массиве

(Почти) Herlihy & Wing Queue: Вставка

```
void Push(
    const int32_t value) noexcept {
  const int32_t i =
    index_.fetch_add(1);
  data_[i] = value;
}
```

∅	24	42	25	∅	∅

- Сохраняем значение по прочитанному индексу
- Следующее значение будет сохранено в следующую ячейку

(Почти) Herlihy & Wing Queue: Удаление

- Для удаления нужно запомнить текущее значение индекса

```
std::optional<int32_t> Pop() noexcept {
  const int32_t last = index_.load();
  for (int32_t i = 0; i < last; ++i) {
    const auto maybe_result =
      data_[i].exchange(std::nullopt);
    if (maybe_result.has_value()) {
      return *maybe_result;
    }
  }
  return std::nullopt;
}
```

∅	24	42	∅	∅	∅

(Почти) Herlihy & Wing Queue: Удаление

- Каждый элемент до запомненной границы пытаемся атомарно заменить на пустое значение

```
std::optional<int32_t> Pop() noexcept {
 const int32_t last = index_.load();
 for (int32_t i = 0; i < last; ++i) {
   const auto maybe_result =
     data_[i].exchange(std::nullopt);
   if (maybe_result.has_value()) {
     return *maybe_result;
   }
 }
 return std::nullopt;
}
```

∅	24	42	∅	∅	∅

(Почти) Herlihy & Wing Queue: Удаление

- Если ранее хранившееся в ячейке значение было пустым, переходим к следующей ячейке

```
std::optional<int32_t> Pop() noexcept {
 const int32_t last = index_.load();
 for (int32_t i = 0; i < last; ++i) {
   const auto maybe_result =
     data_[i].exchange(std::nullopt);
   if (maybe_result.has_value()) {
     return *maybe_result;
   }
 }
 return std::nullopt;
}
```

∅	24	42	∅	∅	∅

(Почти) Herlihy & Wing Queue: Удаление

- Если ранее хранившееся в ячейке значение было пустым, переходим к следующей ячейке

```
std::optional<int32_t> Pop() noexcept {
 const int32_t last = index_.load();
 for (int32_t i = 0; i < last; ++i) {
   const auto maybe_result =
     data_[i].exchange(std::nullopt);
   if (maybe_result.has_value()) {
     return *maybe_result;
   }
 }
 return std::nullopt;
}
```

(Почти) Herlihy & Wing Queue: Удаление

- Первое непустое встреченное значение возвращается из функции Pop()

```
std::optional<int32_t> Pop() noexcept {
 const int32_t last = index_.load();
 for (int32_t i = 0; i < last; ++i) {
   const auto maybe_result =
     data_[i].exchange(std::nullopt);
   if (maybe_result.has_value()) {
     return *maybe_result;
   }
 }
 return std::nullopt;
}
```

∅	∅	42	∅	∅	∅

24

P_1	P_2
data_ = {42, ∅, ∅, ...} index_ = 1	
	Invoke(P_2, Queue, Pop)
	last = index_.load() // 1

- Непустая очередь
- Начинается операция Pop()
- Исполняющий операцию поток вытесняется с ядра планировщиком ОС

Тестирование конкурентного кода

P_1	P_2
data_ = {42, ∅, ∅, ...} index_ = 1	
	Invoke(P_2, Queue, Pop)
	last = index_.load() // 1
Invoke(P_1, Queue, Push, 24)	
i = index_.fetch_add(1) // 1	
data_ = {42, ∅, ∅, ...} index_ = 2	
data_[1] = 24	
data_ = {42, 24, ∅, ...} index_ = 2	
Response(P_1, Queue, Push, Ok)	

Queue.Push(24)

- Другой поток исполняет операцию Push(24)

Тестирование конкурентного кода

	P_1	P_2
	`data_ = {42, ⌀, ⌀, ...}` `index_ = 1`	
		`Invoke(P₂, Queue, Pop)`
		`last = index_.load() // 1`
Queue.Push(24)	`Invoke(P₁, Queue, Push, 24)`	
	`i = index_.fetch_add(1) // 1`	
	`data_ = {42, ⌀, ⌀, ...}` `index_ = 2`	
	`data_[1] = 24`	
	`data_ = {42, 24, ⌀, ...}` `index_ = 2`	
	`Response(P₁, Queue, Push, Ok)`	
Queue.Pop()	`Invoke(P₁, Queue, Pop)`	
	`last = index_.load() // 2`	
	`maybe_result = data_[0].exchange(std::nullopt) // 42`	
	`data_ = {⌀, 24, ⌀, ...}` `index_ = 2`	
	`Response(P₁, Queue, Pop, 42)`	

- Другой поток исполняет операцию Pop()

Тестирование конкурентного кода

P_1	P_2
data_ = {42, ø, ø, ...} index_ = 1	
	Invoke(P_2, Queue, Pop)
	last = index_.load() // 1
Invoke(P_1, Queue, Push, 24)	
i = index_.fetch_add(1) // 1	
data_ = {42, ø, ø, ...} index_ = 2	
data_[1] = 24	
data_ = {42, 24, ø, ...} index_ = 2	
Response(P_1, Queue, Push, Ok)	
Invoke(P_1, Queue, Pop)	
last = index_.load() // 2	
maybe_result = data_[0].exchange(std::nullopt) // 42	
data_ = {ø, 24, ø, ...} index_ = 2	
Response(P_1, Queue, Pop, 42)	
	maybe_result = data_[0].exchange(std::nullopt) // ø
data_ = {ø, 24, ø, ...} index_ = 2	
	Response(P_2, Queue, Pop, ø)

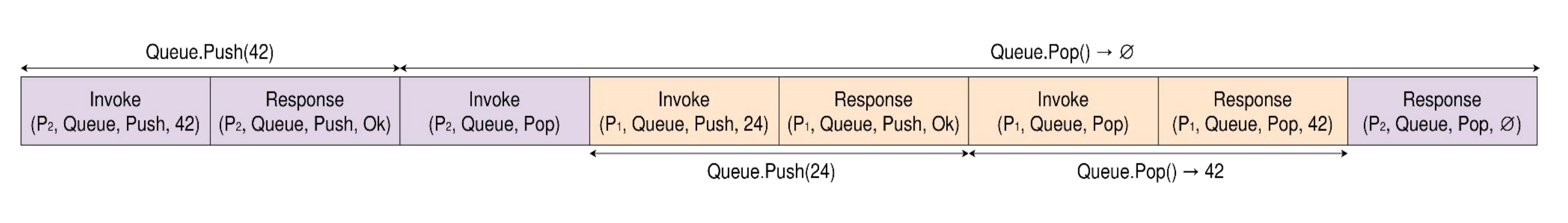
Queue.Push(24)

Queue.Pop()

Queue.Pop()

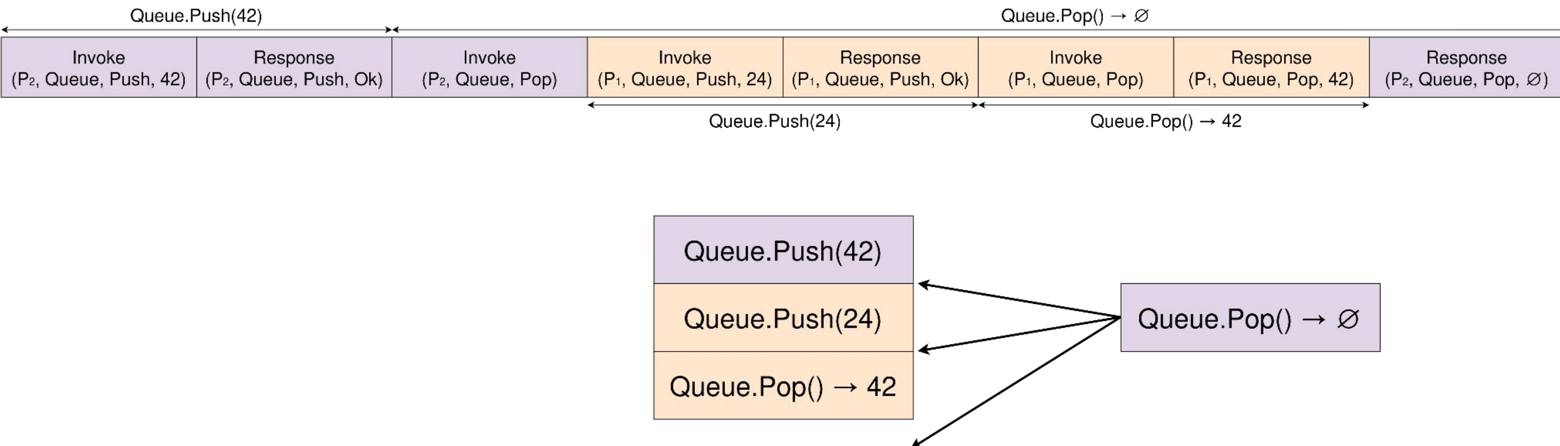
Линеаризуемость

- В последовательности инструкций оставляем только события вызова и завершения каждой операции
- Ищем эквивалентное последовательное исполнение, не нарушающее порядок операций
  - Состоит из тех же операций
  - Очередная операция начинается только после завершения предыдущей
  - Если OP_A закончилась до начала OP_B, в последовательном исполнении должно быть так же



