C++20 · C++23 · C++26

Алексей Примак

Многопоточность на C++

через практику и замеры

От первого потока до архитектуры уровня senior

bench.cpp C++

```cpp
int main() {
    std::atomic<std::uint64_t> served{0};
    {
        std::vector<std::jthread> workers;
        for (int i = 0; i < 8; ++i)
            workers.emplace_back([&](std::stop_token st) {
                while (!st.stop_requested())
                    served.fetch_add(1, std::memory_order_relaxed);
            });
        std::this_thread::sleep_for(1s);
    }   // ~jthread: request_stop() и join()
    std::println("{} ops/s", served.load());
}
```

```
$ ./bench
70032093 ops/s
```

42 задачи с разбором решений · весь код проверен под ThreadSanitizer и AddressSanitizer

Алексей Примак

Многопоточность на C++ через практику и замеры

«Автор»
2026

Примак А.

Многопоточность на C++ через практику и замеры / А. Примак — «Автор», 2026

«У меня локально работало». Последние слова перед инцидентом.Практикум по многопоточности на современном C++ (C++20, C++23, обзор C++26) для тех, кто уже пишет на C++ и хочет дойти до уровня senior. Каждое «так быстрее» здесь подтверждено замером, каждое «так безопасно» — тестом под ThreadSanitizer. Весь код собран и проверен.Сквозной проект — key-value сервер shardkv: за 14 шагов он проходит путь от потока на соединение до асинхронного сервера на корутинах и std::execution.В книге: потоки, мьютексы, дедлоки, condition variables; кеши и модель памяти C++; атомики и lock-free структуры; hazard pointers и RCU; пулы потоков, корутины и senders/receivers; архитектура под нагрузкой и отладка зависаний. В каждой главе — вопросы с собеседований и задачи, все 42 задачи разобраны в конце книги.Для тех, кто готовится к собеседованию на senior C++.

© Примак А., 2026
© Автор, 2026

Содержание

Как читать эту книгу 6
Глава 1. std::thread, std::jthread и stop_token: как запустить поток и не вызвать std::terminate 7
1.1. Что такое поток и сколько он стоит 8
1.2. join, detach и std::terminate 10
Почему detach() — почти всегда ошибка 11
1.3. Исключения и join: почему «просто не забыть» не работает 12
1.4. std::jthread: поток, который умеет за собой убирать 14
1.5. Передача аргументов: копии, ссылки и ловушки 15
1.6. Кооперативная отмена: stop_token, stop_source, stop_callback 18
1.7. Исключения в потоке 21
1.8. thread_local и hardware_concurrency 23
1.9. Завершение программы при живых потоках 24
1.10. Грабли с ревью 25
1.11. Что спросят на собеседовании 26
1.12. Сквозной проект shardkv, шаг 1: поток на соединение 27
Как устроена остановка 28
Ключевой код 29
Сборка, запуск, проверка 36
Тизер главы 2 37
Полный код шага 1 38
Задачки для разминки мозгов 58
Задача 1.1 (лёгкая). Найди три бага 59
Задача 1.2 (средняя). parallel_sum с доставкой исключений 60
Задача 1.3 (сложная). Сервис, который нельзя сломать деструктором 61
Глава 2. Data race, мьютексы и тот самый deadlock 62
2.1. Data race: почему это UB, а не «иногда не то значение» 63
Data race и race condition — разные вещи 65
2.2. std::mutex и RAII-обёртки 66
Сколько стоит мьютекс 68
2.3. Защищайте данные, а не код 69
2.4. Потокобезопасный интерфейс: когда каждый метод безопасен, а класс сломан 72
2.5. Deadlock: четыре условия и как сломать хотя бы одно 75
2.6. shared_mutex: много читателей, редкий писатель 77
2.7. recursive_mutex — запах кода 79
2.8. Таймауты: timed_mutex и правильные часы 80
2.9. Однократная инициализация 83
2.10. Что потокобезопасно в стандартной библиотеке 84
2.11. ThreadSanitizer: первый запуск 86
2.12. Грабли с ревью 87
2.13. Что спросят на собеседовании 88
2.14. Сквозной проект shardkv, шаг 2: глобальный мьютекс и первый бенчмарк 89
Новое хранилище 90
Стресс-тесты 94

kvbench, версия 1 97
Первые замеры 101
Полный код шага 2 102
Задачки для разминки мозгов 104
Задача 2.1 (лёгкая). Банк, который зависает 105
Задача 2.2 (средняя). ThreadSafeStack без гонок в интерфейсе 107
Задача 2.3 (сложная). hierarchical_mutex 108
Глава 3. Condition variables и потокобезопасная очередь 109
3.1. Как ждать правильно 110
3.2. Три способа ошибиться 113
Ошибка 1: wait без предиката и spurious wakeup 113
Ошибка 2: lost wakeup — уведомление, которое никто не услышал 114
Ошибка 3: разные мьютексы или состояние вне мьютекса 115
3.3. notify_one, notify_all и где их вызывать 116
3.4. Ожидание с таймаутом: wait_for и wait_until 117
3.5. condition_variable_any и stop_token 119
3.6. Блокирующая очередь 120
3.7. Graceful shutdown: как остановить систему потоков 124
3.8. Грабли с ревью 125
3.9. Что спросят на собеседовании 126
3.10. Сквозной проект shardkv, шаг 3: TTL и поток, который спит ровно сколько нужно 127
Как устроен фоновый поток 128
Код хранилища 129
Протокол и запуск 138
Тесты 139
Замеры 143
Задачки для разминки мозгов 144
Задача 3.1 (лёгкая). Потерянное пробуждение 144
Задача 3.2 (средняя). BoundedQueue 146
Задача 3.3 (средняя). Пинг-понг и хоровод 147
Глава 4. Железо и ОС: что на самом деле происходит под std::mutex 148
4.1. Кеши и кеш-линии 149
4.2. Когерентность кешей и false sharing 151
hardware_destructive_interference_size 154
Где искать false sharing 155
4.3. Store buffer: почему процессор «переставляет» операции 156
4.4. Как устроен мьютекс в Linux: futex 157
4.5. Законы Амдала и Густафсона 159
4.6. NUMA 160
4.7. Планировщик, приоритеты и привязка к ядрам 161
4.8. fork() в многопоточном процессе 163
4.9. Сигналы и потоки 164
Правильный шаблон: выделенный поток сигналов 165
4.10. Грабли с ревью 168
4.11. Что спросят на собеседовании 169
Конец ознакомительного фрагмента. 170

Многопоточность на C+ + через практику и замеры

Как читать эту книгу

Эта книга — для разработчика, который уже пишет на C++, знает RAII, умные указатели и move-семантику, но с многопоточностью пока знаком на уровне «запустил std::thread, вроде работает». Цель — дойти до уровня, на котором вы проектируете конкурентные подсистемы, обосновываете решения цифрами и уверенно проходите senior-собеседования.

Правило книги: «так быстрее» подтверждается бенчмарком, «так безопасно» — тестом под ThreadSanitizer и рассуждением через happens-before. Весь код книги собран и проверен под ThreadSanitizer и AddressSanitizer; тесты сквозного проекта проходят десять прогонов подряд.

Как устроена глава: вступление; теория и практика с проверенным кодом; грабли с ревью; вопросы с собеседований; шаг сквозного проекта (что делаем, чем проверяем, что меряем); три задачи разной сложности с критериями самопроверки. Решения всех задач с разбором — в главе «Ответы к задачам» в конце книги.

Сквозной проект shardkv — in-memory key-value сервер, который за 14 шагов проходит путь от потока на соединение с гонкой данных до асинхронного сервера с вычислительным пулом, сбросом нагрузки, CI и журналом архитектурных решений. Два шага из четырнадцати — неудавшиеся эксперименты, и это осознанно: умение отказаться от красивой оптимизации, которая не прошла замер, — тоже навык сеньора.

Что нужно для примеров: GCC 14+ или Clang 18+ с -std=c++23, CMake 3.25+, Ninja; для глав 10–13 — Asio (standalone) и stdexec, проект скачивает их сам через FetchContent. Все цифры замеров сняты в контейнере с двумя виртуальными ядрами, где сервер и нагрузочный клиент делят процессор; на вашей машине абсолютные значения будут другими — прогоните бенчмарки сами, это часть обучения.

Для первой работы достаточно глав 1–7, лёгких и средних задач из них и проекта до шага 7. Остальное — путь к уровню senior.

Глава 1. std::thread, std::jthread и stop_token: как запустить поток и не вызвать std::terminate

«У меня локально работало». Последние слова перед инцидентом.

Поздравляю: вы решили заняться многопоточностью. Это примерно как решить заняться жонглированием, только вместо мячиков бензопилы, а зрители — ваши пользователи в продакшене.

Хорошая новость: запустить поток в C++ можно одной строкой. Плохая: уронить программу тоже можно одной строкой, причём той же самой. Эта глава о том, как запускать потоки так, чтобы они гарантированно завершались, не оставляли висячих ссылок и не превращали выход из программы в лотерею.

К концу главы вы:

— поймёте, что такое поток с точки зрения ОС и почему его не создают на каждый запрос;

— узнаете, почему деструктор std::thread вызывает std::terminate и почему detach() — это не решение, а отсрочка проблемы;

— научитесь передавать в поток аргументы и не терять их по дороге;

— освоите std::jthread и кооперативную отмену через stop_token;

— научитесь доставлять исключения из потока туда, где их могут обработать;

— напишете первый шаг сквозного проекта: TCP-сервер shardkv с потоком на соединение, который умеет корректно останавливаться.

Про примеры. Весь код в книге написан на C++23 и собирается GCC 14+ или Clang 18+ с флагом -std=c++23. Для вывода используется std::println из заголовка <print>. Каждый пример собран и прогнан под ThreadSanitizer и AddressSanitizer — иначе какой смысл.

1.1. Что такое поток и сколько он стоит

Процесс — это изолированная песочница: своё адресное пространство, свои открытые файлы, свои ресурсы. Поток — это поток выполнения внутри процесса: свой стек, свои регистры, свой указатель команд. А вот память, куча, глобальные переменные и файловые дескрипторы у всех потоков процесса общие.

Именно в этом «общие» и кроется вся сила многопоточности и вся её боль. Потоки могут работать с одними данными без копирования. И точно так же они могут одновременно эти данные испортить.

В Linux std::thread — это обёртка над pthread, а pthread — над системным вызовом clone. Когда вы создаёте поток, происходит следующее:

— ядро выделяет структуру задачи и регистрирует её в планировщике;

— резервируется стек (в Linux по умолчанию 8 МБ виртуальной памяти, физическая выделяется по мере использования);

— создаётся локальное хранилище потока (thread_local);

— планировщик решает, на каком ядре и когда поток начнёт работать.

Давайте не верить на слово, а померить:

ex01_cost.cpp C++

```cpp
// ex01_cost.cpp — сколько стоит поток. Спойлер: дороже, чем кажется джуну,
// и дешевле, чем кажется параноику.
#include <chrono>
#include <cstdint>
#include <print>
#include <thread>

int main() {
    using clock = std::chrono::steady_clock;  // для замеров только steady_clock
    constexpr int kThreads = 10'000;

    const auto start = clock::now();
    for (int i = 0; i < kThreads; ++i) {
        std::jthread t([] { /* ничего не делаем — меряем чистые накладные */ });
    }   // деструктор jthread сам делает join — никаких сюрпризов
    const auto elapsed = clock::now() - start;

    const auto ns = std::chrono::duration_cast<std::chrono::nanoseconds>(elapsed).count();
    std::println("Создание + join одного потока: ~{} мкс", ns / kThreads / 1000.0);

```

ex01_cost.cpp (продолжение) C++

```
    // Для сравнения: вызов пустой функции стоит около наносекунды.
    // Поток — это системный вызов, стек и работа планировщика.
    // Вывод: потоки не создают на каждый чих. Их переиспользуют
        (глава 7).
}
```

Листинг 1.1 — сколько стоит поток (ex01_cost.cpp)

На моей машине в release-сборке получилось около 55 мкс на поток. На голом железе с хорошим ядром будет от единиц до десятков микросекунд. Кажется, немного? Вызов пустой функции стоит около наносекунды. Поток дороже в десятки тысяч раз.

Отсюда первое правило: потоки не создают на каждый чих. Если сервер на каждый запрос создаёт поток, при нагрузке он будет тратить время не на работу, а на создание и уничтожение потоков. Правильный ответ — пул потоков, мы напишем его в главе 7. А пока, для учебного сервера, поток на соединение — нормальная отправная точка.

Второе правило: потоков полезно иметь примерно столько, сколько ядер. std::thread::hardware_concurrency() вернёт число аппаратных потоков, но это подсказка, а не гарантия: функция имеет право вернуть 0, если не знает. Если вычислительных потоков больше, чем ядер, они не ускоряют работу, а толкаются за процессорное время, и каждый переключатель контекста стоит микросекунды.

1.2. join, detach и std::terminate

Создать поток просто: передаём в конструктор std::thread что-нибудь вызываемое. Дальше у вас есть ровно два варианта, и выбрать один из них нужно обязательно:

— join() — дождаться завершения потока;

— detach() — отпустить поток, он будет работать сам по себе, и вы больше не можете его дождаться.