Проверка на линеаризуемость

- Перебираем все варианты чередования операций, не нарушающие порядка операций



Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

Проверка на линеаризуемость

- Перебираем все варианты чередования операций, не нарушающие порядка операций

Queue.Push(42) | Queue.Pop() → ∅

Invoke (P_2, Queue, Push, 42)	Response (P_2, Queue, Push, Ok)	Invoke (P_2, Queue, Pop)	Invoke (P_1, Queue, Push, 24)	Response (P_1, Queue, Push, Ok)	Invoke (P_1, Queue, Pop)	Response (P_1, Queue, Pop, 42)	Response (P_2, Queue, Pop, ∅)

Queue.Push(24) | Queue.Pop() → 42

Queue.Push(42)
Queue.Pop() → ∅
Queue.Push(24)
Queue.Pop() → 42

Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

Проверка на линеаризуемость

- Перебираем все варианты чередования операций, не нарушающие порядка операций

Invoke (P2, Queue, Push, 42)	Response (P2, Queue, Push, Ok)	Invoke (P2, Queue, Pop)	Invoke (P1, Queue, Push, 24)	Response (P1, Queue, Push, Ok)	Invoke (P1, Queue, Pop)	Response (P1, Queue, Pop, 42)	Response (P2, Queue, Pop, ∅)

✗
Queue.Push(42)
Queue.Pop() → ∅
Queue.Push(24)
Queue.Pop() → 42

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → ∅
Queue.Pop() → 42

Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

Проверка на линеаризуемость

- Перебираем все варианты чередования операций, не нарушающие порядка операций

Invoke (P2, Queue, Push, 42)	Response (P2, Queue, Push, Ok)	Invoke (P2, Queue, Pop)	Invoke (P1, Queue, Push, 24)	Response (P1, Queue, Push, Ok)	Invoke (P1, Queue, Pop)	Response (P1, Queue, Pop, 42)	Response (P2, Queue, Pop, ∅)

Queue.Push(42) · Queue.Pop() → ∅ · Queue.Push(24) · Queue.Pop() → 42

✗
Queue.Push(42)
Queue.Pop() → ∅
Queue.Push(24)
Queue.Pop() → 42

✗
Queue.Push(42)
Queue.Push(24)
Queue.Pop() → ∅
Queue.Pop() → 42

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → 42
Queue.Pop() → ∅

Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

- Перебираем все варианты чередования операций, не нарушающие порядка операций

Queue.Push(42) ←→ | Queue.Pop() → ∅ ←→

Invoke (P_2, Queue, Push, 42)	Response (P_2, Queue, Push, Ok)	Invoke (P_2, Queue, Pop)	Invoke (P_1, Queue, Push, 24)	Response (P_1, Queue, Push, Ok)	Invoke (P_1, Queue, Pop)	Response (P_1, Queue, Pop, 42)	Response (P_2, Queue, Pop, ∅)

Queue.Push(24) ←→ | Queue.Pop() → 42 ←→

✗

Queue.Push(42)
Queue.Pop() → ∅
Queue.Push(24)
Queue.Pop() → 42

✗

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → ∅
Queue.Pop() → 42

✗

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → 42
Queue.Pop() → ∅

Queue.Push(24)
Queue.Push(42)
Queue.Pop() → 42
Queue.Pop() → ∅

Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects

Тестирование конкурентного кода

- Тест почему-то успешно проходит

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 || *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 || *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

```
[ RUN      ] concurrent_queue.simple
[       OK ] concurrent_queue.simple (0 ms)
```

Тестирование конкурентного кода

- Не линеаризуемое исполнение можно не найти

P₁	P₂

```
data_ = {42, ∅, ∅, ...}
index_ = 1
```

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

Тестирование конкурентного кода

- t1 целиком завершается до запуска t2

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

Queue.Push(24)

P_1	P_2
data_ = {42, ∅, ∅, ...} index_ = 1	
Invoke(P1, Queue, Push, 24)	
i = index_.fetch_add(1) // 1	
data_ = {42, ∅, ∅, ...} index_ = 2	
data_[1] = 24	
data_ = {42, 24, ∅, ...} index_ = 2	
Response(P1, Queue, Push, Ok)	

Тестирование конкурентного кода

- t1 целиком завершается до запуска t2

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

P1	P2
data_ = {42, ∅, ∅, ...} index_ = 1	
Invoke(P1, Queue, Push, 24)	
i = index_.fetch_add(1) // 1	
data_ = {42, ∅, ∅, ...} index_ = 2	
data_[1] = 24	
data_ = {42, 24, ∅, ...} index_ = 2	
Response(P1, Queue, Push, Ok)	
Invoke(P2, Queue, Pop)	
last = index_.load() // 2	
maybe_result = data_[0].exchange(std::nullopt) // 42	
data_ = {∅, 24, ∅, ...} index_ = 2	
Response(P1, Queue, Pop, 42)	

Queue.Push(24)

Queue.Pop()

Тестирование конкурентного кода

- t1 целиком завершается до запуска t2

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

Queue.Push(24)

Queue.Pop()

<table>
<tr><th>P_1</th><th>P_2</th></tr>
<tr><td colspan="2">data_ = {42, ∅, ∅, ...}
index_ = 1</td></tr>
<tr><td>Invoke(P_1, Queue, Push, 24)</td><td></td></tr>
<tr><td>i = index_.fetch_add(1) // 1</td><td></td></tr>
<tr><td colspan="2">data_ = {42, ∅, ∅, ...}
index_ = 2</td></tr>
<tr><td>data_[1] = 24</td><td></td></tr>
<tr><td colspan="2">data_ = {42, 24, ∅, ...}
index_ = 2</td></tr>
<tr><td>Response(P_1, Queue, Push, Ok)</td><td></td></tr>
<tr><td>Invoke(P_2, Queue, Pop)</td><td></td></tr>
<tr><td>last = index_.load() // 2</td><td></td></tr>
<tr><td>maybe_result = data_[0].exchange(std::nullopt) // 42</td><td></td></tr>
<tr><td colspan="2">data_ = {∅, 24, ∅, ...}
index_ = 2</td></tr>
<tr><td>Response(P_1, Queue, Pop, 42)</td><td></td></tr>
<tr><td></td><td>Invoke(P_2, Queue, Pop)</td></tr>
<tr><td></td><td>last = index_.load() // 2</td></tr>
<tr><td></td><td>maybe_result = data_[0].exchange(std::nullopt) // ∅</td></tr>
<tr><td colspan="2">data_ = {∅, 24, ∅, ...}
index_ = 2</td></tr>
<tr><td></td><td>maybe_result = data_[1].exchange(std::nullopt) // 24</td></tr>
<tr><td colspan="2">data_ = {∅, 24, ∅, ...}
index_ = 2</td></tr>
<tr><td></td><td>Response(P_2, Queue, Pop, 24)</td></tr>
</table>

Тестирование конкурентного кода

Invoke (P2, Queue, Push, 42)	Response (P2, Queue, Push, Ok)	Invoke (P1, Queue, Push, 24)	Response (P1, Queue, Push, Ok)	Invoke (P1, Queue, Pop)	Response (P1, Queue, Pop, 42)	Invoke (P2, Queue, Pop)	Response (P2, Queue, Pop, ∅)

Queue.Push(42) — Queue.Push(24) — Queue.Pop() → 42 — Queue.Pop() → 24

```
TEST(concurrent_queue, simple) {
  queue_t q{3};
  q.Push(42);
  std::thread t1([&q] {
    q.Push(24);
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 ||
           *pop_res == 42);
  });
  std::thread t2([&q] {
    const auto pop_res = q.Pop();
    assert(pop_res.has_value());
    assert(*pop_res == 24 || *pop_res == 42);
  });
  t1.join();
  t2.join();
}
```

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → 42
Queue.Pop() → 24

- У корректного кода все исполнения линеаризуемые

Стресс-тестирование конкурентного кода

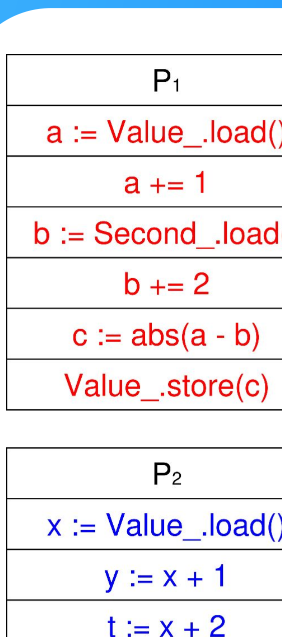
- Иногда может помочь стресс-тестирование
- Планировщик ОС управляет очерёдностью исполнения потоков
    - Ещё контроллер памяти
- Нет детерминированного воспроизведения

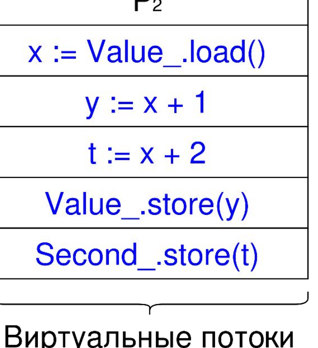
```
[  FAILED  ] 1 test, listed below:
[  FAILED  ] concurrent_queue.stress

 1 FAILED TEST
```