А если не выбрать ни один?

ex02_terminate.cpp C++

```cpp
// ex02_terminate.cpp — как уронить программу одной строкой.
#include <print>
#include <thread>

void work() { std::println("Работаю..."); }

int main() {
    std::thread t(work);
    // Забыли t.join() или t.detach().
    // Деструктор std::thread видит joinable() == true и вызывает
        std::terminate().
    // Никаких исключений, никаких деструкторов — сразу abort.
}   // <- вот здесь программа и умрёт
```

Листинг 1.2 — как уронить программу одной строкой (ex02_terminate.cpp)

Программа упадёт через std::terminate. Это не баг стандартной библиотеки, это осознанное решение. Объект std::thread, у которого joinable() == true, представляет живой поток. Если такой объект разрушается, у деструктора есть три варианта: дождаться потока (а вдруг он вечный, и программа зависнет), отпустить поток (а вдруг он работает с локальными переменными, которые сейчас умрут) или убить программу. Комитет выбрал третий: громкая смерть лучше тихого UB.

Почему detach() — почти всегда ошибка

Джун, впервые увидевший terminate, обычно делает так: «а, нужно вызвать detach, и всё заработает». Работать действительно начинает. До первого инцидента.

После detach() у вас нет никакого способа узнать, когда поток закончил работу. А значит, нет способа гарантировать, что всё, к чему поток обращается, ещё живо:

— локальные переменные функции, которая создала поток, умрут при выходе из неё;

— объект, чей метод запущен в потоке, может быть уничтожен владельцем;

— при выходе из main разрушаются глобальные и статические объекты, а отпущенный поток продолжает ими пользоваться (подробности в разделе 1.9).

Это классический use-after-free, только с отложенным действием и проявляется он «иногда». Под нагрузкой. В пятницу.

У detach есть законные применения, но их мало: поток, который не обращается ни к чему, кроме собственных данных, переданных по значению, и время жизни которого вам действительно безразлично. На практике это редкость. Правило для ревью: каждый detach() в коде требует письменного обоснования, почему все используемые данные переживут поток.

1.3. Исключения и join: почему «просто не забыть» не работает

Хорошо, договорились: всегда вызываем join(). Но посмотрите на этот код:

ex03_exception_join.cpp

```cpp
// ex03_exception_join.cpp — почему «просто не забыть join()» не работает.
#include <print>
#include <stdexcept>
#include <thread>

void risky_setup() { throw std::runtime_error("конфиг не найден"); }

// ПЛОХО: исключение между созданием потока и join() приводит к std::terminate.
void bad() {
    std::thread t([] { std::println("фоновая работа"); });
    risky_setup();  // бросает -> раскрутка стека -> ~thread() с joinable -> terminate
    t.join();       // сюда мы не дойдём
}

// ХОРОШО: RAII. jthread делает join в деструкторе, в том числе при раскрутке стека.
void good() {
    std::jthread t([] { std::println("фоновая работа"); });
    risky_setup();  // бросает -> ~jthread() -> request_stop() + join() -> исключение летит дальше
}

```

ex03_exception_join.cpp (продолжение)

```cpp
int main() {
    try {
        good();
    } catch (const std::exception& e) {
        std::println("Поймали честно: {}", e.what());
    }
    // bad();  // раскомментируйте, если хотите увидеть abort своими глазами
}
```

Листинг 1.3 — почему «просто не забыть join()» не работает (ex03_exception_join.cpp)

В функции bad() между созданием потока и join() есть код, который может бросить исключение. Исключение начинает раскрутку стека, деструктор std::thread видит живой поток и вызывает terminate. Исключение, которое можно было бы аккуратно обработать, превращается в падение программы.

Можно, конечно, обложить всё try/catch и звать join() в каждой ветке. Но мы пишем на C++, а в C++ ресурс, который нужно освободить, отдают RAII-объекту. Поток — тоже ресурс.

1.4. std::jthread: поток, который умеет за собой убирать

В C++20 появился std::jthread («joining thread»). Его отличия от std::thread:

— деструктор не вызывает terminate, а сначала просит поток остановиться (request_stop()), затем делает join();

— у него встроенный механизм кооперативной отмены: std::stop_source и std::stop_token;

— если первый параметр функции потока имеет тип std::stop_token, jthread сам передаст туда свой токен.

Функция good() из предыдущего примера использует именно jthread. Исключение летит дальше, а поток корректно дожидается в деструкторе. Никаких terminate.

Правило книги, которое будет действовать до последней главы: **по умолчанию используйте std::jthread.** std::thread оставьте для случаев, когда вам действительно нужно поведение «terminate, если забыли» (почти никогда) или вы работаете со старым стандартом.

Одна тонкость: join в деструкторе означает, что деструктор может ждать. Если поток не проверяет запрос остановки, деструктор jthread будет ждать вечно. Программа не упадёт, а зависнет, и это тоже плохо, поэтому потоки должны уметь останавливаться (раздел 1.6).

1.5. Передача аргументов: копии, ссылки и ловушки

Аргументы конструктора потока копируются (точнее, проходят decay-copy) во внутреннее хранилище потока в том потоке, который его создаёт. Потом новый поток вызывает функцию, передав ей эти копии как rvalue. Из этого следует несколько неочевидных вещей.

ex04_args.cpp C++

```cpp
// ex04_args.cpp — как аргументы попадают в поток и где они теряются по дороге.
#include <memory>
#include <print>
#include <string>
#include <thread>
#include <vector>

void append(std::vector<int>& v, int x) { v.push_back(x); }
void greet(const std::string& name) { std::println("Привет, {}", name); }
void consume(std::unique_ptr<int> p) { std::println("Забрал {}", *p); }

int main() {
    // 1. Аргументы КОПИРУЮТСЯ (decay-copy) во внутреннее хранилище потока.
    //    Поэтому ссылку так просто не передать:
    std::vector<int> data;
    // std::jthread t1(append, data, 42);   // не скомпилируется: rvalue-копию
    //                                      // нельзя привязать к std::vector<int>&
    {
        std::jthread t1(append, std::ref(data), 42);  // явно говорим «да, ссылка»
    }   // join здесь: data гарантированно живее потока
    std::println("data.size() = {}", data.size());
```

ex04_args.cpp (продолжение) C++

```cpp

    // 2. Ловушка с указателем на локальный буфер.
    //    Преобразование char* -> std::string произойдёт УЖЕ В НОВОМ ПОТОКЕ,
    //    когда буфер может быть мёртв (если бы поток был detach-нут).
    char buffer[] = "Гонка";
    {
        std::jthread t2(greet, std::string(buffer));  // конвертируем ДО создания потока
    }

    // 3. Move-only типы просто перемещаем.
    auto p = std::make_unique<int>(7);
    {
        std::jthread t3(consume, std::move(p));
    }

    // 4. Лямбда — самый читаемый вариант. Но захват по ссылке = обещание,
    //    что объект переживёт поток. Обещания надо выполнять.
    int counter = 0;
    {
        std::jthread t4([&counter] { ++counter; });
    }   // join: обещание выполнено
    std::println("counter = {}", counter);
}
```

Листинг 1.4 — как аргументы попадают в поток и где они теряются по дороге (ex04_args.cpp)

Ссылки не передаются неявно. Если функция принимает std::vector<int>&, передать туда вектор просто так нельзя: копия — это rvalue, и к неконстантной ссылке она не привязывается. Код не скомпилируется, и это хорошо. Если вы действительно хотите ссылку, оборачиваете в std::ref и тем самым берёте на себя обязательство, что объект переживёт поток.

Неявные преобразования случаются в новом потоке. Если функция принимает const std::string&, а вы передаёте char*, сам указатель скопируется сразу, а превращение в std::string произойдёт позже, уже внутри нового потока. Если к этому моменту буфер умер (например, функция-создатель вернулась, а поток был отпущен через detach), вы получите строку из мусора. Лечение: конвертируйте явно, до передачи в поток.

Move-only типы перемещаются. std::unique_ptr передаётся через std::move — и это лучший способ отдать потоку владение данными: если поток владеет данными, никто не может уничтожить их раньше времени.

Лямбды с захватом по ссылке — это обещание. Захват [&] читается приятнее всего, но означает ровно то же, что std::ref: «я гарантирую, что всё это переживёт поток». С jthread в той же области видимости обещание выполняется автоматически: деструктор jthread сработает

раньше, чем захваченные переменные, объявленные до него. Порядок объявления здесь не случаен — к этому мы вернёмся в задаче 1.3.

1.6. Кооперативная отмена: stop_token, stop_source, stop_callback

Первое желание новичка — «убить поток снаружи». В C++ такой возможности нет, и это правильно. Поток, убитый посреди работы, может держать захваченный мьютекс, недописанный файл, наполовину изменённую структуру данных. После такой операции вся программа находится в неизвестном состоянии.

Поэтому в C++ поток **просят** остановиться, а он сам решает, в какой точке это безопасно. Механизм состоит из трёх частей:

— **std::stop_source** — «кнопка». Метод request_stop() нажимает её. Нажать можно один раз, это необратимо.

— **std::stop_token** — «лампочка», которую выдаёт stop_source через get_token(). Поток спрашивает stop_requested() и узнаёт, нажата ли кнопка. Токены дешёво копируются, и все они смотрят на одну кнопку.

— **std::stop_callback** — «реакция на нажатие». Регистрирует функцию, которая будет вызвана при запросе остановки.

У каждого jthread есть свой stop_source. Его токен приходит в функцию потока первым параметром, если она его принимает:

ex05_stop.cpp C++

```cpp
// ex05_stop.cpp — кооперативная отмена: просим поток остановиться, а не убиваем его.
#include <atomic>
#include <chrono>
#include <print>
#include <stop_token>
#include <thread>

using namespace std::chrono_literals;

int main() {
    std::atomic<int> ticks{0};  // атомик, потому что читаем из main (подробности в главе 5)

    // Если первый параметр callable — std::stop_token, jthread передаст его сам.
    std::jthread worker([&ticks](std::stop_token st) {
        // stop_callback вызывается, когда кто-то попросит остановку.
        // Тут можно разбудить блокирующую операцию: закрыть сокет, отменить I/O.
        std::stop_callback on_stop(st, [] { std::println("worker: меня попросили остановиться"); });

        while (!st.stop_requested()) {
            ticks.fetch_add(1, std::memory_order_relaxed);
            std::this_thread::sleep_for(10ms);  // «полезная работа»
        }
        std::println("worker: аккуратно закрыл дела и выхожу");
```

ex05_stop.cpp (продолжение) C++

```cpp
    });

    std::this_thread::sleep_for(100ms);
    worker.request_stop();  // можно не звать: деструктор jthread
        сделает это сам
    worker.join();
    std::println("main: поток сделал ~{} итераций", ticks.load());

    // Один stop_source на несколько потоков — общая кнопка «стоп».
    std::stop_source stop_all;
    std::atomic<int> finished{0};
    {
        std::jthread a([tok = stop_all.get_token(), &finished] {
            while (!tok.stop_requested())
                std::this_thread::sleep_for(1ms);
            finished.fetch_add(1);
        });
        std::jthread b([tok = stop_all.get_token(), &finished] {
            while (!tok.stop_requested())
                std::this_thread::sleep_for(1ms);
            finished.fetch_add(1);
        });
        std::this_thread::sleep_for(20ms);
        stop_all.request_stop();  // остановили оба одной командой
    }   // join обоих
    std::println("main: остановлено потоков: {}", finished.load());
}
```

Листинг 1.5 — кооперативная отмена (ex05_stop.cpp)

Обратите внимание на три детали.

Проверка флага работает только для потоков, которые не спят. Цикл while (!st.stop_requested()) хорош, пока поток крутится и периодически возвращается к проверке. Но если поток заблокирован в системном вызове (ждёт данные из сокета в recv() или клиента в accept()), он флаг не проверит, пока вызов не вернётся. А вернуться он может через час. Вот тут и нужен stop_callback: в нём можно разбудить блокирующую операцию — закрыть сокет на чтение, отменить ввод-вывод, разбудить condition variable (глава 3). Именно так работает остановка в нашем сервере.

Где выполняется stop_callback. Колбэк вызывается в том потоке, который вызвал request_stop(), а не в потоке, который ждёт. Если на момент регистрации остановка уже запрошена, колбэк выполнится сразу, в конструкторе stop_callback. Деструктор stop_callback подождёт, если колбэк прямо сейчас выполняется в другом потоке. Эта гарантия нам ещё пригодится.

Один stop_source может управлять многими потоками. Во второй части примера два потока получают токены от одного источника и останавливаются одной командой. Это стандартный приём для сервисов: одна кнопка «стоп» на все рабочие потоки.

1.7. Исключения в потоке

Если исключение вылетает из функции потока наружу, вызывается std::terminate. Поймать его в main нельзя: у каждого потока свой стек, и раскрутка стека одного потока не может «перепрыгнуть» в другой.

Значит, исключение нужно перевезти. Для этого есть std::exception_ptr — указатель, который может хранить исключение любого типа:

ex06_exceptions.cpp

```cpp
// ex06_exceptions.cpp — исключение, вылетевшее из функции потока, убивает ВСЮ программу.
// Поймать его в main нельзя: у каждого потока свой стек. Значит, его надо перевезти.
#include <exception>
#include <print>
#include <stdexcept>
#include <thread>

int main() {
    std::exception_ptr error;  // «контейнер» для исключения любого типа

    {
        std::jthread t([&error] {
            try {
                throw std::runtime_error("диск кончился");
            } catch (...) {
                error = std::current_exception();  // упаковали
            }
        });
    }   // join: после него запись в error гарантированно видна в main

```

ex06_exceptions.cpp (продолжение) C++

```
    if (error) {
        try {
            std::rethrow_exception(error);  // распаковали и бросили
                уже здесь
        } catch (const std::exception& e) {
            std::println("main поймал исключение из потока: {}",
                e.what());
        }
    }
    // В главе 7 то же самое сделает за нас std::future. А пока —
        руками.
}
```

Листинг 1.6 — исключение, вылетевшее из функции потока, убивает всю программу (ex06_exceptions.cpp)

Внутри потока ловим всё через catch (…), упаковываем через std::current_exception(), а в вызывающем потоке после join() распаковываем через std::rethrow_exception(). Почему после join()? Потому что join() гарантирует, что всё, что сделал поток, видно потоку, который его дождался. Без этой гарантии чтение error в main было бы гонкой данных. Формально это отношение называется happens-before, и в главе 5 мы разберём его досконально.