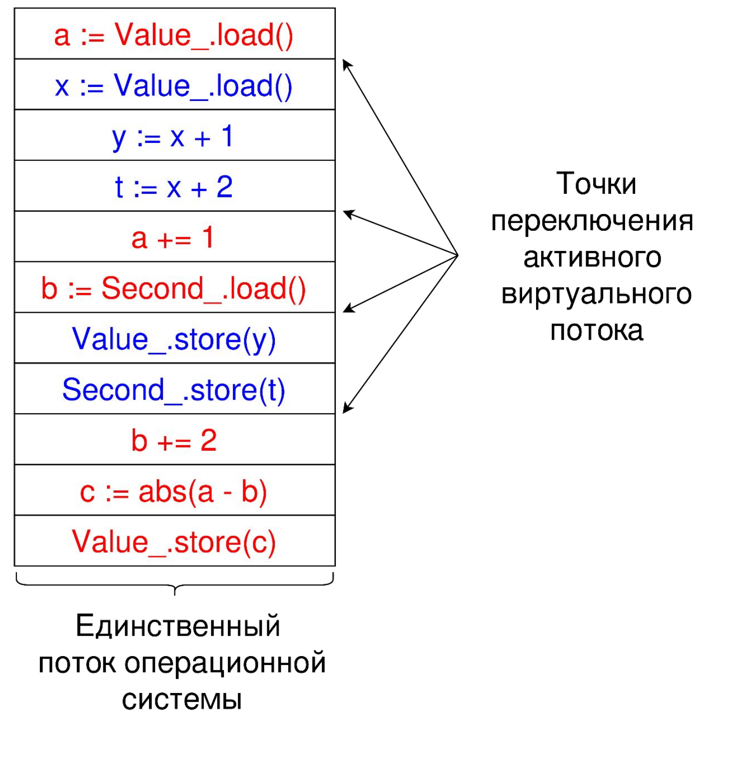
```
TEST(concurrent_queue, stress) {
  for (int32_t i = 0; i < 1'000'000; ++i) {
    queue_t q{3};
    q.Push(42);

    std::thread t1([&q] {
      q.Push(24);
      const auto pop_res = q.Pop();
      assert(pop_res.has_value());
      assert(*pop_res == 24 ||
             *pop_res == 42);
    });
    std::thread t2([&q] {
      const auto pop_res = q.Pop();
      assert(pop_res.has_value());
      assert(*pop_res == 24 ||
             *pop_res == 42);
    });
    t1.join();
    t2.join();
  }
}
```

Принцип тестирования

- Виртуальный поток — последовательность низкоуровневых инструкций
- Каждая такая инструкция — часть операции, исполняемой над структурой данных
- Инструкции чередуются в одном физическом потоке







Полнота и детерминизм

- Можем перебрать все возможные варианты чередования инструкций
- Легко воспроизводим интересные исполнения

P₁
x := A_.load()
B_.store(x + 1)

P₂
A_.store(42)
B_.store(24)

Виртуальные потоки

~~Аналогов нет~~ Аналоги есть: Lincheck

- Оказал на нас огромное влияние

Lincheck

JetBrains official | License MPL 2.0

Lincheck is a practical and user-friendly framework for writing deterministic and robust concurrent tests on JVM. When detecting an error, Lincheck provides a reproducible execution trace an ability to debug it step-by-step in IntelliJ IDEA.

- Поддержка C++ очень экспериментальная
- Только стресс-тесты

Kotlin/Native and C/C++ support postponed

#68 opened on Dec 4, 2020 by Krock21 • Changes requested

~~Аналогов нет~~ Аналоги есть: GenMC

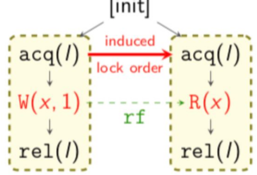
- Очень много продвинутых алгоритмов
- Нет поддержки C++ (только небольшие программы на C)
- Интерпретирует код

GenMC: A model checker for weak memory models

Summary

GenMC is an open-source state-of-the-art model checker for verifying concurrent C/C++ programs under the RC11, IMM, and LKMM memory models.

GenMC is based on a stateless model checking algorithm that is parametric in the choice of memory model. Subject to a few basic conditions about the memory model, our algorithm is sound, complete and optimal, in that it explores each consistent execution of the program according to the model exactly once, and does not explore inconsistent executions or embark on futile exploration paths.



It incorporates many optimizations, such as lock-aware and barrier-aware partial order reduction, symmetry reduction, and automatic spinloop bounding.

~~Аналогов нет~~ Аналоги есть: Twist

- Наконец-то поддержка C++
- Современного C++
- Код приходится менять руками
- Не все нужные нам стратегии перебора исполнений
- Не проверяет линеаризуемость

Twist 🧵

> *He must look to meet whatever events his own fate and the stern Klothes twisted into his thread of destiny when he entered the world and his mother bore him.*
>
> – Homer 🏛️, Odyssey 7.193 (trans. Walter Shewring)

Systematic concurrency testing for modern C++

- LTest хранит несколько виртуальных потоков
- Каждый виртуальный поток содержит инструкции одной операции над структурой данных

```
void f() {
  x := X_.load();
  y := Y_.load();
  T_.store(x + y);
}
```

```
int g() {
  x := X_.load();
  t := T_.load();
  return t - x;
}
```

LTest

```
x := X_.load();
x := X_.load();
y := Y_.load();
t := T_.load();
T_.store(x + y);
return t - x;
```

```
x := X_.load();
y := Y_.load();
x := X_.load();
t := T_.load();
return t - x;
T_.store(x + y);
```

```
x := X_.load();
x := X_.load();
t := T_.load();
return t - x;
y := Y_.load();
T_.store(x + y);
```

...

Архитектура тестирующей системы

- После каждой интересующей нас операции вставляем вызов функции `context_switch()`
  - Возвращает управление в тестирующую систему
- Делаем это на этапе статической предобработки кода

```
void f() {
   x := X_.load();
   y := Y_.load();
   T_.store(x + y);
}
```

→

```
void f() {
   x := X_.load();
   context_switch();
   y := Y_.load();
   context_switch();
   T_.store(x + y);
}
```

Использование корутин

- Изначально управление находится в тестирующей системе

```
n := Index_.load();
context_switch();
for i := 0; i < n; ++n {
  v := Arr_[i].load();
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;
```

```
Index_.store(5);
context_switch();
Arr[0].store(42);
context_switch();
Arr[1].store(24);
context_switch();
/* ... */
```

LTest

Использование корутин

- Управление передаётся пользовательскому коду
- Функция исполняется до первого `context_switch()`

<pre>n := Index_.load();
context_switch();
for i := 0; i < n; ++n {
  v := Arr_[i].load();
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;</pre>
	<pre>Index_.store(5);
	context_switch();
	Arr[0].store(42);
	context_switch();
	Arr[1].store(24);
	context_switch();
	/* ... */</pre>

LTest	

Использование корутин

- Управление возвращается в тестирующую систему
- Можно начать исполнять другой виртуальный поток

```
n := Index_.load();
context_switch();
for i := 0; i < n; ++n {
  v := Arr_[i].load();
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;
```

```
Index_.store(5);
context_switch();
Arr[0].store(42);
context_switch();
Arr[1].store(24);
context_switch();
/* ... */
```

LTest

- Виртуальные потоки обмениваются информацией

```
n := Index_.load(); // n = 5
context_switch();
for i := 0; i < n; ++n {
  v := Arr_[i].load();
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;
```

```
Index_.store(5);
context_switch();
Arr[0].store(42);
context_switch();
Arr[1].store(24);
context_switch();
/* ... */
```

LTest

Использование корутин

- Управление опять
  в тестирующей системе

`n := Index_.load(); // n = 5`
`context_switch();`
`for i := 0; i < n; ++n {`
`  v := Arr_[i].load();`
`  context_switch();`
`  success :=`
`    Arr_[i].compare_exhange(v, v + 1);`
`  context_switch();`
`  if success return true;`
`}`
`return false;`
	`Index_.store(5);`
	`context_switch();`
	`Arr[0].store(42);`
	`context_switch();`
	`Arr[1].store(24);`
	`context_switch();`
	`/* ... */`

LTest	

- Дважды подряд выбираем активным один и тот же виртуальный поток

```
n := Index_.load(); // n = 5
context_switch();
for i := 0; i < n; ++n { // i = 0
  v := Arr_[i].load();   // v = 0
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;
```

```
Index_.store(5);
context_switch();
Arr[0].store(42);
context_switch();
Arr[1].store(24);
context_switch();
/* ... */
```

LTest

Использование корутин

- Управление снова
  в тестирующей системе

<pre>n := Index_.load(); // n = 5
context_switch();
for i := 0; i < n; ++n { // i = 0
  v := Arr_[i].load();   // v = 0
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;</pre>
	<pre>Index_.store(5);
	context_switch();
	Arr[0].store(42);
	context_switch();
	Arr[1].store(24);
	context_switch();
	/* ... */</pre>

LTest	

Использование корутин

- Исполнение продолжается с инструкции следующей за последним исполненным `context_switch()`

``n := Index_.load(); // n = 5
context_switch();
for i := 0; i < n; ++n { // i = 0
  v := Arr_[i].load();   // v = 0
  context_switch();
  success :=
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;```
	``Index_.store(5);
	context_switch();
	Arr[0].store(42);
	context_switch();
	Arr[1].store(24);
	context_switch();
	/* ... */```

LTest	

Использование корутин

- Управление снова
  в тестирующей системе

`n := Index_.load(); // n = 5`
`context_switch();`
`for i := 0; i < n; ++n { // i = 0`
`  v := Arr_[i].load();   // v = 0`
`  context_switch();`
`  success :=`
`    Arr_[i].compare_exhange(v, v + 1);`
`  context_switch();`
`  if success return true;`
`}`
`return false;`
	`Index_.store(5);`
	`context_switch();`
	`Arr[0].store(42);`
	`context_switch();`
	`Arr[1].store(24);`
	`context_switch();`
	`/* ... */`

LTest	

- Arr_[0].CAS(0, 1) проваливается
  - В Arr_[0] записано 42

`n := Index_.load(); // n = 5`
`context_switch();`
`for i := 0; i < n; ++n { // i = 0`
`  v := Arr_[i].load();   // v = 0`
`  context_switch();`
`  success :=  // success = false`
`    Arr_[i].compare_exhange(v, v + 1);`
`  context_switch();`
`  if success return true;`
`}`
`return false;`
	`Index_.store(5);`
	`context_switch();`
	`Arr[0].store(42);`
	`context_switch();`
	`Arr[1].store(24);`
	`context_switch();`
	`/* ... */`

LTest	

Использование корутин

- Снова в тестирующей системе

```
n := Index_.load(); // n = 5
context_switch();
for i := 0; i < n; ++n { // i = 0
  v := Arr_[i].load();   // v = 0
  context_switch();
  success :=  // success = false
    Arr_[i].compare_exhange(v, v + 1);
  context_switch();
  if success return true;
}
return false;
```

```
Index_.store(5);
context_switch();
Arr[0].store(42);
context_switch();
Arr[1].store(24);
context_switch();
/* ... */
```

LTest

Использование корутин

- При следующей передаче управления начнём следующую итерацию цикла

```n := Index_.load(); // n = 5```
```context_switch();```
```for i := 0; i < n; ++n { // i = 1```
``` v := Arr_[i].load();```
``` context_switch();```
``` success :=```
```   Arr_[i].compare_exhange(v, v + 1);```
``` context_switch();```
``` if success return true;```
```}```
```return false;```
	```Index_.store(5);```
	```context_switch();```
	```Arr[0].store(42);```
	```context_switch();```
	```Arr[1].store(24);```
	```context_switch();```
	```/* ... */```

LTest	

Оптимизация корутин

- Обращения к локальным переменным не меняют разделяемого состояния

```
val_cur := A_.load();
B_.store(val_cur);
val_new := val_cur;
++val_new;
++val_new;
C_.store(val_new);
```

→

```
val_cur := A_.load();
context_switch();
B_.store(val_cur);
context_switch();
val_new := val_cur;
context_switch();
++val_new;
context_switch();
++val_new;
context_switch();
C_.store(val_new);
```

B_ = val_cur
C_ = 0

- Вставляем `context_switch()` только перед обращением к разделяемым переменным

```
val_cur := A_.load();
B_.store(val_cur);
val_new := val_cur;
++val_new;
++val_new;
C_.store(val_new);
```

→

```
val_cur := A_.load();
context_switch();
B_.store(val_cur);
context_switch();
val_new := val_cur;
context_switch();   (зачёркнуто)
++val_new;
context_switch();   (зачёркнуто)
++val_new;
context_switch();
C_.store(val_new);
```

B_ = val_cur
C_ = 0

Проверка на линеаризуемость

- В случае SIGABRT/SIGSEGV/SIGFPE/etc нашли ошибку

P1	P2
	Invoke(P2, Queue, Push, 42)
	i = index_.fetch_add(1) // 0
	data_[0] = 42
	Response(P2, Queue, Push, Ok)
	Invoke(P2, Queue, Pop)
	last = index_.load() // 1
Invoke(P1, Queue, Push, 24)	
i = index_.fetch_add(1) // 1	
data_[1] = 24	
Response(P1, Queue, Push, Ok)	
Invoke(P1, Queue, Pop)	
last = index_.load() // 2	
maybe_result = data_[0].exchange(std::nullopt) // 42	
Response(P1, Queue, Pop, 42)	
	maybe_result = data_[0].exchange(std::nullopt) // ∅
	Response(P2, Queue, Pop, ∅)

Проверка на линеаризуемость

- Иначе проверим на линеаризуемость
- Перебираем все варианты чередования операций, не нарушающие порядок операций

Queue.Push(42) | Queue.Pop() → ∅

Invoke (P_2, Queue, Push, 42)	Response (P_2, Queue, Push, Ok)	Invoke (P_2, Queue, Pop)	Invoke (P_1, Queue, Push, 24)	Response (P_1, Queue, Push, Ok)	Invoke (P_1, Queue, Pop)	Response (P_1, Queue, Pop, 42)	Response (P_2, Queue, Pop, ∅)

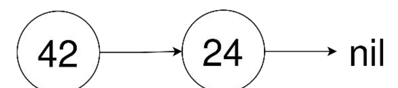
Queue.Push(24) | Queue.Pop() → 42

Queue.Push(42)	Queue.Push(42)	Queue.Push(42)	Queue.Push(24)
Queue.Pop() → ∅	Queue.Push(24)	Queue.Push(24)	Queue.Push(42)
Queue.Push(24)	Queue.Pop() → ∅	Queue.Pop() → 42	Queue.Pop() → 42
Queue.Pop() → 42	Queue.Pop() → 42	Queue.Pop() → ∅	Queue.Pop() → ∅

Проверка на линеаризуемость

- Можно ли быстрее?
- Используем кеш состояний для ускорения применения общих префиксов

Queue.Push(42)
Queue.Push(24)
Queue.Pop() → ∅
Queue.Pop() → 42



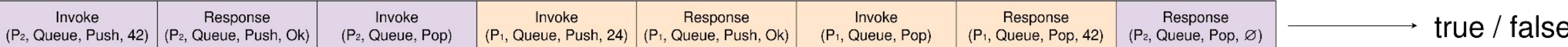
Queue.Push(42)
Queue.Push(24)
Queue.Pop() → 42
Queue.Pop() → ∅

Wing J. M., Gong C. Testing and verifying concurrent objects

Lowe G. Testing for linearizability

Проверка на линеаризуемость

- Экспоненциальный алгоритм
- Задача проверки одного исполнения на линеаризуемость в общем случае NP-полна
  - *Gibbons P. B., Korach E. Testing shared memories*
- Для некоторых структур данных можно решить за полиномиальное время
  - *Emmi M., Enea C. Sound, complete, and tractable linearizability monitoring for concurrent collections*
  - Ручная проверка
- std::vector<std::variant<Invoke, Response>>



true / false

Интерлюдия: тестируем userver

- Тестируем

```
userver::concurrent::impl::IntrusiveStack
```

Интерлюдия: тестируем userver

- Внутри стек Трайбера
- *Treiber R. K. et al. Systems programming: Coping with parallelism*

```
void Push(T& node) noexcept {
    NodeTaggedPtr expected = stack_head_.load();
    while (true) {
        GetNext(node).store(expected.GetDataPtr());
        const NodeTaggedPtr desired(
            &node, expected.GetTag());
        if (stack_head_.compare_exchange_weak(
                expected, desired)) {
            break;
        }
    }
}
```

```
T* TryPop() noexcept {
    NodeTaggedPtr expected = stack_head_.load();
    while (true) {
        T* const expected_ptr = expected.GetDataPtr();
        if (!expected_ptr) return nullptr;
        const NodeTaggedPtr desired(
            GetNext(*expected_ptr).load(), expected.GetNextTag());
        if (stack_head_.compare_exchange_weak(expected, desired)) {
            GetNext(*expected_ptr).store(
                nullptr, std::memory_order_relaxed);
            return expected_ptr;
        }
    }
}
```

Интерлюдия: тестируем userver

- Пишем обёртку для интрузивного стека
  - Дополнительно содержит хранилище элементов

```
struct BoxInt {
   BoxInt(int v) : x{v} {}

   concurrent::impl::
     SinglyLinkedHook<BoxInt>
     stack_hook;
   int x;
};
```

```
struct IntrusiveStack {
public:
  IntrusiveStack() {
    for (size_t i = 0; i < 100; ++i)
      nodes.emplace_back(i);
  }

  non_atomic int Push(size_t index) { /* ... */ }

  non_atomic int TryPop() { /* ... */ }

private:
  std::deque<BoxInt> nodes{};
  concurrent::impl::IntrusiveStack<
    BoxInt,
    concurrent::impl::MemberHook<
      &BoxInt::stack_hook>>
  intrusive_stack{};
};
```