В главе 7 то же самое за нас будет делать std::future, но полезно один раз сделать это руками, чтобы понимать, что происходит внутри.

1.8. thread_local и hardware_concurrency

Переменная с thread_local существует в отдельной копии для каждого потока. Каждый поток видит и меняет только свою копию, поэтому гонок на ней не бывает:

ex07_thread_local.cpp

```cpp
// ex07_thread_local.cpp — у каждого потока своя копия переменной.
#include <print>
#include <string>
#include <thread>
#include <vector>

thread_local int calls = 0;  // своя для каждого потока, создаётся при первом обращении

void handle() { ++calls; }

int main() {
    std::vector<std::jthread> threads;
    for (int id = 0; id < 3; ++id) {
        threads.emplace_back([id] {
            for (int i = 0; i <= id; ++i) handle();
            std::println("поток {}: calls = {}", id, calls);  // 1, 2, 3 — без гонок
        });
    }
    threads.clear();  // деструкторы jthread: join всех
    std::println("main: calls = {}", calls);  // 0: у main своя копия
}
```

Листинг 1.7 — у каждого потока своя копия переменной (ex07_thread_local.cpp)

Где это полезно: счётчики и статистика по потокам (сливаем в общий итог редко), кеши и буферы, которые дорого создавать на каждый вызов, генераторы случайных чисел (std::mt19937 не потокобезопасен, а заводить мьютекс ради rand — грустно).

Где это опасно: копия создаётся для каждого потока, и с пулом из 64 потоков «маленький кеш на 10 МБ» внезапно становится 640 МБ. И деструкторы thread_local-объектов выполняются при завершении потока, то есть у отпущенного через detach потока — неизвестно когда.

1.9. Завершение программы при живых потоках

Вот сценарий, который ломает программы, считающиеся рабочими годами. Поток отпущен через detach() и пишет в лог через глобальный объект-логгер. main() возвращает управление. Что происходит дальше:

— возврат из main эквивалентен вызову std::exit();

— std::exit() запускает деструкторы статических и глобальных объектов, в том числе вашего логгера;

— отпущенный поток в это время продолжает работать и вызывает методы уже разрушенного логгера;

— в лучшем случае программа падает при выходе, в худшем — портит данные, которые успела записать на диск.

Самое неприятное: это проявляется не всегда, а только когда поток не успел закончить до выхода. На машине разработчика он успевает. В продакшене под нагрузкой — нет.

Что можно сделать:

— **Правильно:** все потоки завершаются до выхода из main. Владелец потоков (обычно какой-то сервис) останавливает их через stop_token и дожидается в деструкторе. С jthread это получается само собой.

— **Аварийно:** std::quick_exit() завершает программу без деструкторов статических объектов (выполняются только функции, зарегистрированные через std::at_quick_exit). Это способ выйти, когда корректная остановка невозможна, а не способ лечить архитектуру.

— **Никогда:** надеяться, что «поток успеет».

1.10. Грабли с ревью

Список того, что я регулярно встречаю в pull request'ах. Каждый пункт — повод для комментария «Request changes».

— **detach() без обоснования.** Почти всегда означает «я не знаю, как остановить этот поток».

— **std::thread в коде на C++20.** Используйте std::jthread, если нет конкретной причины.

— **Захват [&] в лямбде потока, который живёт дольше текущей области видимости.** Особенно в циклах: счётчик цикла, захваченный по ссылке, — классическая гонка.

— **Поток, который нельзя остановить.** Бесконечный цикл без проверки stop_token или блокирующий вызов без способа его разбудить.

— **Исключения, вылетающие из функции потока.** Это std::terminate, а не «обработается где-нибудь».

— **sleep_for как способ синхронизации.** «Подождём 100 мс, поток наверняка успеет» — не синхронизация, а надежда.

— **Создание потока на каждый запрос.** Работает на тестах, умирает под нагрузкой.

— **Чтение результатов потока без join() или другой синхронизации.** Это data race, даже если «там просто int».

1.11. Что спросят на собеседовании

Что произойдёт, если не вызвать join() или detach() у std::thread? (middle) Деструктор вызовет std::terminate. Объяснение «почему» ценится больше, чем сам факт: у деструктора нет безопасного способа решить, ждать или отпустить поток.

Чем std::jthread отличается от std::thread? (middle) Деструктор делает request_stop() и join() вместо terminate; встроенная кооперативная отмена через stop_source/stop_token; автоматическая передача stop_token в функцию потока.

Как остановить поток? (middle) Только кооперативно: поток периодически проверяет stop_token или его будят через stop_callback. Убить поток снаружи в стандартном C++ нельзя, и это правильно. Сильный ответ отдельно разбирает поток, заблокированный в системном вызове.

Почему detach() опасен? (middle) Нельзя узнать, когда поток закончится, а значит, нельзя гарантировать время жизни данных, которые он использует. Плюс проблема разрушения статических объектов при выходе из программы.

Как передать исключение из потока в вызывающий код? (middle) std::exception_ptr через std::current_exception()/std::rethrow_exception(), либо std::promise/std::future, которые делают то же самое.

Как передать в поток аргумент по ссылке и чем это опасно? (junior/middle) std::ref или захват по ссылке в лямбде. Опасно тем, что объект должен пережить поток. Бонус: рассказать про преобразование char* в std::string, которое происходит уже в новом потоке.

Сколько потоков стоит создавать? (senior) Для вычислительной работы — порядка hardware_concurrency(). Для работы с ожиданием I/O больше, но лучше асинхронный I/O (глава 10). Сеньор говорит «зависит от профиля нагрузки» и сразу объясняет, как это измерить.

Что произойдёт, если отпущенный поток работает в момент выхода из main? (senior) exit() разрушит статические объекты, пока поток ими пользуется: UB. Правильное решение — останавливать и дожидаться всех потоков до выхода.

1.12. Сквозной проект shardkv, шаг 1: поток на соединение

Пора писать сервер. Напомню: shardkv — in-memory key-value хранилище с текстовым протоколом в духе Redis. В течение книги оно пройдёт путь от наивной версии до архитектуры, которую не стыдно обсуждать на senior-собеседовании. На каждом шаге мы делаем, проверяем и меряем.

Что делаем на шаге 1:

— скелет проекта: CMake с пресетами debug, release, tsan, asan;

— однопоточное хранилище на std::unordered_map;

— протокол: PING, GET, SET, DEL, SETNX, INCR;

— TCP-сервер на блокирующих сокетах, по потоку std::jthread на каждое соединение;

— корректную остановку: сервер выходит, даже если клиенты подключены и молчат.

Что специально оставляем сломанным: хранилище не защищено от одновременного доступа. Несколько соединений одновременно меняют один unordered_map. Это data race, и в главе 2 мы начнём именно с того, что ThreadSanitizer его найдёт.

Что меряем: ничего. Мерить сломанный код рано.

Как устроена остановка

Это главная многопоточная часть шага. Поток сервера почти всё время висит в accept(), а потоки соединений — в recv(). Проверка stop_requested() в цикле их не разбудит. Поэтому:

— сервер регистрирует stop_callback, который вызывает shutdown() на слушающем сокете. В Linux это заставляет accept() вернуть ошибку, и цикл приёма заканчивается;

— у сервера есть собственный std::stop_source для соединений. Он срабатывает и когда остановку попросили снаружи, и когда accept() сломался сам по себе. Без этого при фатальной ошибке accept сервер вышел бы из цикла и навсегда завис бы в ожидании соединений, которые никто не разбудил;

— каждое соединение регистрирует свой stop_callback, который делает shutdown() своего сокета. recv() возвращает 0, цикл чтения заканчивается;

— объекты Connection хранят потоки соединений, и при выходе из run() все они дожидаются через деструкторы jthread.

Отдельно посмотрите на порядок объявления в serve(): stop_callback объявлен после сокета, а значит, разрушается раньше. Деструктор stop_callback подождёт, если колбэк выполняется в другом потоке. Поэтому shutdown() никогда не попадёт в уже закрытый дескриптор.

Ещё одна деталь: сервер убирает отработавшие соединения из списка (reap_finished). Флаг done читается из потока сервера и пишется из потока соединения, поэтому он std::atomic с порядками release/acquire. Что это значит, мы подробно разберём в главе 5, а пока примите как данность: без атомика это была бы ещё одна гонка.

Ключевой код

Заголовок сервера:

server.hpp C++

```cpp
// server.hpp — TCP-сервер shardkv, шаг 1: поток на соединение.
#pragma once

#include <atomic>
#include <cstdint>
#include <list>
#include <stop_token>
#include <thread>

#include "core/storage.hpp"
#include "net/socket.hpp"

namespace shardkv {

class Server {
public:
    // Сразу открывает слушающий сокет: ошибка bind видна в конструкторе, а не «где-то потом».
    Server(Storage& storage, std::uint16_t port);

    // Порт, на котором реально слушаем (если просили 0).
    [[nodiscard]] std::uint16_t port() const noexcept { return port_; }

```

server.hpp (продолжение) C++

```cpp
    // Цикл приёма соединений. Возвращается, когда запрошена остановка через token.
    // Перед выходом дожидается завершения всех потоков соединений.
    void run(std::stop_token token);

private:
    struct Connection {
        std::atomic<bool> done{false};  // поток соединения закончил работу (читает acceptor)
        std::jthread thread;            // порядок членов не случаен — подумайте почему (задача 1.3)
    };

    void serve(net::Fd client, std::stop_token token,
        std::atomic<bool>& done);
    void reap_finished();

    Storage& storage_;
    net::Fd listener_;
    std::uint16_t port_;
    std::list<Connection> connections_;  // list: адреса элементов стабильны при вставке и удалении
};

}  // namespace shardkv
```

Листинг 1.8 — TCP-сервер shardkv, шаг 1 (server.hpp)

Реализация:

server.cpp C++

```cpp
#include "server/server.hpp"

#include <sys/socket.h>

#include <cerrno>
#include <print>

#include "core/protocol.hpp"

namespace shardkv {

Server::Server(Storage& storage, std::uint16_t port)
    : storage_(storage), listener_(net::listen_tcp(port)),
      port_(net::local_port(listener_)) {}

void Server::run(std::stop_token token) {
    // Свой источник остановки для соединений. Он срабатывает в двух случаях:
    // остановку попросили снаружи (через token) или accept сломался сам.
    std::stop_source connections_stop;

    // accept() блокирует поток. Проверка token.stop_requested() в цикле его не разбудит.
    // Поэтому при остановке «выбиваем» accept: shutdown на слушающем сокете
    // заставляет его вернуть ошибку (поведение Linux).
    std::stop_callback on_stop(token, [this, &connections_stop] {
```

server.cpp (продолжение) C++

```cpp
        connections_stop.request_stop();
        net::shutdown_both(listener_);
    });

    while (!token.stop_requested()) {
        net::Fd client(::accept4(listener_.get(), nullptr, nullptr,
            SOCK_CLOEXEC));
        if (!client.valid()) {
            if (errno == EINTR || errno == ECONNABORTED) continue;
            if (!token.stop_requested()) std::println(stderr,
                "accept: фатальная ошибка, errno={}", errno);
            break;
        }

        reap_finished();  // заодно убираем отработавшие соединения,
            чтобы список не рос вечно

        auto& conn = connections_.emplace_back();
        conn.thread = std::jthread(
            [this, fd = std::move(client), stop =
                connections_stop.get_token(), &done = conn.done]()
                mutable {
                serve(std::move(fd), stop, done);
            });
    }

```

server.cpp (продолжение) C++

```
    connections_stop.request_stop();  // будим все соединения, даже если вышли из-за ошибки
    connections_.clear();              // ~jthread каждого соединения: join
}

void Server::serve(net::Fd client, std::stop_token token, std::atomic<bool>& done) {
    {
        // Порядок объявления важен: колбэк разрушается раньше client.
        // ~stop_callback ждёт, если колбэк прямо сейчас выполняется в другом потоке,
        // поэтому shutdown никогда не попадёт в уже закрытый дескриптор.
        std::stop_callback wake_recv(token, [&client] { net::shutdown_both(client); });

        net::LineReader reader(client);
        while (auto line = reader.next()) {
            // !!! DATA RACE: storage_ общий для всех потоков соединений и ничем не защищён.
            // Шаг 1 оставляет это намеренно. Исправление — глава 2.
            const std::string reply = execute(storage_, *line) + "\n";
            if (!net::send_all(client, reply)) break;
        }
    }
```

server.cpp (продолжение) C++

```
    done.store(true, std::memory_order_release);  // подробности про memory_order — глава 5
}

void Server::reap_finished() {
    // Удаляем только те соединения, чьи потоки уже вышли из serve():
    // join в деструкторе jthread для них мгновенный.
    connections_.remove_if([](const Connection& c) { return c.done.load(std::memory_order_acquire); });
}

}  // namespace shardkv
```

Листинг 1.9 — реализация (server.cpp)