Интерлюдия: тестируем userver

- Push вставляет в интрузивный стек i-ый элемент из хранилища
- Pop удаляет элемент из стека и возвращает индекс удалённого элемента в хранилище

```
non_atomic int Push(size_t index) {
   intrusive_stack.Push(nodes[index]);
   return 0;
}

non_atomic int TryPop() {
   auto res = intrusive_stack.TryPop();
   return res ? res->x : -1;
}
```

- Пишем hash code и equals для тестируемой структуры
    - Чтобы заработал кэш состояний
    - Без них работать будет дольше

```
struct IntrusiveStackHash {
  size_t operator()(const IntrusiveStack &r) const {
    int res = 0;
    for (int elem : r.intrusive_stack) res += elem;
    return res;
  }
};

struct IntrusiveStackEquals {
  template <typename PushArgTuple, int ValueIndex>
  bool operator()(const IntrusiveStack &lhs,
                  const IntrusiveStack &rhs) const {
    return lhs.intrusive_stack == rhs.intrusive_stack;
  }
};

using spec_t = ltest::Spec<IntrusiveStack,
                           IntrusiveStack,
                           IntrusiveStackHash,
                           IntrusiveStackEquals>;
```

- Отмечаем, какую структуру данных тестируем и какие у неё методы
  - И как генерировать аргументы каждого из методов

```
auto generateInt([[maybe_unused]] size_t arg_size) {
   static int a = 0;
   if (a == 100) a = 0;
   return ltest::generators::makeSingleArg(a++);
}

LTEST_ENTRYPOINT(spec_t);
target_method(generateInt, void, IntrusiveStack, Push, int);
target_method(ltest::generators::genEmpty, int, IntrusiveStack, TryPop);
```

Интерлюдия: тестируем userver

- Компилируем clang -fpass-plugin
- Расставляет точки переключения контекста
- Поддержка thread-local, etc...

```
; Function Attrs: mustprogress noinline nounwind optnone sanitize_address uwtable
define linkonce_odr hidden void
@_ZN7userver8v2_11_rc10concurrent4impl14IntrusiveStackINS0_6BoxIntENS2_10MemberHookIXadL_ZNS4_10stack_hookEEEEEE4Pus
hERS4_(ptr noundef nonnull align 8 dereferenceable(8) %this, ptr noundef nonnull align 8 dereferenceable(12) %node)
#2 comdat align 2 personality ptr @__gxx_personality_v0 !dbg !27806 {
entry:
  %this.addr = alloca ptr, align 8
  %node.addr = alloca ptr, align 8
  %agg.tmp = alloca %"class.std::basic_string_view", align 8
  %agg.tmp8 = alloca %"class.std::basic_string_view", align 8
  %expected = alloca %"class.userver::v2_11_rc::concurrent::impl::TaggedPtr", align 8
  %desired = alloca %"class.userver::v2_11_rc::concurrent::impl::TaggedPtr", align 8
  %agg.tmp25 = alloca %"class.userver::v2_11_rc::concurrent::impl::TaggedPtr", align 8
  %cleanup.dest.slot = alloca i32, align 4
  store ptr %this, ptr %this.addr, align 8
    #dbg_declare(ptr %this.addr, !27807, !DIExpression(), !27809)
  call void @context_switch()
  store ptr %node, ptr %node.addr, align 8
    #dbg_declare(ptr %node.addr, !27810, !DIExpression(), !27811)
  call void @context_switch()
 ...
```

Интерлюдия: тестируем userver

- Указываем параметры тестирования
  - Сколько исполнений перебирать, как перебирать исполнения, etc
- Наслаждаемся результатом!

```
⚡ root  utest  develop ● ?  ✘   ./stack-ltest-no-asan --tasks 40 --rounds 1000 --strategy pct --minimize
verbose: false
threads  = 2
tasks    = 40
switches = 100000000
rounds   = 1000
minimize = true
exploration runs = 15
minimization runs = 15
targets  = 2
strategy = pct


success!
```

Интерлюдия: тестируем userver

- В случае ошибки получаем проблемное исполнение
- Которое можно многократно воспроизводить в целях отладки

```
12: Test command: /__w/LTest/LTest/build/verifying/targets/nonlinear_queue_with_custom_round "--rounds" "0" "--
exploration_runs" "100000" "--strategy" "random"
12: Working Directory: /__w/LTest/LTest/build/verifying/targets
12: Test timeout computed to be: 1500
12: verbose: false
12: threads  = 2
12: tasks    = 15
12: switches = 100000000
12: rounds   = 0
12: minimize = false
12: targets  = 0
12: strategy = random
12:
12:
12: non linearized:
12: *--------------------------------------------------*--------------------------------------------------*
12: |                        T0                        |                        T1                        |
12: *--------------------------------------------------*--------------------------------------------------*
12: |                                                  | [3] Push(3)                                      |
12: | [0] Push(1)                                      |                                                  |
12: |                                                  | <-- void                                         |
12: |                                                  | [4] Pop()                                        |
12: | <-- void                                         |                                                  |
12: | [1] Push(2)                                      |                                                  |
12: | <-- void                                         |                                                  |
12: | [2] Pop()                                        |                                                  |
12: | <-- 3                                            |                                                  |
12: |                                                  | <-- 0                                            |
12: *--------------------------------------------------*--------------------------------------------------*
```

Действие второе:
жить сложно

Как тестировать большие программы

Как тестировать структуры данных, использующие слабые memory order’ы

Как тестировать программы, которые используют блокировки

Действие второе:
жить сложно

Как тестировать большие программы

Нужно лучше понимать, что мы перебираем

Interleavings vs states

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_.store(42)
B_.store(24)
x := A_.load()
B_.store(x + 1)
A_.store(x + 2)

A_ = 44
B_ = 43

A_.store(42)
x := A_.load()
B_.store(24)
B_.store(x + 1)
A_.store(x + 2)

A_ = 44
B_ = 43

State space

- Нас интересуют разные состояния нашей программы
- Граф*
- Вершины — состояния

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_ = 0
B_ = 0

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_ = 42
B_ = 0

*на самом деле это LTS (labeled transition system)

State space

- Нас интересуют разные состояния нашей программы
- Граф*
- Вершины — состояния
- Ребро между вершинами — исполнение инструкции
- Исполнение — путь в графе

*на самом деле это LTS (labeled transition system)

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_ = 0
B_ = 0

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_ = 42
B_ = 0

State space

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

A_ = 0
B_ = 0

x := A_.load();
B_.store(x + 1)

A_ = 0
B_ = 1

A_.store(42)

A_ = 42
B_ = 0

State space explosion

- Ввели формальную модель
- Состояний экспоненциально много
- Исполнений еще больше
- Два типа алгоритмов: Model checking и Controlled Concurrency Testing (CCT)
- Хотим перебрать все состояния и проверить на них свойство
- Хотим перебрать часть состояний и проверить на них свойство

Model checking

Обходим граф всех возможных состояний при помощи DFS

Каждое рассмотренное исполнение проверяется на корректность

Model checking

P_1	P_2	P_3
A.store(1)	B.store(1)	a :=A.load()
		b := B.load()

P_1	P_2	P_3
A.store(1)	B.store(1)	a :=A.load()
		b := B.load()

A_ = 0
B_ = 0

A.store(1)

A_ = 1
B_ = 0

a :=A.load()

...

B.store(1)

a :=A.load()

...

...

B.store(1)

P_1	P_2	P_3
A.store(1)	B.store(1)	a :=A.load()
		b := B.load()

A_ = 1
B_ = 1

a :=A.load()

...

Уменьшение количества исполнений

- Неоптимально
- Пробуем два пути вместо одного

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 0

B_.store(10)

A_.store(42)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 10

A_.store(20) …

A_.store(42)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 42
B_ = 0

B_.store(10)

B_.store(24) …

A_ = 42
B_ = 10

A_.store(20) …

B_.store(24) …

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

Уменьшение количества исполнений

- Две операции из разных потоков могут коммутировать
- Исполнение OP1; OP2; приводит в то же состояние, что и OP2; OP1;

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 0

B_.store(10)

A_ = 0
B_ = 10

A_.store(20)

...

A_.store(42)

A_.store(42)

A_ = 42
B_ = 0

B_.store(10)

B_.store(24)

...

A_ = 42
B_ = 10

A_.store(20)

...

B_.store(24)

...

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

Уменьшение количества исполнений

- Две читающие операции коммутируют всегда

```
A_.load(seq_cst); B_.load(seq_cst); коммутируют
```

```
A_.load(seq_cst); A_.load(seq_cst); коммутируют
```

Уменьшение количества исполнений

- Пишущая операция коммутирует только с операцией по другому регистру

```
A_.store(x, seq_cst); B_.load(seq_cst);
```
коммутируют

```
A_.store(x, seq_cst); B_.store(y, seq_cst);
```
коммутируют

```
A_.store(x, seq_cst); A_.load(seq_cst);
```
не коммутируют

```
A_.store(x, seq_cst); A_.store(y, seq_cst);
```
не коммутируют

- Не коммутирующие операции называются конфликтующими

Уменьшение количества исполнений

- `A_.store(x); A_.store(y);` не коммутируют

P_1	P_2
A_.store(24)	A_.store(42)
...	...

P_1	P_2
A_.store(24)	A_.store(42)
...	...

P_1	P_2
A_.store(24)	A_.store(42)
...	...