Точка входа. Обработку SIGTERM мы добавим в главе 4, пока сервер останавливается по Enter:

main.cpp C++

```cpp
// main.cpp — точка входа shardkv, шаг 1.
// Остановка — по Enter в консоли. Нормальную обработку SIGTERM
    добавим в главе 4.
#include <charconv>
#include <cstdint>
#include <cstdio>
#include <exception>
#include <iostream>
#include <print>
#include <string_view>
#include <thread>

#include "core/storage.hpp"
#include "server/server.hpp"

int main(int argc, char** argv) {
    std::uint16_t port = 6380;
    if (argc > 1) {
        const std::string_view arg = argv[1];
        if (std::from_chars(arg.data(), arg.data() + arg.size(),
            port).ec != std::errc{}) {
            std::println(stderr, "Использование: {} [порт]",
                argv[0]);
            return 2;
        }
    }

```

main.cpp (продолжение) C++

```cpp
    try {
        shardkv::Storage storage;
        shardkv::Server server(storage, port);
        std::println("shardkv слушает 127.0.0.1:{}. Enter —
            остановить.", server.port());

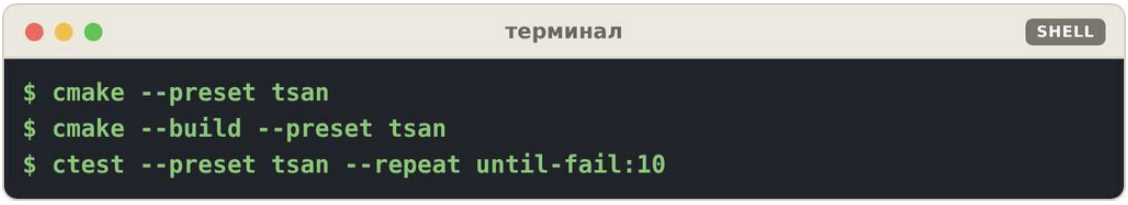
        std::jthread acceptor([&server](std::stop_token st) {
            server.run(st); });

        std::cin.get();
        acceptor.request_stop();  // будит accept и все соединения
            через stop_callback
        acceptor.join();          // явно: хотим напечатать
            сообщение уже после остановки
        std::println("Остановлено корректно. Ключей в памяти: {}",
            storage.size());
    } catch (const std::exception& e) {
        std::println(stderr, "Ошибка: {}", e.what());
        return 1;
    }
}
```

Листинг 1.10 — точка входа shardkv, шаг 1 (main.cpp)

Сборка, запуск, проверка

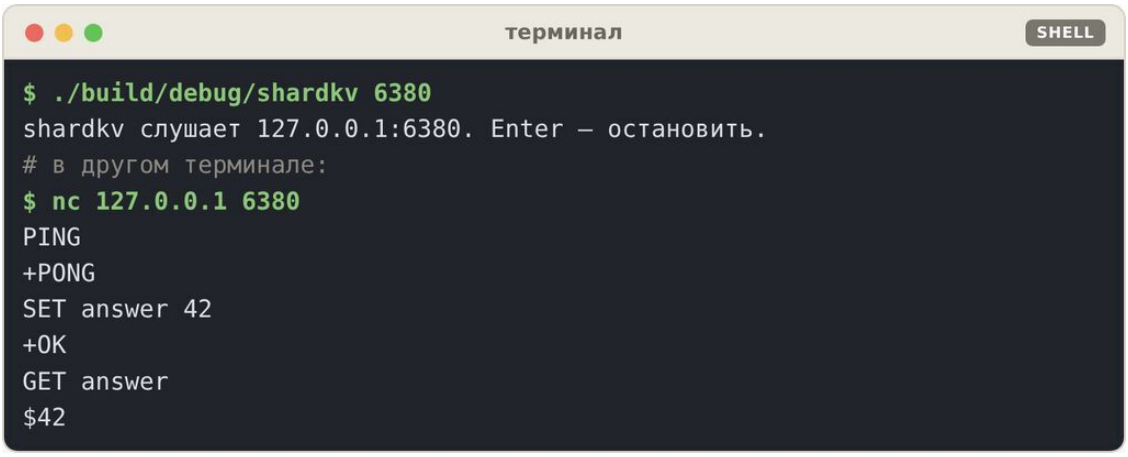
Сборка и тесты под ThreadSanitizer, десять прогонов подряд:



Листинг 1.11 — сборка и тесты под TSan, десять прогонов подряд

Тесты проверяют протокол и две вещи про потоки: сервер отвечает и корректно останавливается, а остановка будит соединения, которые подключились и молчат. Если остановка сломана, тест не упадёт, а повиснет, поэтому у тестов стоит таймаут.

Запуск руками:



Листинг 1.12 — ручная проверка сервера через nc

Тизер главы 2

Теперь запустим сервер под TSan и подключим четыре клиента, которые одновременно делают INCR одного ключа:

терминал SHELL

```
$ ./build/tsan/shardkv 6380
# в другом терминале:
$ for i in 1 2 3 4; do
$   (for j in $(seq 200); do echo "INCR hits"; done) | nc -q1 127.0.0.1
    6380 > /dev/null &
$ done
```

Листинг 1.13 — четыре клиента одновременно делают INCR

В логе сервера появится:

терминал SHELL

```
$ ./build/tsan/shardkv 6380
shardkv слушает 127.0.0.1:6380. Enter — остановить.
WARNING: ThreadSanitizer: data race (pid=9101)
# … отчёт сокращён до сути
```

Листинг 1.14 — отчёт TSan в логе сервера

Это не ложное срабатывание. Четыре потока одновременно меняют один unordered_map. В release-сборке без санитайзера такой код может терять инкременты, возвращать мусор или упасть посреди перехеширования таблицы. Как это чинить и почему «оно же почти всегда работает» — не аргумент, разберём в следующей главе.

Полный код шага 1

Для тех, кто собирает проект с нуля, — остальные файлы. В них нет ничего многопоточного: хранилище, протокол и RAII-обёртка над сокетами.

storage.hpp C++

```cpp
// storage.hpp — хранилище shardkv, шаг 1.
//
// ВНИМАНИЕ: этот класс НАМЕРЕННО не потокобезопасен.
// В шаге 1 к нему обращаются потоки всех соединений одновременно —
    это data race,
// то есть неопределённое поведение. В главе 2 TSan поймает нас за
    руку, и мы это исправим.
#pragma once

#include <cstdint>
#include <optional>
#include <string>
#include <string_view>
#include <unordered_map>

namespace shardkv {

class Storage {
public:
    [[nodiscard]] std::optional<std::string> get(std::string_view
        key) const {
        if (auto it = map_.find(std::string(key)); it != map_.end())
            return it->second;
        return std::nullopt;
    }

```

storage.hpp (продолжение) C++

```cpp
    void set(std::string_view key, std::string_view value) {
        map_.insert_or_assign(std::string(key), std::string(value));
    }

    // true, если ключ был и удалён
    bool del(std::string_view key) { return
        map_.erase(std::string(key)) > 0; }

    // SET if Not eXists: true, если записали
    bool setnx(std::string_view key, std::string_view value) {
        return map_.try_emplace(std::string(key), value).second;
    }

    // Увеличивает число по ключу на 1. Отсутствующий ключ считается
        нулём.
    // Возвращает nullopt, если значение не число.
    std::optional<std::int64_t> incr(std::string_view key);

    [[nodiscard]] std::size_t size() const noexcept { return
        map_.size(); }

private:
    std::unordered_map<std::string, std::string> map_;
};

}  // namespace shardkv
```

Листинг 1.15 — хранилище shardkv, шаг 1 (storage.hpp)

storage.cpp C++

```cpp
#include "core/storage.hpp"

#include <charconv>
#include <limits>

namespace shardkv {

std::optional<std::int64_t> Storage::incr(std::string_view key) {
    auto [it, inserted] = map_.try_emplace(std::string(key), "0");
    const std::string& text = it->second;

    std::int64_t value = 0;
    const auto [end, ec] = std::from_chars(text.data(), text.data()
        + text.size(), value);
    if (ec != std::errc{} || end != text.data() + text.size())
        return std::nullopt;
    if (value == std::numeric_limits<std::int64_t>::max()) return
        std::nullopt;  // переполнение — тоже UB

    ++value;
    it->second = std::to_string(value);
    return value;
}

}  // namespace shardkv
```

Листинг 1.16 — реализация (storage.cpp)

protocol.hpp C++

```cpp
// protocol.hpp — текстовый протокол shardkv.
//
// Запрос — одна строка: КОМАНДА [аргументы...], разделитель — пробел.
// Ответ — одна строка:
//   +OK          успех без значения
//   $значение    строка
//   $nil         ключа нет
//   :число       целое
//   -ERR текст   ошибка
#pragma once

#include <string>
#include <string_view>

#include "core/storage.hpp"

namespace shardkv {

// Разбирает одну строку запроса, выполняет её над хранилищем и возвращает ответ без '\n'.
[[nodiscard]] std::string execute(Storage& storage, std::string_view line);

}  // namespace shardkv
```

Листинг 1.17 — текстовый протокол shardkv (protocol.hpp)

protocol.cpp C++

```cpp
#include "core/protocol.hpp"

#include <algorithm>
#include <cctype>
#include <ranges>
#include <string>
#include <vector>

namespace shardkv {
namespace {

std::vector<std::string_view> split(std::string_view line) {
    std::vector<std::string_view> parts;
    for (auto part : line | std::views::split(' ')) {
        if (!part.empty()) parts.emplace_back(part.begin(),
            part.end());
    }
    return parts;
}

std::string upper(std::string_view s) {
    std::string out(s);
    std::ranges::transform(out, out.begin(), [](unsigned char c) {
        return static_cast<char>(std::toupper(c)); });
    return out;
}

```

protocol.cpp (продолжение) C++

```cpp
std::string wrong_args(std::string_view cmd) { return "-ERR wrong number of arguments for " + std::string(cmd); }

}  // namespace

std::string execute(Storage& storage, std::string_view line) {
    if (!line.empty() && line.back() == '\r') line.remove_suffix(1);
        // telnet/nc шлют \r\n

    const auto args = split(line);
    if (args.empty()) return "-ERR empty command";

    const std::string cmd = upper(args[0]);

    if (cmd == "PING") return "+PONG";

    if (cmd == "GET") {
        if (args.size() != 2) return wrong_args(cmd);
        const auto value = storage.get(args[1]);
        return value ? "$" + *value : "$nil";
    }
    if (cmd == "SET") {
        if (args.size() != 3) return wrong_args(cmd);
        storage.set(args[1], args[2]);
        return "+OK";
    }
    if (cmd == "DEL") {
        if (args.size() != 2) return wrong_args(cmd);
```

protocol.cpp (продолжение) C++

```
52          return storage.del(args[1]) ? ":1" : ":0";
53      }
54      if (cmd == "SETNX") {
55          if (args.size() != 3) return wrong_args(cmd);
56          return storage.setnx(args[1], args[2]) ? ":1" : ":0";
57      }
58      if (cmd == "INCR") {
59          if (args.size() != 2) return wrong_args(cmd);
60          const auto value = storage.incr(args[1]);
61          return value ? ":" + std::to_string(*value) : "-ERR value is
 ↪              not an integer or out of range";
62      }
63      return "-ERR unknown command " + std::string(args[0]);
64  }
65
66  }  // namespace shardkv
```

Листинг 1.18 — реализация (protocol.cpp)

socket.hpp C++

```cpp
// socket.hpp — минимальная RAII-обёртка над POSIX-сокетами.
// Ничего многопоточного здесь нет: это просто «чтобы дескрипторы не текли».
#pragma once

#include <cstdint>
#include <optional>
#include <string>
#include <string_view>
#include <utility>

namespace shardkv::net {

// Владеет файловым дескриптором. Move-only, закрывает в деструкторе.
class Fd {
public:
    Fd() noexcept = default;
    explicit Fd(int fd) noexcept : fd_(fd) {}
    Fd(Fd&& other) noexcept : fd_(std::exchange(other.fd_, -1)) {}
    Fd& operator=(Fd&& other) noexcept {
        if (this != &other) {
            reset();
            fd_ = std::exchange(other.fd_, -1);
        }
        return *this;
    }
    Fd(const Fd&) = delete;
    Fd& operator=(const Fd&) = delete;
```

socket.hpp (продолжение) C++

```cpp
    ~Fd() { reset(); }

    [[nodiscard]] int get() const noexcept { return fd_; }
    [[nodiscard]] bool valid() const noexcept { return fd_ >= 0; }
    void reset() noexcept;

private:
    int fd_ = -1;
};

// Слушающий TCP-сокет на 127.0.0.1:port. port == 0 — ОС выберет
    свободный.
// Бросает std::system_error при ошибке.
[[nodiscard]] Fd listen_tcp(std::uint16_t port);

// Порт, на котором реально слушает сокет (полезно при port == 0).
[[nodiscard]] std::uint16_t local_port(const Fd& socket);

// Подключение к 127.0.0.1:port (для тестов и клиента).
[[nodiscard]] Fd connect_tcp(std::uint16_t port);

// Отправляет всё. false — соединение закрыто или ошибка.
bool send_all(const Fd& socket, std::string_view data);

```

socket.hpp (продолжение) C++

```cpp
// Будит поток, заблокированный в accept()/recv() на этом сокете.
// Безопасно вызывать из другого потока: дескриптор при этом не закрывается.
void shutdown_both(const Fd& socket) noexcept;

// Читает сокет построчно (разделитель '\n').
class LineReader {
public:
    explicit LineReader(const Fd& socket) noexcept : socket_(socket) {}
    // Следующая строка без '\n' или nullopt, если соединение закрыто.
    std::optional<std::string> next();

private:
    const Fd& socket_;
    std::string buffer_;
};

}  // namespace shardkv::net
```