P_1	P_2
A_.store(24)	A_.store(42)
...	...

P_1	P_2
A_.store(24)	A_.store(42)
...	...

Определение коммутирующих операций

- Делаем `context_switch` перед каждым обращением к разделяемой переменной
- Перед возвращением управления тестирующей системе запоминаем тип обращения к памяти, которая должна произойти следующей
  - Адрес переменной, к которой произойдёт обращение
  - Размер переменной, memory order, etc

Определение коммутирующих операций

```
const int32_t i = calculate_index();
Array_[i].store(42);
```

```
const int32_t i = calculate_index();
G_Scheduler_State_->OpType = op_types_t::WRITE;
G_Scheduler_State_->Addr = &Array_[i];
context_switch();
Array_[i].store(42);
```

Partial order reduction

- Для достижения оптимальности должны учитывать коммутативность
- Dynamic Partial Order Reduction (DPOR)

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

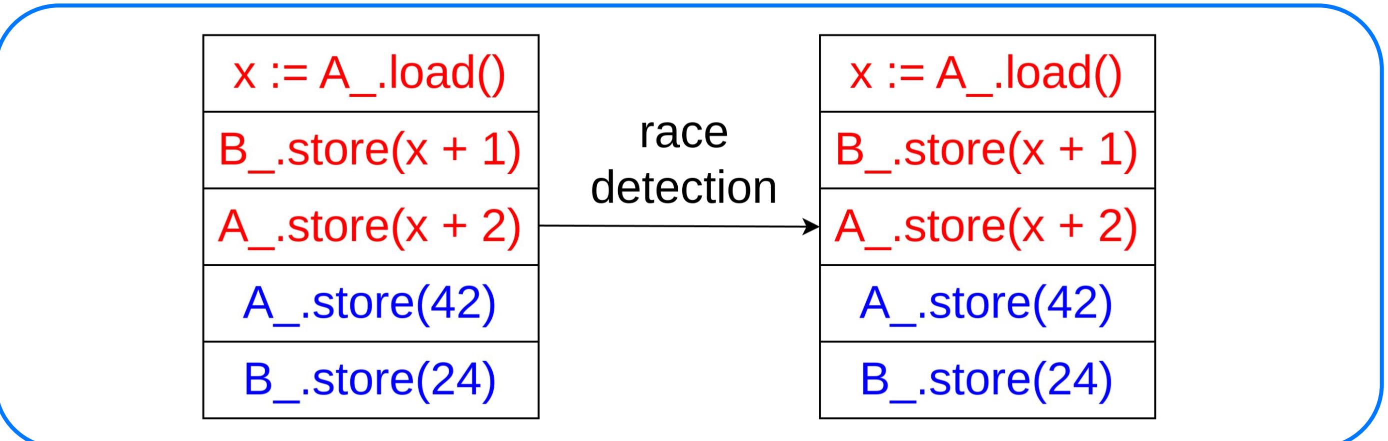
Partial order reduction

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

x := A_.load()
B_.store(x + 1)
A_.store(x + 2)
A_.store(42)
B_.store(24)

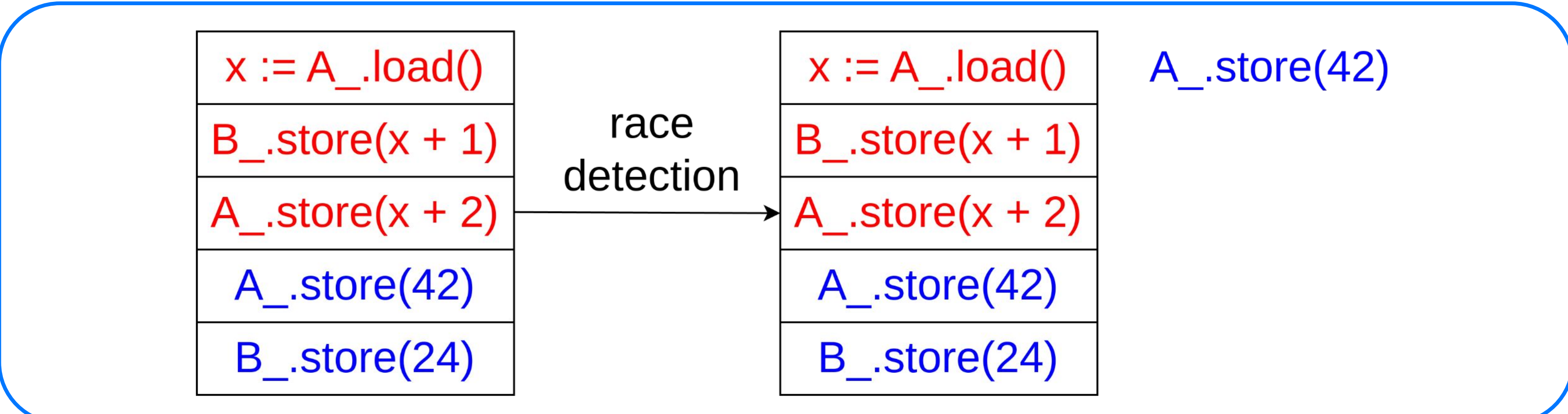
Partial order reduction

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	



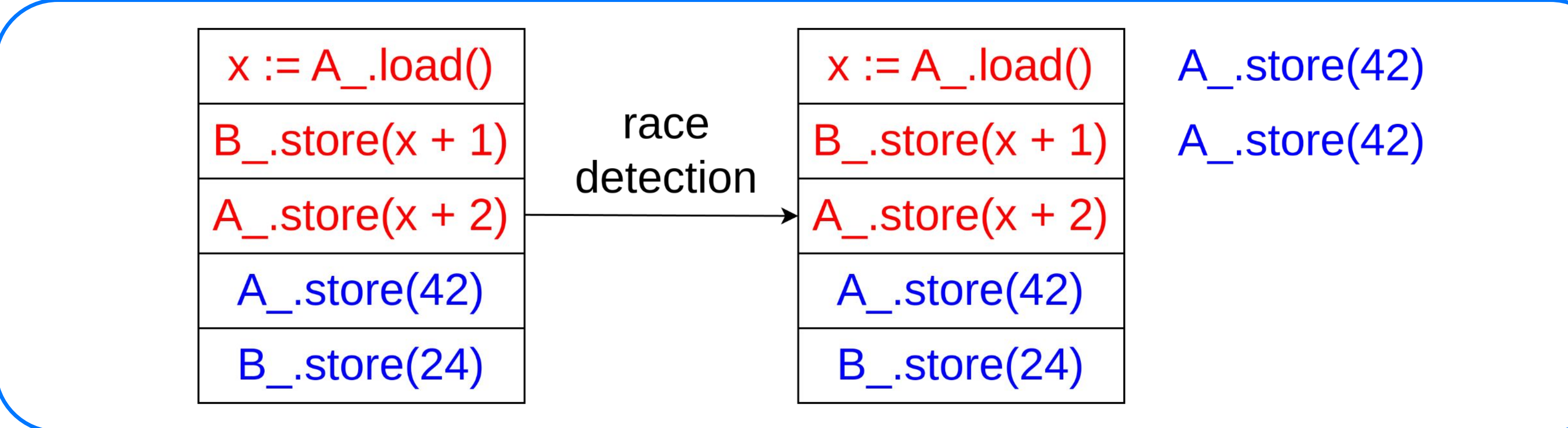
Partial order reduction

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	



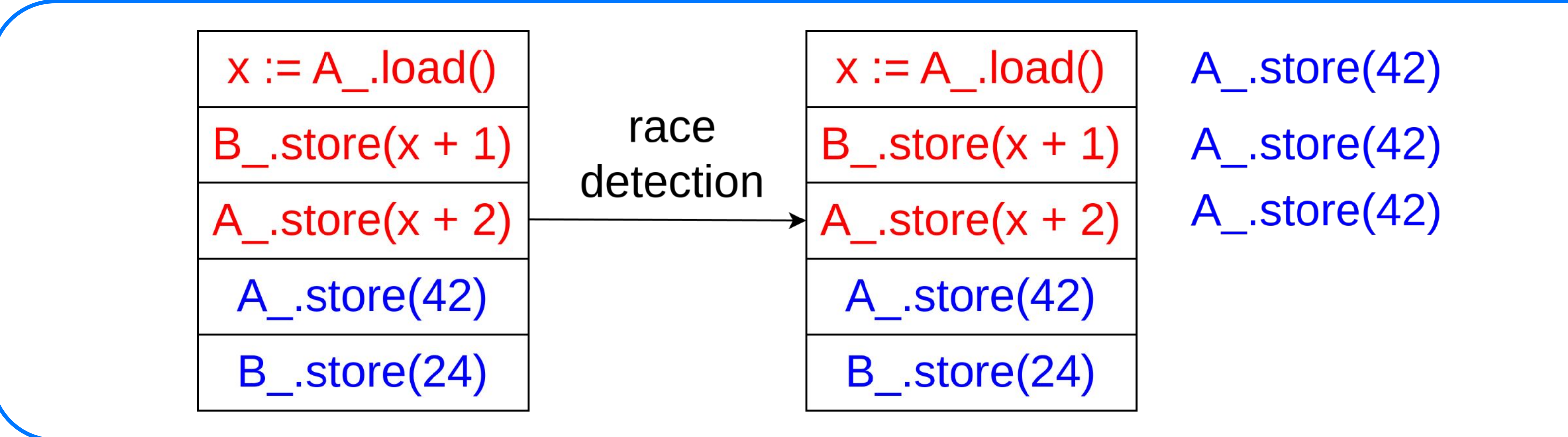
Partial order reduction

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	



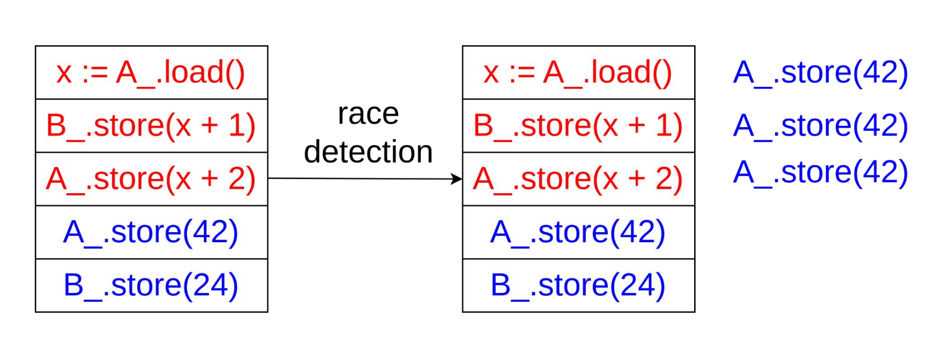
Partial order reduction

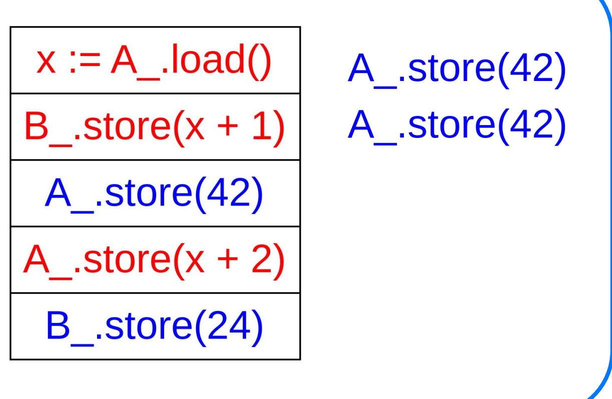
P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	



Partial order reduction

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	





State space explosion

Оптимально

Все еще долго для реальных программ

А если время на тесты ограничено?

Controlled Concurrency Testing (CCT)

Хотим обойти часть state space

Хотим зайти в каждое состояние с одинаковой вероятностью

Семплинг исполнений

- Случайный выбор

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

P_1	P_2
x := A_.load()	A_.store(42)
B_.store(x + 1)	B_.store(24)
A_.store(x + 2)	

Семплинг исполнений

- Подкидывание монетки
  на каждом шаге работает плохо

```
std::atomic<object_t*> Ptr_{
  new object_t{.field = 0}
};
std::atomic_int32_t Value_{42};
```

T1	T2
++Value_;	Ptr_.load()->field = 42;
...	
++Value_;	
Ptr_.store(nullptr);	

k

Семплинг исполнений

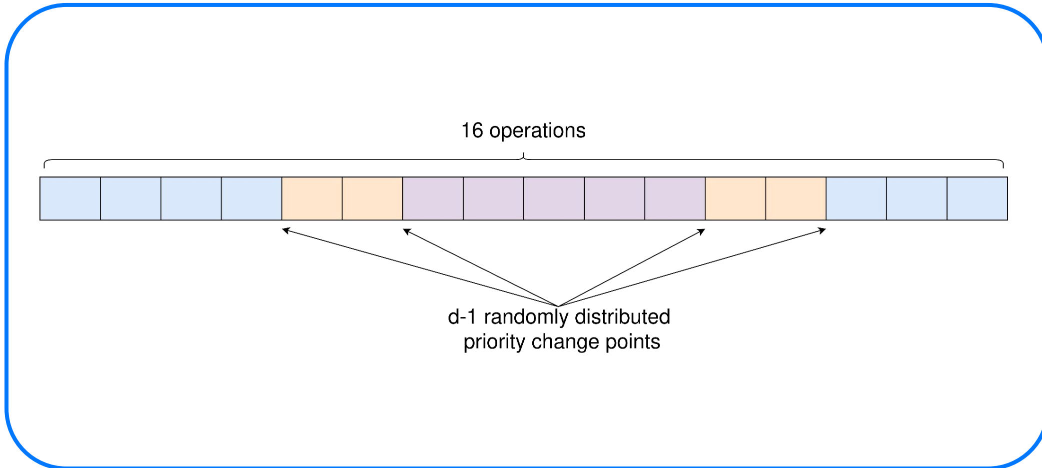
- Вероятность найти баг уменьшается экспоненциально

At the outset, it may seem impossible to provide effective probabilistic guarantees without exercising an astronomical number of schedules. Let us take a program with n threads that together execute at most k instructions. This program, to the first-order of approximation, has n^k possible thread schedules. If an adversary picks any one of these schedules to be the only buggy schedule, then no randomized scheduler can find the bug with a probability greater than $1/n^k$. Given that realistic programs create tens of threads (n) and can execute millions, if not billions, of instructions (k), such a bound is not useful.

```
++Value_;
++Value_;
++Value_;
...
++Value_;
Ptr_.store(nullptr);
Ptr_.load()→field = 42;
```

Семплинг исполнений: PCT

- Выбираем заранее количество шагов, сколько проработает каждый поток до смены активного потока
- Вероятность найти баг уменьшается полиномиально



Given inputs n, k, and d, PCT works as follows.

1. Assign the n priority values $d, d+1, \ldots, d+n$ randomly to the n threads (we reserve the lower priority values $1, \ldots, (d-1)$ for change points).
2. Pick $d-1$ random priority change points $k_1, \ldots, k_{d-1}$ in the range $[1, k]$. Each k_i has an associated priority value of i.
3. Schedule the threads by honoring their priorities. When a thread reaches the i-th change point (that is, when it executes the k_i-th step of the execution), change the priority of that thread to i.

This randomized scheduler provides the following guarantee.

Given a program that creates at most n threads and executes at most k instructions, PCT finds a bug of depth d with probability at least $1/nk^{d-1}$.

Burckhardt S. et al. A randomized scheduler with probabilistic guarantees of finding bugs

Семплинг исполнений

- Снова не учитываем коммутативность

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 0

B_.store(10)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 10

A_.store(42)

A_.store(42)

A_ = 42
B_ = 0

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

B_.store(10)

A_ = 42
B_ = 10

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_.store(20)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 20
B_ = 10

B_.store(24)

B_.store(24)

A_ = 42
B_ = 24

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_.store(20)

A_ = 20
B_ = 24

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

Семплинг исполнений

- Снова не учитываем коммутативность
- Попробуем учесть

Yuan X., Yang J., Gu R. Partial order aware concurrency sampling

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 0

B_.store(10)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 0
B_ = 10

A_.store(42)

A_.store(42)

A_ = 42
B_ = 0

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

B_.store(10)

A_ = 42
B_ = 10

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_.store(20)

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_ = 20
B_ = 10

B_.store(24)

B_.store(24)

A_ = 42
B_ = 24

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

A_.store(20)

A_ = 20
B_ = 24

P_1	P_2
B_.store(10)	A_.store(42)
A_.store(20)	B_.store(24)

Семплинг исполнений

- Снова не учитываем коммутативность
- Попробуем учесть
- Partial Order Sampling (POS)
- Распределим приоритеты инструкциям
- Будем менять приоритеты, конфликтующим операциям, когда исполним одну из них

Итог

Поговорили, как тестировать большие программы

А если у нас есть более слабые memory order’ы?

Поддержка слабых моделей памяти

- Может ли программа упасть по ассёрту?

```
std::atomic_int32_t X_{0}, Y_{0};

void thread1() noexcept {
  X_.store(1, std::memory_order_relaxed);
  Y_.store(1, std::memory_order_relaxed);
}

void thread2() const noexcept{
  int y = Y_.load(std::memory_order_relaxed);
  int x = X_.load(std::memory_order_relaxed);
  assert(!(y == 1 && x == 0));
}
```

Поддержка слабых моделей памяти

- Может ли программа упасть по ассёрту?
- Может
  - Но не при исполнении в единственном потоке ОС

```
std::atomic<int> X{0}, Y{0};
X.store(1, std::memory_order_relaxed);
Y.store(1, std::memory_order_relaxed);
int y = Y.load(std::memory_order_relaxed);
int x =X.load(std::memory_order_relaxed);
assert(!(y == 1 && x == 0));
```

Поддержка слабых моделей памяти

- Каждая переменная хранит историю своих изменений
- Для каждого чтения определяем диапазон возможных прочитанных значений и возвращаем одно из них

```
struct MpLitmusTest {
  std::atomic_int32_t X_{0}, Y_{0};

  void thread1() noexcept {
    X_.store(1, std::memory_order_relaxed);
    Y_.store(1, std::memory_order_relaxed);
  }

  void thread2() const noexcept{
    int y = Y_.load(std::memory_order_relaxed);
    int x = X_.load(std::memory_order_relaxed);
    assert(!(y == 1 && x == 0));
  }
};
```

test failed: !(y == 1 && x == 0) at /Ltest/verifying/targets/wmm_litmus/__tmp_litmus_mp.cpp:19

Примитивы синхронизации

А если мой код использует блокировки?