Листинг 1.19 — минимальная RAII-обёртка над POSIX-сокетами (socket.hpp)

socket.cpp C++

```cpp
#include "net/socket.hpp"

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

#include <cerrno>
#include <system_error>

namespace shardkv::net {
namespace {

[[noreturn]] void throw_errno(const char* what) { throw
    std::system_error(errno, std::generic_category(), what); }

sockaddr_in loopback(std::uint16_t port) {
    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    return addr;
}

}  // namespace

```

socket.cpp (продолжение) C++

```cpp
void Fd::reset() noexcept {
    if (fd_ >= 0) {
        ::close(fd_);
        fd_ = -1;
    }
}

Fd listen_tcp(std::uint16_t port) {
    Fd fd(::socket(AF_INET, SOCK_STREAM | SOCK_CLOEXEC, 0));
    if (!fd.valid()) throw_errno("socket");

    const int on = 1;
    ::setsockopt(fd.get(), SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);

    auto addr = loopback(port);
    if (::bind(fd.get(), reinterpret_cast<sockaddr*>(&addr), sizeof
        addr) < 0) throw_errno("bind");
    if (::listen(fd.get(), SOMAXCONN) < 0) throw_errno("listen");
    return fd;
}

std::uint16_t local_port(const Fd& socket) {
    sockaddr_in addr{};
    socklen_t len = sizeof addr;
    if (::getsockname(socket.get(), reinterpret_cast<sockaddr*>(&addr
        ), &len) < 0) throw_errno("getsockname");
    return ntohs(addr.sin_port);
}
```

socket.cpp (продолжение) C++

```cpp

Fd connect_tcp(std::uint16_t port) {
    Fd fd(::socket(AF_INET, SOCK_STREAM | SOCK_CLOEXEC, 0));
    if (!fd.valid()) throw_errno("socket");
    auto addr = loopback(port);
    if (::connect(fd.get(), reinterpret_cast<sockaddr*>(&addr),
        sizeof addr) < 0) throw_errno("connect");
    return fd;
}

bool send_all(const Fd& socket, std::string_view data) {
    while (!data.empty()) {
        const auto n = ::send(socket.get(), data.data(),
            data.size(), MSG_NOSIGNAL);
        if (n < 0 && errno == EINTR) continue;
        if (n <= 0) return false;
        data.remove_prefix(static_cast<std::size_t>(n));
    }
    return true;
}

void shutdown_both(const Fd& socket) noexcept {
    ::shutdown(socket.get(), SHUT_RDWR); }

```

socket.cpp (продолжение) C++

```cpp
std::optional<std::string> LineReader::next() {
    for (;;) {
        if (const auto pos = buffer_.find('\n'); pos !=
            std::string::npos) {
            std::string line = buffer_.substr(0, pos);
            buffer_.erase(0, pos + 1);
            return line;
        }
        char chunk[4096];
        const auto n = ::recv(socket_.get(), chunk, sizeof chunk, 0);
        if (n < 0 && errno == EINTR) continue;
        if (n <= 0) return std::nullopt;  // 0 — клиент закрыл
            соединение или нас разбудил shutdown
        buffer_.append(chunk, static_cast<std::size_t>(n));
    }
}

}  // namespace shardkv::net
```

Листинг 1.20 — реализация (socket.cpp)

CMakeLists.txt CMAKE

```
cmake_minimum_required(VERSION 3.25)
project(shardkv LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Санитайзер выбирается пресетом: -DSHARDKV_SANITIZER=thread |
    address
set(SHARDKV_SANITIZER "" CACHE STRING "thread, address или пусто")

# Строгие предупреждения только для нашего кода, не для googletest
set(SHARDKV_WARNINGS -Wall -Wextra -Wpedantic -Werror)
if(SHARDKV_SANITIZER STREQUAL "thread")
    add_compile_options(-fsanitize=thread -g -O1)
    add_link_options(-fsanitize=thread)
elseif(SHARDKV_SANITIZER STREQUAL "address")
    add_compile_options(-fsanitize=address,undefined
        -fno-omit-frame-pointer -g -O1)
    add_link_options(-fsanitize=address,undefined)
endif()

find_package(Threads REQUIRED)

```

CMakeLists.txt (продолжение) CMAKE

```cmake
add_library(shardkv_lib
    src/core/storage.cpp
    src/core/protocol.cpp
    src/net/socket.cpp
    src/server/server.cpp)
target_include_directories(shardkv_lib PUBLIC src)
target_link_libraries(shardkv_lib PUBLIC Threads::Threads)
target_compile_options(shardkv_lib PRIVATE ${SHARDKV_WARNINGS})

add_executable(shardkv src/server/main.cpp)
target_link_libraries(shardkv PRIVATE shardkv_lib)
target_compile_options(shardkv PRIVATE ${SHARDKV_WARNINGS})

# --- Тесты ---
include(FetchContent)
FetchContent_Declare(googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG v1.15.2)
FetchContent_MakeAvailable(googletest)

enable_testing()
add_executable(shardkv_tests tests/test_protocol.cpp
    tests/test_server.cpp)
target_link_libraries(shardkv_tests PRIVATE shardkv_lib
    GTest::gtest_main)
target_compile_options(shardkv_tests PRIVATE ${SHARDKV_WARNINGS})
include(GoogleTest)
gtest_discover_tests(shardkv_tests PROPERTIES TIMEOUT 30)
```

Листинг 1.21 — файл сборки CMake (CMakeLists.txt)

CMakePresets.json JSON

```json
{
  "version": 6,
  "configurePresets": [
    { "name": "base", "hidden": true, "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/${presetName}" },
    { "name": "debug",   "inherits": "base", "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug" } },
    { "name": "release", "inherits": "base", "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release" } },
    { "name": "tsan",    "inherits": "base",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug",
          "SHARDKV_SANITIZER": "thread" } },
    { "name": "asan",    "inherits": "base",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug",
          "SHARDKV_SANITIZER": "address" } }
  ],
  "buildPresets": [
    { "name": "debug", "configurePreset": "debug" },
    { "name": "release", "configurePreset": "release" },
    { "name": "tsan", "configurePreset": "tsan" },
    { "name": "asan", "configurePreset": "asan" }
  ],
  "testPresets": [
    { "name": "tsan", "configurePreset": "tsan", "output": {
        "outputOnFailure": true } },
    { "name": "asan", "configurePreset": "asan", "output": {
        "outputOnFailure": true } }
  ]
```

CMakePresets.json (продолжение) JSON

```
}
```

Листинг 1.22 — пресеты сборки CMake (CMakePresets.json)

test_protocol.cpp C++

```cpp
#include <gtest/gtest.h>

#include "core/protocol.hpp"

using shardkv::execute;
using shardkv::Storage;

TEST(Protocol, SetGetDel) {
    Storage s;
    EXPECT_EQ(execute(s, "GET name"), "$nil");
    EXPECT_EQ(execute(s, "SET name shardkv"), "+OK");
    EXPECT_EQ(execute(s, "get name"), "$shardkv");  // команды регистронезависимы
    EXPECT_EQ(execute(s, "DEL name"), ":1");
    EXPECT_EQ(execute(s, "DEL name"), ":0");
}

TEST(Protocol, IncrAndSetnx) {
    Storage s;
    EXPECT_EQ(execute(s, "INCR hits"), ":1");
    EXPECT_EQ(execute(s, "INCR hits"), ":2");
    EXPECT_EQ(execute(s, "SET word abc"), "+OK");
    EXPECT_EQ(execute(s, "INCR word"), "-ERR value is not an integer or out of range");
    EXPECT_EQ(execute(s, "SETNX lock me"), ":1");
    EXPECT_EQ(execute(s, "SETNX lock you"), ":0");
    EXPECT_EQ(execute(s, "GET lock"), "$me");
}
```

test_protocol.cpp (продолжение) C++

```cpp

TEST(Protocol, Errors) {
    Storage s;
    EXPECT_EQ(execute(s, ""), "-ERR empty command");
    EXPECT_EQ(execute(s, "GET"), "-ERR wrong number of arguments for GET");
    EXPECT_EQ(execute(s, "FLY away"), "-ERR unknown command FLY");
    EXPECT_EQ(execute(s, "PING\r"), "+PONG");  // \r от telnet отрезается
}
```

Листинг 1.23 — тесты (test_protocol.cpp)

test_server.cpp C++

```cpp
// Однопоточные с точки зрения хранилища тесты сервера: в каждый момент
// с хранилищем работает максимум одно соединение, поэтому гонки шага 1 тут не проявляются.
#include <gtest/gtest.h>

#include <chrono>
#include <future>
#include <string>
#include <thread>

#include "core/storage.hpp"
#include "net/socket.hpp"
#include "server/server.hpp"

using namespace std::chrono_literals;
namespace net = shardkv::net;

namespace {

std::string request(const net::Fd& conn, net::LineReader& reader,
    const std::string& line) {
    EXPECT_TRUE(net::send_all(conn, line + "\n"));
    auto reply = reader.next();
    return reply.value_or("<closed>");
}

```

test_server.cpp (продолжение) C++

```cpp
} // namespace

TEST(Server, AnswersAndStopsCleanly) {
    shardkv::Storage storage;
    shardkv::Server server(storage, 0);  // 0 — ОС выдаст свободный порт
    std::jthread acceptor([&server](std::stop_token st) { server.run(st); });

    {
        auto conn = net::connect_tcp(server.port());
        net::LineReader reader(conn);
        EXPECT_EQ(request(conn, reader, "PING"), "+PONG");
        EXPECT_EQ(request(conn, reader, "SET k v"), "+OK");
        EXPECT_EQ(request(conn, reader, "GET k"), "$v");
    }
    acceptor.request_stop();
    acceptor.join();  // если остановка сломана, тест повиснет и упадёт по таймауту ctest
    EXPECT_EQ(storage.size(), 1u);
}

TEST(Server, StopWakesIdleConnections) {
    shardkv::Storage storage;
    shardkv::Server server(storage, 0);
    std::jthread acceptor([&server](std::stop_token st) { server.run(st); });

```

test_server.cpp (продолжение) C++

```cpp
    // Клиент подключился и молчит: поток соединения висит в recv().
    auto idle = net::connect_tcp(server.port());
    net::LineReader reader(idle);
    EXPECT_EQ(request(idle, reader, "PING"), "+PONG");

    // Останавливаем сервер, НЕ закрывая клиента. Сервер обязан выйти сам.
    auto stopped = std::async(std::launch::async, [&] {
        acceptor.request_stop();
        acceptor.join();
    });
    ASSERT_EQ(stopped.wait_for(5s), std::future_status::ready) <<
        "сервер не остановился";

    EXPECT_FALSE(reader.next().has_value());  // сервер закрыл соединение со своей стороны
}
```

Листинг 1.24 — тесты (test_server.cpp)

Задачки для разминки мозгов

Правило самопроверки для всех задач книги: решение засчитывается, только если тесты проходят под TSan и ASan десять раз подряд (ctest --repeat until-fail:10). Многопоточный тест, прошедший один раз, ничего не доказывает.

Задача 1.1 (лёгкая). Найди три бага

В этом коде три бага. Код компилируется без предупреждений. Иногда даже печатает то, что нужно.

task1_1_bugs.cpp

```cpp
// Задача 1.1. В этом коде три бага. Код компилируется. Иногда даже работает.
#include <print>
#include <string>
#include <thread>
#include <vector>

void log_async(const std::string& message) {
    std::thread([&message] {
        std::println("[log] {}", message);
    }).detach();
}

int main() {
    std::vector<std::thread> workers;
    std::vector<int> results(4);

    for (int i = 0; i < 4; ++i) {
        workers.emplace_back([&] { results[i] = i * i; });
    }
    for (auto& w : workers) w.join();

    log_async("готово: " + std::to_string(results[3]));
}
```

Листинг 1.25 — код к задаче 1.1 (task1_1_bugs.cpp)

Найдите все три бага. Для каждого объясните в одной фразе, что именно сломается и при каких условиях. Затем перепишите код на std::jthread так, чтобы он был корректным, а log_async по-прежнему не блокировал вызывающий код.

Проверка: исправленный код чист под ASan и TSan; объяснение каждого бага укладывается в одну фразу и называет конкретную причину (не «потому что многопоточность»).

Задача 1.2 (средняя). parallel_sum с доставкой исключений

Реализуйте функцию:

фрагмент.cpp C++

```cpp
std::int64_t parallel_sum(std::span<const int> data,
                          std::size_t threads,
                          const std::function<std::int64_t(int)>&
                              transform);
```

Листинг 1.26 — сигнатура функции parallel_sum

Функция применяет transform к каждому элементу и возвращает сумму результатов. Работа делится на threads примерно равных частей, каждая обрабатывается в своём потоке. Каждый поток пишет результат в свою ячейку, без общего счётчика. Если transform бросает исключение в любом потоке, parallel_sum дожидается всех потоков и бросает это исключение вызывающему коду.

Учтите граничные случаи: пустой data, threads == 0, threads больше, чем элементов.

Проверка: результат совпадает с std::transform_reduce на случайных данных; тест, в котором transform бросает на одном элементе, получает именно это исключение; ни один поток не остаётся работать после выхода из функции.

Задача 1.3 (сложная). Сервис, который нельзя сломать деструктором

Напишите класс TickService:

— конструктор принимает число воркеров N и запускает N потоков;

— каждый воркер увеличивает свою ячейку в std::vector<std::uint64_t>, который является членом класса, пока его не остановят;

— метод stop_and_total() останавливает всех воркеров и возвращает сумму ячеек;

— деструктор корректно останавливает воркеров, даже если stop_and_total() не вызывался;

— для остановки используется один общий std::stop_source.

Главное требование: класс должен быть корректным при любом сценарии использования, включая «создали и сразу уничтожили». Подсказка: обратите внимание, в каком порядке разрушаются члены класса и что из этого следует для потоков, которые к ним обращаются. Объясните, почему ваш порядок членов правильный, а обратный — нет.

Проверка: тест «создать и сразу уничтожить» 1000 раз подряд проходит под ASan и TSan; тест stop_and_total() возвращает ненулевую сумму после короткой работы; повторный вызов stop_and_total() не ломает объект.

Решения и разбор ваших неизбежных ошибок — в главе «Ответы к задачам» в конце книги.

Глава 2. Data race, мьютексы и тот самый deadlock

«Там же просто int, что может пойти не так?» — надпись на надгробии одного сервиса.

В прошлой главе мы оставили shardkv с бомбой внутри: несколько потоков одновременно меняют один std::unordered_map. На тестах всё работает. Под четырьмя клиентами ThreadSanitizer кричит «data race». Сегодня мы разберёмся, почему это не «предупреждение, которое можно игнорировать», а неопределённое поведение, и научимся его лечить.

Лекарство — мьютексы. У лекарства есть побочные эффекты: дедлоки, медленный код и интерфейсы, которые потокобезопасны по отдельности и сломаны вместе. Их мы тоже разберём.

К концу главы вы:

— будете отличать data race от race condition (на собеседованиях их путают постоянно);
— научитесь выбирать между lock_guard, unique_lock, scoped_lock и shared_lock;
— поймёте, почему защищать нужно данные и инварианты, а не строчки кода;
— научитесь проектировать интерфейс, в котором нельзя устроить гонку снаружи;
— узнаете четыре условия дедлока и как сломать хотя бы одно;
— выясните, что в стандартной библиотеке потокобезопасно, а что только притворяется;
— сделаете хранилище shardkv потокобезопасным и впервые измерите сервер нагрузкой.

2.1. Data race: почему это UB, а не «иногда не то значение»

Начнём с самой короткой гонки в мире:

ex01_race.cpp

```cpp
// ex01_race.cpp — самая короткая гонка данных в мире и её последствия.
// Соберите БЕЗ санитайзера с -O0 и запустите несколько раз. Потом соберите с -fsanitize=thread.
#include <print>
#include <thread>
#include <vector>

int main() {
    constexpr int kThreads = 4;
    constexpr int kIncrements = 1'000'000;

    long counter = 0;  // общая переменная без защиты

    {
        std::vector<std::jthread> threads;
        for (int t = 0; t < kThreads; ++t) {
            threads.emplace_back([&counter] {
                for (int i = 0; i < kIncrements; ++i) {
                    ++counter;  // DATA RACE: чтение-изменение-запись из нескольких потоков
                }
            });
        }
    }
```

ex01_race.cpp (продолжение)

```cpp
    std::println("ожидали {}, получили {}", kThreads * kIncrements, counter);
    // Типичный вывод на -O0: «ожидали 4000000, получили 1873402».
    // На -O2 компилятор может свернуть цикл в counter += 1000000 — и «всё работает».
    // Это не исправление. Это UB, которому сегодня повезло.
}
```

Листинг 2.1 — самая короткая гонка данных в мире и её последствия (ex01_race.cpp)

Четыре потока по миллиону раз делают ++counter. Ожидаем 4 000 000. На моей машине в сборке -O0 получилось 2 176 855. Почти половина инкрементов исчезла.

Почему? ++counter — это не одна операция, а три: прочитать значение из памяти в регистр, увеличить регистр, записать обратно. Два потока читают 41, оба увеличивают до 42, оба записывают 42. Один инкремент потерян.

Но объяснение «операции перемешались» — это только половина правды, и, честно говоря, менее страшная половина. Стандарт C++ определяет data race так: два обращения к одной ячейке памяти из разных потоков, хотя бы одно из них — запись, хотя бы одно не атомарное, и между ними нет отношения happens-before (то есть никакой синхронизации). И дальше стандарт говорит: программа с data race имеет **неопределённое поведение**.

Не «может вернуть неправильное значение». Неопределённое поведение. Компилятор оптимизирует код, исходя из предположения, что гонок нет. Он может держать переменную в регистре и никогда не перечитывать её из памяти. Может свернуть цикл в одну запись. Может переставить операции местами. Поэтому с -O2 этот же пример иногда выдаёт ровно 4 000 000: компилятор превратил цикл в counter += 1000000, и окно для гонки стало крошечным. Это не исправление. Это UB, которому сегодня повезло.

Классический пример того, как это стреляет:

фрагмент.cpp — C++

```cpp
bool ready = false;  // не атомик
// Поток 1:
while (!ready) { }   // компилятор вправе прочитать ready один раз и
    крутиться вечно
// Поток 2:
ready = true;
```

Листинг 2.2 — гонка на обычном bool

Компилятор видит цикл, в котором ready не меняется (по его мнению, ведь гонок нет), выносит чтение из цикла, и поток 1 зависает навсегда. На -O0 работает, на -O2 висит. Отлаживать такое в продакшене — отдельный вид удовольствия.

Вывод: data race — это не «маленькая неточность», а ошибка того же класса, что выход за границу массива. Её не терпят «потому что значение не критичное». Её исправляют.

Data race и race condition — разные вещи

— **Data race** — формальное понятие стандарта: несинхронизированный конкурентный доступ с записью. Всегда UB. Ловится ThreadSanitizer.

— **Race condition** — логическая ошибка: результат зависит от того, в каком порядке выполнились потоки, и некоторые порядки дают неправильный результат. Может существовать без всякого data race: все обращения под мьютексом, а логика всё равно сломана. TSan такие ошибки **не ловит**.

Пример race condition без data race будет в разделе 2.4. Запомните разницу сейчас: на собеседовании вас о ней спросят.

2.2. std::mutex и RAII-обёртки

Мьютекс (mutual exclusion) — объект, который в каждый момент может быть захвачен только одним потоком. Остальные, кто пытается его захватить, ждут. Код между захватом и освобождением называется критической секцией: в ней в каждый момент находится не больше одного потока.

ex02_mutex.cpp C++

```cpp
// ex02_mutex.cpp — тот же счётчик, но честный. И четыре способа держать мьютекс.
#include <mutex>
#include <print>
#include <thread>
#include <vector>

int main() {
    constexpr int kThreads = 4;
    constexpr int kIncrements = 1'000'000;

    std::mutex m;
    long counter = 0;  // защищён мьютексом m — и это стоит написать в комментарии

    {
        std::vector<std::jthread> threads;
        for (int t = 0; t < kThreads; ++t) {
            threads.emplace_back([&] {
                for (int i = 0; i < kIncrements; ++i) {
                    std::lock_guard lock(m);  // захват в конструкторе, освобождение в деструкторе
                    ++counter;
                }
            });
        }
    }
```

ex02_mutex.cpp (продолжение) C++

```cpp
    std::println("ожидали {}, получили {}", kThreads * kIncrements,
        counter);

    // lock_guard — самый простой: захватил и держит до конца
        области видимости.
    // unique_lock — можно отпустить раньше, захватить позже,
        передать владение,
    //               нужен для condition_variable (глава 3):
    {
        std::unique_lock lock(m);
        ++counter;
        lock.unlock();  // дальше работаем без лока
        // ... долгая работа, которой не нужен мьютекс ...
    }
    // scoped_lock — для одного или НЕСКОЛЬКИХ мьютексов сразу, без
        дедлока (раздел 2.5).
    // m.lock()/m.unlock() руками — только если вы пишете свою
        RAII-обёртку.
}
```

Листинг 2.3 — тот же счётчик, но честный (ex02_mutex.cpp)

Теперь результат всегда 4 000 000. Мьютекс даёт сразу две гарантии, и вторая важнее, чем кажется:

— **взаимное исключение:** в критической секции один поток;

— **видимость:** всё, что поток записал до unlock(), гарантированно увидит поток, который после этого сделал lock(). Освобождение мьютекса синхронизируется с его следующим захватом. Без этой гарантии второй поток мог бы прочитать устаревшее значение из кеша процессора или регистра. Формально это то самое отношение happens-before, разберём его в главе 5.

Руками вызывать m.lock() и m.unlock() не нужно: если между ними вылетит исключение или кто-то добавит ранний return, мьютекс останется захваченным навсегда, и следующий поток повиснет. Для этого есть RAII-обёртки:

— **std::lock_guard** — захватывает в конструкторе, освобождает в деструкторе. Ничего больше не умеет, и это хорошо. Выбор по умолчанию для одного мьютекса.

— **std::unique_lock** — может отпустить мьютекс раньше конца области видимости, захватить позже (std::defer_lock), попробовать захватить с таймаутом, передать владение. Нужен для condition_variable (глава 3). Чуть дороже lock_guard: хранит флаг владения.

— **std::scoped_lock** (C++17) — захватывает один или несколько мьютексов сразу, без риска дедлока. Для одного мьютекса работает как lock_guard, поэтому многие используют его везде.

— **std::shared_lock** — разделяемый захват shared_mutex (раздел 2.6).

Благодаря CTAD (вывод аргументов шаблона класса) тип мьютекса писать не нужно: std::lock_guard lock(m) вместо std::lock_guard<std::mutex> lock(m).

Сколько стоит мьютекс

Незахваченный мьютекс в Linux стоит очень дёшево: одна атомарная операция на захват и одна на освобождение, без системных вызовов (как это устроено через futex, разберём в главе 4). Дорогим мьютекс становится при **конкуренции** (contention): когда потоки постоянно сталкиваются на нём, они уходят спать в ядро, просыпаются, переключают контекст. Отсюда главное правило:

Держите мьютекс как можно меньше. Всё, что можно сделать до захвата (аллокации, вычисления, разбор строк) или после освобождения (логирование, отправка по сети), делайте снаружи критической секции. В хранилище shardkv мы так и сделаем: строки для ключа и значения создаются до захвата лока.

2.3. Защищайте данные, а не код

Типичная ошибка новичка: «вот эта функция вызывается из нескольких потоков, обложу-ка её мьютексом». Мьютекс защищает не функцию, а **данные и инварианты**. Если те же данные меняются в другой функции без того же мьютекса, защита первой бессмысленна.

Правило: у каждого общего изменяемого объекта должен быть ровно один мьютекс, который его охраняет, и это должно быть очевидно из кода. Лучший способ сделать это очевидным — не давать доступа к данным без мьютекса вообще:

ex03_synchronized.cpp C++

```cpp
// ex03_synchronized.cpp — защищаем данные, а не код.
// Synchronized<T> делает невозможным доступ к данным без мьютекса:
    забыть лок нельзя.
#include <map>
#include <mutex>
#include <print>
#include <string>
#include <thread>
#include <utility>
#include <vector>

template <class T>
class Synchronized {
public:
    template <class... Args>
    explicit Synchronized(Args&&... args) :
        value_(std::forward<Args>(args)...) {}

    // Единственный способ добраться до данных — передать функцию,
        которая выполнится под локом.
    template <class F>
    decltype(auto) with_lock(F&& f) {
        std::lock_guard lock(mutex_);
        return std::forward<F>(f)(value_);
    }

```

ex03_synchronized.cpp (продолжение) C++

```cpp
    template <class F>
    decltype(auto) with_lock(F&& f) const {
        std::lock_guard lock(mutex_);
        return std::forward<F>(f)(std::as_const(value_));
    }

private:
    mutable std::mutex mutex_;  // mutable: блокировать нужно и в const-методах
    T value_;
};

int main() {
    Synchronized<std::map<std::string, int>> word_count;

    {
        std::vector<std::jthread> workers;
        for (int t = 0; t < 4; ++t) {
            workers.emplace_back([&word_count] {
                for (int i = 0; i < 1000; ++i) {
                    word_count.with_lock([](auto& m) { ++m["кофе"]; });
                }
            });
        }
    }
```

ex03_synchronized.cpp (продолжение) C++

```cpp
    // Возвращаем КОПИЮ значения, а не ссылку: ссылка, вынесенная из-под лока, — та же гонка.
    const int total = word_count.with_lock([](const auto& m) { return m.at("кофе"); });
    std::println("кофе упомянут {} раз", total);

    // АНТИПАТТЕРН (не делайте так):
    // auto* leaked = word_count.with_lock([](auto& m) { return &m; });
    // leaked->clear();  // мьютекс уже отпущен — data race
}
```

Листинг 2.4 — защищаем данные, а не код (ex03_synchronized.cpp)

Synchronized<T> — маленькая обёртка, которая делает «забыть мьютекс» невозможным: до данных можно добраться только через with_lock, а он захватывает мьютекс. Подобные классы есть в folly (folly::Synchronized) и Boost (boost::synchronized_value).

Но и тут есть дыра: если функция вернёт ссылку или указатель на защищённые данные, этот указатель будет жить после освобождения мьютекса. Любое обращение через него — data race. Правило: **из-под лока выносят копии, а не ссылки.** То же относится к хранилищу shardkv: get() возвращает копию строки, а не const std::string&.