Поддержка примитивов синхронизации

- Тестируемый код может использовать примитивы синхронизации
  - Блокировки, барьеры, семафоры, …

```
Mutex_.Lock()
```

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- Первый поток зашёл в критическую секцию

Mutex_.Lock()
value += 1

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- Сменился активный поток
- Второй поток пытается зайти в критическую секцию

Mutex_.Lock()
value += 1
Mutex_.Lock()

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- Undefined behvaiour в случае нереентерабельной блокировки
- В лучшем случае получим быстрый сбой

Mutex_.Lock()
value += 1
Mutex_.Lock()

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- Может случиться зависание
- Поток пытается взять блокировку, но не может, потому что сам её держит

```
Mutex_.Lock()
value += 1
Mutex_.Lock()
```

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- В случае реентерабельной блокировки два виртуальных потока одновременно в критической секции

Mutex_.Lock()
value += 1
Mutex_.Lock()
value += 1

```
struct counter {
  void Inc() {
    mutex_.lock();
    ++data_;
    mutex_.unlock();
  }
private:
  std::mutex mutex_;
  int32_t data_{0};
};
```

Поддержка примитивов синхронизации

- На этапе компиляции подменим примитив синхронизации на его mock-версию
- Тот же интерфейс, но правильное взаимодействие с LTest

Поддержка примитивов синхронизации

- Мьютекс знает, что он взят

LTest

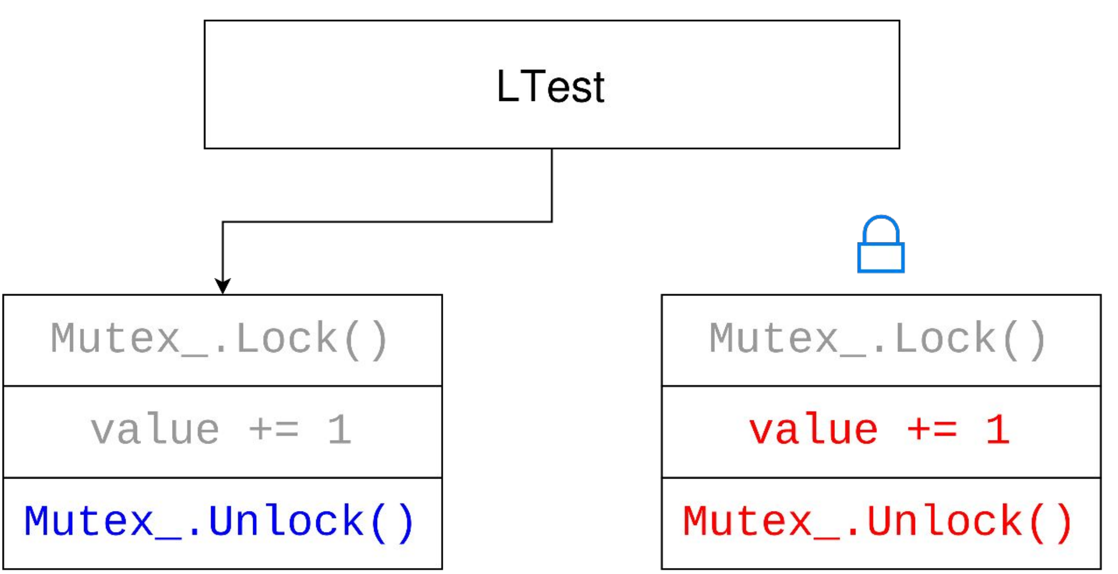
Mutex_.Lock()
value += 1
Mutex_.Unlock()

Mutex_.Lock()
value += 1
Mutex_.Unlock()

Mutex_.Lock()
value += 1

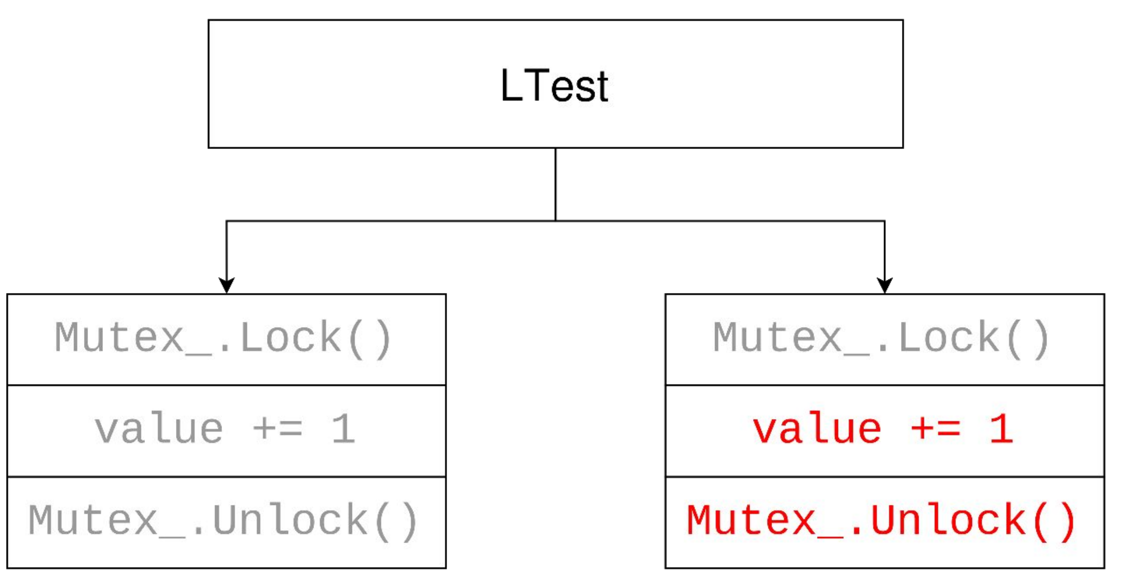
Поддержка примитивов синхронизации

- При попытке повторного взятия помечает виртуальный поток как заблокированный
- Заблокированный поток не может стать активным



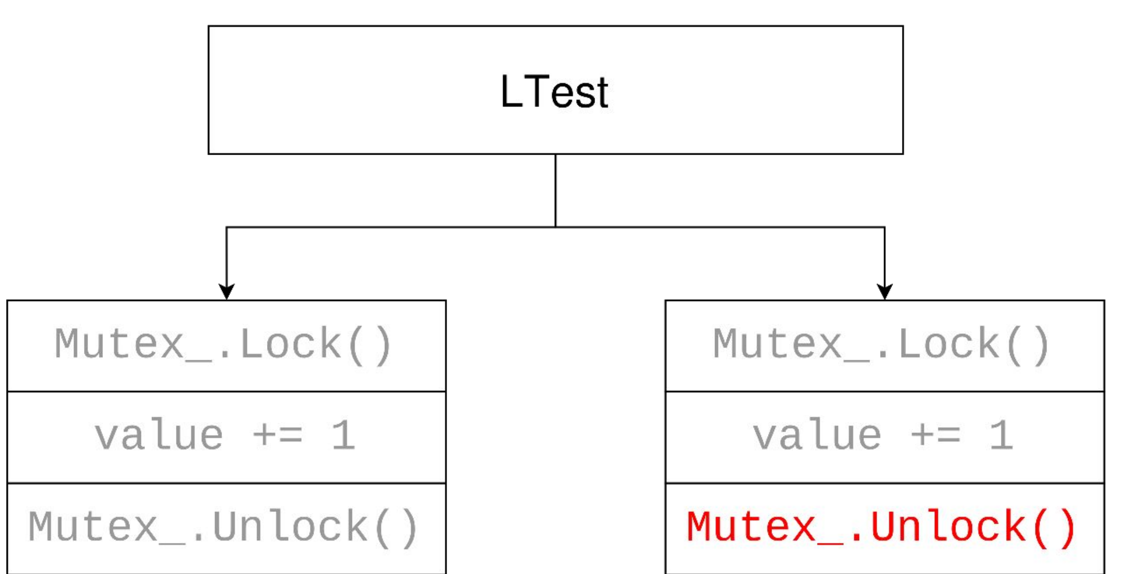
Поддержка примитивов синхронизации

- После отпускания блокировки разблокируем всех ожидающих
- Снова можем исполнять их инструкции



Поддержка примитивов синхронизации

- Нужны подобные mock-версии всех используемых примитивов



Библиография

- Truly stateless, optimal dynamic partial order reduction
- A randomized scheduler with probabilistic guarantees of finding bugs
- Partial Order Aware Concurrency Sampling
- Selectively Uniform Concurrency Testing
- Repairing sequential consistency in C/C++11
- A Practical Approach for Model Checking C/C++11 Code

Старший разработчик
в команде баз данных

vk.com/rpc

t.me/ilyambda

Разработчик в команде
core infrastructure

vk.com/svilex

t.me/lim123123123

github.com/ITMO-PTDC-Team/LTest

Thanks for your attention

my dudes