Ещё одна дыра — вызов пользовательского кода под локом. Если вы под мьютексом вызываете колбэк, а колбэк вызывает метод того же объекта, который захватывает тот же мьютекс, это дедлок (std::mutex нерекурсивен). А если колбэк захватывает другой мьютекс, вы получаете неконтролируемый порядок захвата (раздел 2.5). Правило: **не вызывайте под локом код, которого не контролируете.**

2.4. Потокобезопасный интерфейс: когда каждый метод безопасен, а класс сломан

Вот обещанный пример race condition без data race:

ex04_check_then_act.cpp C++

```cpp
// ex04_check_then_act.cpp — каждый метод потокобезопасен, а класс всё равно сломан.
#include <mutex>
#include <print>
#include <thread>
#include <vector>

// ПЛОХО: потокобезопасные «кирпичики», из которых нельзя собрать корректную операцию.
class WalletBad {
public:
    long balance() const { std::lock_guard l(m_); return balance_; }
    void withdraw(long amount) { std::lock_guard l(m_); balance_ -= amount; }
private:
    mutable std::mutex m_;
    long balance_ = 100;
};

// ХОРОШО: проверка и действие — одна операция под одним локом.
class Wallet {
public:
    // Возвращает true, если списали. Нельзя уйти в минус ни при каком чередовании потоков.
    bool try_withdraw(long amount) {
        std::lock_guard l(m_);
        if (balance_ < amount) return false;
        balance_ -= amount;
```

ex04_check_then_act.cpp (продолжение) C++

```
        return true;
    }
    long balance() const { std::lock_guard l(m_); return balance_; }
private:
    mutable std::mutex m_;
    long balance_ = 100;
};

int main() {
    WalletBad bad;
    Wallet good;
    {
        std::vector<std::jthread> buyers;
        for (int i = 0; i < 8; ++i) {
            buyers.emplace_back([&] {
                // Race condition: между balance() и withdraw()
                    другой поток успевает списать.
                // Data race тут нет — TSan промолчит. А деньги
                    уйдут в минус.
                if (bad.balance() >= 30) bad.withdraw(30);
                good.try_withdraw(30);
            });
        }
    }
    std::println("WalletBad: {} (может быть отрицательным)",
        bad.balance());
    std::println("Wallet:    {} (никогда не меньше нуля)",
        good.balance());
```

ex04_check_then_act.cpp (продолжение) C++

```
}
```

Листинг 2.5 — каждый метод потокобезопасен, а класс всё равно сломан (ex04_check_then_act.cpp)

У WalletBad каждый метод захватывает мьютекс. Data race нет, TSan промолчит. Но код «проверь баланс, потом спиши» состоит из двух операций, и между ними другой поток успевает списать деньги. Восемь покупателей одновременно видят баланс 100, и каждый списывает 30. Итог −140.

Это ошибка проектирования интерфейса, а не реализации. Мьютекс внутри методов не может защитить последовательность вызовов снаружи. Решение — сделать операцию, которую нужно выполнить атомарно, одним методом: try_withdraw проверяет и списывает под одним захватом.

Классический пример того же класса ошибок — стек или очередь с методами empty(), top() и pop():

фрагмент.cpp C++

```cpp
if (!stack.empty()) {      // проверили: не пусто
    auto x = stack.top();  // другой поток уже забрал последний
        элемент — UB
    stack.pop();
}
```

Листинг 2.6 — проверка и действие по отдельности не атомарны

Каждый метод может быть потокобезопасным, а связка всё равно сломана. Правильный интерфейс объединяет проверку и действие: std::optional<T> try_pop(). Именно такой стек вы напишете в задаче 2.2.

Признаки сломанного потокобезопасного интерфейса на ревью:

— методы-«вопросы» (empty, size, contains, count), результат которых используется для принятия решения;

— методы, возвращающие ссылки на внутренние данные;

— необходимость вызвать два метода подряд, чтобы выполнить одно логическое действие.

В shardkv по этой причине INCR и SETNX реализованы внутри хранилища под одним захватом, а не через get() + set() в обработчике протокола.

2.5. Deadlock: четыре условия и как сломать хотя бы одно

Дедлок — ситуация, когда потоки ждут друг друга по кругу и никто не может продвинуться. Программа не падает, не пишет ошибок, не жрёт CPU. Она просто стоит. Мониторинг в это время показывает, что «всё зелёное», потому что процесс жив.

Дедлок возможен, только если одновременно выполнены четыре условия Коффмана:

1. **Взаимное исключение:** ресурс может держать только один поток.
2. **Удержание и ожидание:** поток держит один ресурс и ждёт другой.
3. **Нет принудительного освобождения:** ресурс нельзя отобрать, его можно только отдать.
4. **Циклическое ожидание:** A ждёт B, B ждёт A (или длиннее цикл).

Первые три для мьютексов выполнены всегда, по определению. Значит, бороться можно только с четвёртым: не допускать циклов.

ex05_deadlock.cpp C++

```cpp
// ex05_deadlock.cpp — два мьютекса, два потока, вечность ожидания.
    И как этого избежать.
#include <mutex>
#include <print>
#include <string>
#include <thread>
#include <utility>

struct Document {
    std::mutex m;
    std::string text;
};

// ПЛОХО: порядок захвата зависит от порядка аргументов.
// Поток 1: merge_bad(a, b) захватил a.m, ждёт b.m.
// Поток 2: merge_bad(b, a) захватил b.m, ждёт a.m. Всё. Навсегда.
void merge_bad(Document& to, Document& from) {
    std::lock_guard l1(to.m);
    std::lock_guard l2(from.m);
    to.text += from.text;
}

// ХОРОШО: std::scoped_lock захватывает несколько мьютексов
    алгоритмом,
// который не допускает дедлока при любом порядке аргументов.
void merge(Document& to, Document& from) {
    if (&to == &from) return;  // один и тот же мьютекс дважды — UB
        для std::mutex
```

ex05_deadlock.cpp (продолжение) C++

```cpp
    std::scoped_lock lock(to.m, from.m);
    to.text += from.text;
    from.text.clear();
}

int main() {
    Document a{.m = {}, .text = "A"};
    Document b{.m = {}, .text = "B"};
    {
        std::jthread t1([&] { for (int i = 0; i < 10'000; ++i)
            merge(a, b); });
        std::jthread t2([&] { for (int i = 0; i < 10'000; ++i)
            merge(b, a); });
    }
    std::println("длина итогового текста: {}", a.text.size() +
        b.text.size());
    // Замените merge на merge_bad — и программа, скорее всего,
        зависнет. Проверьте сами.
}
```

Листинг 2.7 — два мьютекса, два потока, вечность ожидания (ex05_deadlock.cpp)

Функция merge_bad захватывает мьютексы в порядке аргументов. Поток 1 вызывает merge_bad(a, b), поток 2 — merge_bad(b, a). Поток 1 захватил a и ждёт b, поток 2 захватил b и ждёт a. Цикл.

Способы не допустить цикла:

— **std::scoped_lock с несколькими мьютексами** (или std::lock). Внутри используется алгоритм, который захватывает мьютексы так, чтобы дедлок был невозможен при любом порядке аргументов: если не удаётся захватить все, он отпускает уже захваченные и пробует снова. Самый простой и надёжный способ, когда все нужные мьютексы известны заранее.

— **Глобальный порядок захвата.** Все мьютексы упорядочены (например, по идентификатору объекта), и любой код захватывает их только по возрастанию. Тогда цикл невозможен. Работает, даже когда мьютексы захватываются в разных функциях в разное время, но требует дисциплины.

— **Иерархия блокировок.** Каждому мьютексу назначается уровень, и поток, держащий мьютекс уровня N, может захватывать только мьютексы уровня ниже N. Нарушение сразу видно, а не «когда-нибудь под нагрузкой». Такой мьютекс вы напишете в задаче 2.3.

— **Не держать два мьютекса одновременно.** Самый надёжный способ. Если архитектура позволяет — лучшее решение.

Отдельно про вызов merge(a, a): попытка захватить std::mutex, который уже держит текущий поток, — это UB (на практике обычно дедлок). Поэтому в merge есть проверка на один и тот же объект.

2.6. shared_mutex: много читателей, редкий писатель

Если данные в основном читают и редко меняют, обычный мьютекс заставляет читателей ждать друг друга, хотя читать одновременно безопасно. std::shared_mutex позволяет захватить его двумя способами:

— **разделяемо** (std::shared_lock) — сколько угодно читателей одновременно;
— **эксклюзивно** (std::unique_lock или std::lock_guard) — один писатель, читателей нет.

ex06_shared_mutex.cpp C++

```cpp
// ex06_shared_mutex.cpp — много читателей, редкий писатель.
#include <mutex>
#include <optional>
#include <print>
#include <shared_mutex>
#include <string>
#include <thread>
#include <unordered_map>
#include <vector>

class Config {
public:
    std::optional<std::string> get(const std::string& key) const {
        std::shared_lock lock(m_);  // разделяемый захват: читатели не мешают друг другу
        if (auto it = values_.find(key); it != values_.end()) return it->second;
        return std::nullopt;
    }
    void set(std::string key, std::string value) {
        std::unique_lock lock(m_);  // эксклюзивный захват: писатель один, читателей нет
        values_.insert_or_assign(std::move(key), std::move(value));
    }
private:
    mutable std::shared_mutex m_;
    std::unordered_map<std::string, std::string> values_;
};
```

ex06_shared_mutex.cpp (продолжение) C++

```cpp

int main() {
    Config cfg;
    cfg.set("mode", "fast");
    {
        std::vector<std::jthread> readers;
        for (int i = 0; i < 4; ++i) {
            readers.emplace_back([&cfg] {
                for (int j = 0; j < 10'000; ++j)
                    (void)cfg.get("mode");
            });
        }
        std::jthread writer([&cfg] { cfg.set("mode", "safe"); });
    }
    std::println("mode = {}", cfg.get("mode").value_or("?"));
    // shared_mutex дороже обычного mutex. Он выигрывает, только когда чтения
    // много, оно долгое, а записей мало. Короткое чтение map — не тот случай. Меряйте.
}
```

Листинг 2.8 — много читателей, редкий писатель (ex06_shared_mutex.cpp)

Звучит как бесплатное ускорение, но это не так:

— shared_mutex дороже обычного: даже разделяемый захват меняет общий счётчик читателей, и при большом числе ядер этот счётчик сам становится горячей точкой (почему — в главе 4);

— если критическая секция короткая (поиск в хеш-таблице), накладные расходы shared_mutex могут съесть весь выигрыш;

— поведение при постоянном потоке читателей зависит от реализации: писатель может долго ждать.

Правило: shared_mutex выигрывает, когда чтение долгое, читателей много, а записи редкие. Во всех остальных случаях — меряйте. В главе 4 мы сделаем это на shardkv.

2.7. recursive_mutex — запах кода

std::recursive_mutex позволяет одному потоку захватить мьютекс несколько раз (и отпустить столько же раз). Обычно его берут, когда публичный метод под локом вызывает другой публичный метод, который тоже берёт лок.

Это почти всегда признак плохой структуры: класс не понимает, в каком состоянии его инварианты в момент вызова. Если метод A захватил мьютекс и временно нарушил инвариант, а потом вызвал метод B, рассчитывающий на целый инвариант, recursive_mutex этот баг не исправит, а спрячет.

Правильное решение: разделить методы на публичные (берут лок) и приватные (предполагают, что лок уже взят, часто с суффиксом _locked). Публичные вызывают приватные, приватные никогда не вызывают публичные.

2.8. Таймауты: timed_mutex и правильные часы

Иногда ждать мьютекс вечно нельзя: лучше сообщить «занято» и попробовать позже. Для этого есть std::timed_mutex с методами try_lock_for и try_lock_until:

ex07_timeouts_once.cpp

```cpp
// ex07_timeouts_once.cpp — захват с таймаутом и однократная инициализация.
#include <chrono>
#include <mutex>
#include <print>
#include <string>
#include <thread>

using namespace std::chrono_literals;

std::timed_mutex device;

void try_use_device(int id) {
    // try_lock_for меряет время по steady_clock: часы, которые не прыгают.
    // system_clock может перевести NTP или админ — и ваш «таймаут 2 секунды» станет часом.
    std::unique_lock lock(device, std::defer_lock);
    if (lock.try_lock_for(50ms)) {
        std::println("поток {}: устройство моё", id);
        std::this_thread::sleep_for(100ms);
    } else {
        std::println("поток {}: устройство занято, попробую позже", id);
    }
}

```

ex07_timeouts_once.cpp (продолжение) C++

```cpp
// Однократная инициализация, вариант 1: локальная статическая переменная.
// С C++11 её инициализация потокобезопасна («magic statics»).
const std::string& app_name() {
    static const std::string name = [] {
        std::println("инициализация app_name (один раз)");
        return std::string("shardkv");
    }();
    return name;
}

// Вариант 2: std::call_once — когда инициализация не привязана к одной функции.
std::once_flag init_flag;
void init_logging() { std::println("логгер инициализирован (один раз)"); }

int main() {
    {
        std::jthread a(try_use_device, 1);
        std::jthread b(try_use_device, 2);
    }
    {
        std::jthread x([] { (void)app_name();
            std::call_once(init_flag, init_logging); });
        std::jthread y([] { (void)app_name();
            std::call_once(init_flag, init_logging); });
    }
```

ex07_timeouts_once.cpp (продолжение) C++

```cpp
}
```

Листинг 2.9 — захват с таймаутом и однократная инициализация (ex07_timeouts_once.cpp)

Две важные детали.

Используйте steady_clock для таймаутов. std::chrono::system_clock — это «настенные часы»: их может перевести NTP, администратор или переход на летнее время. Таймаут «до 12:00:05» превратится в час ожидания, если часы откатили назад. steady_clock монотонны и не прыгают. try_lock_for с длительностью сам использует steady_clock, а вот в try_lock_until вы передаёте момент времени, и часы выбираете сами — выбирайте steady_clock.

Таймаут на мьютексе — часто симптом. Если поток не может дождаться мьютекса за разумное время, обычно проблема в том, что кто-то держит его слишком долго. Таймаут — способ деградировать, а не способ лечить.

И практическая ремарка про инструменты. В некоторых связках GCC и libtsan этот пример вызывает ложное срабатывание TSan: «unlock of an unlocked mutex». Причина: try_lock_for в libstdc++ вызывает pthread_mutex_clocklock, а libtsan этот вызов не перехватывает и не знает, что мьютекс был захвачен. Это ошибка инструмента, а не кода. Лечится обновлением тулчейна или файлом подавлений:

tsan.supp TEXT

```
1 mutex:std::timed_mutex::unlock
```

Листинг 2.10 — файл подавлений TSan (tsan.supp)

терминал SHELL

```
$ TSAN_OPTIONS=suppressions=tsan.supp ./program
```

Листинг 2.11 — запуск с файлом подавлений

Подавления — это последнее средство: каждая строка в таком файле должна иметь комментарий, почему это ложное срабатывание. Подробнее о работе с TSan в CI поговорим в главе 13.

2.9. Однократная инициализация

Ленивая инициализация («создадим объект при первом обращении») — классическое место для гонок. Два потока одновременно видят, что объекта ещё нет, и оба его создают.

С C++11 есть два правильных способа (оба в примере выше):

— **локальная статическая переменная.** Стандарт гарантирует, что её инициализация выполнится ровно один раз, даже если функцию вызывают из многих потоков одновременно; остальные потоки подождут. Это называют «magic statics». Самый простой способ, и почти всегда достаточный;

— **std::call_once с std::once_flag** — когда инициализация не привязана к одной функции, или флаг должен быть членом класса.

И отдельно о том, чего делать не надо: «блокировка с двойной проверкой» (double-checked locking) на обычном указателе. Это data race на чтении указателя без лока. Правильная версия требует атомиков, и мы разберём её в главе 5. А пока: magic statics.

2.10. Что потокобезопасно в стандартной библиотеке

Общее правило стандарта: **конкурентный вызов const-методов одного объекта безопасен, если тип не говорит обратного; конкурентная запись или запись одновременно с чтением — гонка.** Разные объекты можно использовать из разных потоков свободно.

ex08_stdlib.cpp C++

```cpp
// ex08_stdlib.cpp — что в стандартной библиотеке потокобезопасно, а что притворяется.
#include <chrono>
#include <format>
#include <iostream>
#include <random>
#include <syncstream>
#include <thread>
#include <vector>

int random_dice() {
    // rand() использует скрытое глобальное состояние: не потокобезопасен.
    // Генератор на поток — без мьютекса и без гонок.
    thread_local std::mt19937 gen{std::random_device{}()};
    return std::uniform_int_distribution{1, 6}(gen);
}

int main() {
    const std::vector<int> table{1, 2, 3, 4, 5};
    {
        std::vector<std::jthread> threads;
        for (int t = 0; t < 4; ++t) {
            threads.emplace_back([&table, t] {
                // Конкурентное ЧТЕНИЕ контейнера через const-методы безопасно.
                // Одновременная запись и чтение — уже гонка.
                const int sum = table[0] + table.back();
```

ex08_stdlib.cpp (продолжение) C++

```
26
27                  // std::cout не даст data race, но строки разных
                        потоков перемешаются.
28                  // std::osyncstream (C++20) копит вывод и сбрасывает
                        его целиком.
29                  std::osyncstream(std::cout) << "поток " << t << ":
                        sum=" << sum
30                                              << " dice=" <<
                                    random_dice() << '\n';
31
32                  // localtime/strtok/setlocale — глобальное
                        состояние. Вместо localtime —
33                  // std::chrono: std::format форматирует время без
                        общих буферов.
34                  const auto now = std::chrono::floor<std::chrono::seco
                        nds>(std::chrono::system_clock::now());
35                  std::osyncstream(std::cout) << std::format("поток
                        {}: {:%F %T} UTC\n", t, now);
36              });
37          }
38      }
39  }
```

Листинг 2.12 — что в стандартной библиотеке потокобезопасно, а что притворяется (ex08_stdlib.cpp)

Конкретно:

— **Контейнеры.** Читать один std::vector или std::map из многих потоков можно, если никто в это время не пишет. Осторожно с «читающими» методами, которые на самом деле пишут: operator[] у std::map вставляет элемент, если ключа нет.

— **std::cout и другие стандартные потоки.** Data race не будет, но вывод разных потоков может перемешаться посимвольно. std::osyncstream (C++20) накапливает вывод и сбрасывает его одним куском.

— **rand().** Использует скрытое глобальное состояние, не потокобезопасен. Используйте генератор из <random>, по одному на поток (thread_local).

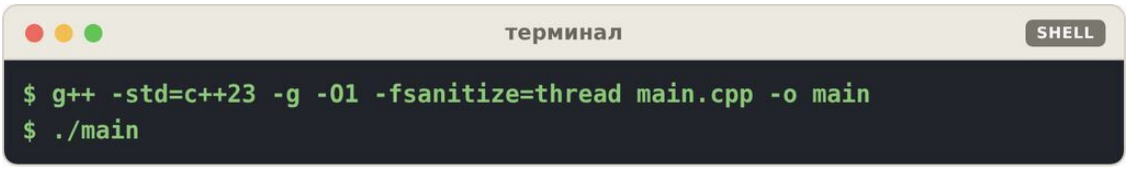
— **localtime(), gmtime(), strtok(), setlocale(), getenv() при одновременном setenv().** Работают с глобальным или статическим состоянием. Для времени используйте std::chrono и std::format, вместо strtok — std::string_view и std::views::split.

— **std::shared_ptr.** Тут хитро: счётчик ссылок атомарный, а объект и сам указатель — нет. Это отдельный большой разговор в главе 5.

2.11. ThreadSanitizer: первый запуск

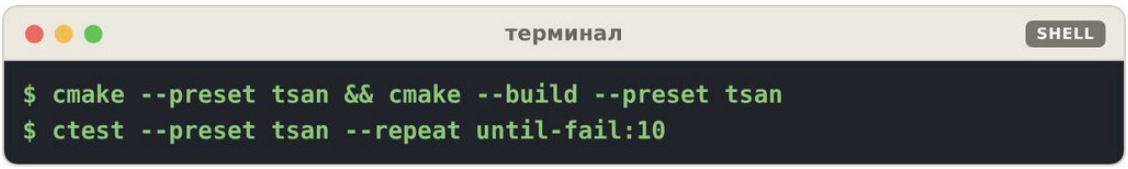
ThreadSanitizer (TSan) — инструмент, встроенный в GCC и Clang, который находит data race во время выполнения программы. Он отслеживает каждое обращение к памяти и каждую синхронизацию и сообщает, если два обращения не упорядочены через happens-before.

Как включить:



Листинг 2.13 — сборка и запуск под TSan вручную

В нашем проекте это пресет tsan:



Листинг 2.14 — пресет tsan: сборка и десять прогонов тестов

Как читать отчёт:

— **WARNING: ThreadSanitizer: data race** — тип проблемы;

— **Write of size 8… by thread T2** и стек вызовов — первое обращение;

— **Previous read/write… by thread T1** и стек — второе, конфликтующее обращение;

— **Location is …** — где находится память (глобальная переменная, куча, стек какого потока);

— **Thread T2 created at** — где был создан каждый поток.

Что нужно знать про TSan:

— он находит только гонки, которые **реально произошли** в этом запуске. Если код с гонкой не выполнился, TSan промолчит. Поэтому тесты под TSan гоняют многократно и под нагрузкой;

— он не находит race condition без data race (как WalletBad);

— замедляет программу в 5–15 раз и увеличивает потребление памяти в 5–10 раз — в продакшен его не ставят, только в тесты и CI;

— TSan нельзя совмещать с ASan в одной сборке, поэтому у нас два отдельных пресета;

— ложные срабатывания редки, но бывают (раздел 2.8). Ложным срабатывание можно считать только после того, как вы можете объяснить, какая синхронизация его исключает.

2.12. Грабли с ревью

— «Это просто bool/int, гонка тут безвредна». Безвредных data race не бывает, бывает UB, которое ещё не выстрелило.

— Мьютекс, который защищает функцию, а не данные. Те же данные меняются в другом месте без лока.

— Ссылка или указатель на данные, вынесенные из-под лока.

— Пары «проверка + действие» через два публичных вызова. empty() + pop(), contains() + insert(), get() + set().

— Вызов колбэков, виртуальных функций или чужого кода под локом.

— Захват двух мьютексов по очереди через два lock_guard. Используйте std::scoped_lock.

— Тяжёлая работа под локом: логирование, сетевые вызовы, аллокации, которые можно вынести.

— recursive_mutex «чтобы не думать».

— Таймауты на system_clock.

— Ручные m.lock()/m.unlock() вне RAII-обёрток.

2.13. Что спросят на собеседовании

Что такое data race и чем он отличается от race condition? (middle) Data race — несинхронизированный конкурентный доступ к одной памяти, хотя бы один из которых запись; по стандарту UB. Race condition — логическая зависимость результата от порядка выполнения; может быть и без data race. Хороший ответ приводит пример второго без первого.

Почему ++counter из двух потоков — это не просто «потеря инкрементов»? (middle/senior) Это UB: компилятор оптимизирует код в предположении, что гонок нет, и может, например, вынести чтение из цикла. Потеря инкрементов — лишь одно из проявлений.

Чем lock_guard отличается от unique_lock и scoped_lock? (junior/middle) lock_guard — простейший RAII; unique_lock — гибкий (отложенный захват, ранний unlock, таймауты, передача владения, нужен для condition_variable); scoped_lock — несколько мьютексов без дедлока.

Назовите условия дедлока и способы его избежать. (middle) Четыре условия Коффмана; на практике ломаем циклическое ожидание: std::scoped_lock, глобальный порядок захвата, иерархия блокировок, отказ от удержания двух мьютексов.

Почему интерфейс стека с empty(), top(), pop() плох в многопоточном коде? (middle) Последовательность вызовов не атомарна, между ними другой поток меняет состояние. Решение — try_pop(), возвращающий std::optional<T>.

Когда shared_mutex медленнее обычного mutex? (senior) При коротких критических секциях и большом числе ядер: разделяемый захват всё равно пишет в общий счётчик, кешлиния с ним прыгает между ядрами. Сильный ответ: «меряю, а не угадываю».

Потокобезопасна ли инициализация локальной статической переменной? (middle) Да, с C++11 (magic statics). Бонус: рассказать, почему double-checked locking на обычном указателе — data race.

Можно ли читать std::vector из нескольких потоков одновременно? (junior/middle) Да, через const-методы, если никто не пишет. Бонус: operator[] у std::map пишет.

2.14. Сквозной проект shardkv, шаг 2: глобальный мьютекс и первый бенчмарк

Что делаем:

— убеждаемся, что TSan видит гонку шага 1 (мы это сделали в конце главы 1);

— делаем Storage потокобезопасным: один мьютекс на всё хранилище;

— INCR и SETNX выполняются внутри Storage под одним захватом — проверка и изменение не разделены;

— добавляем стресс-тесты под TSan;

— пишем первую версию нагрузочного клиента kvbench и впервые меряем сервер.

Что проверяем:

— 8 потоков по 5000 INCR дают ровно 40 000;

— 8 потоков одновременно делают SETNX одного ключа — ровно один выигрывает, 200 раундов подряд;

— 6 клиентов через настоящий TCP-сервер делают INCR — ни один не потерян;

— всё это под TSan и ASan, 10 прогонов подряд.

Что меряем: пропускную способность сервера в зависимости от числа клиентов.

Новое хранилище

storage.hpp C++

```cpp
// storage.hpp — хранилище shardkv, шаг 2: потокобезопасное.
//
// Инвариант класса: map_ читается и меняется ТОЛЬКО под mutex_.
// Каждая публичная операция атомарна целиком: INCR и SETNX не разваливаются
// на «прочитали — подумали — записали», которые мог бы перемешать другой поток.
//
// Цена: один мьютекс на всё хранилище. Все соединения стоят в одной очереди.
// Бенчмарк этого шага покажет, во что это обходится; глава 4 это исправит.
#pragma once

#include <cstdint>
#include <mutex>
#include <optional>
#include <string>
#include <string_view>
#include <unordered_map>

namespace shardkv {

class Storage {
public:
    // Возвращаем КОПИЮ значения: ссылка на элемент map, вынесенная из-под лока, — гонка.
```

storage.hpp (продолжение) C++

```cpp
    [[nodiscard]] std::optional<std::string> get(std::string_view
        key) const {
        std::lock_guard lock(mutex_);
        if (auto it = map_.find(std::string(key)); it != map_.end())
            return it->second;
        return std::nullopt;
    }

    void set(std::string_view key, std::string_view value) {
        std::string k(key), v(value);  // аллокации — ДО захвата
            лока, чтобы держать его меньше
        std::lock_guard lock(mutex_);
        map_.insert_or_assign(std::move(k), std::move(v));
    }

    bool del(std::string_view key) {
        const std::string k(key);
        std::lock_guard lock(mutex_);
        return map_.erase(k) > 0;
    }

    bool setnx(std::string_view key, std::string_view value) {
        std::string k(key), v(value);
        std::lock_guard lock(mutex_);
        return map_.try_emplace(std::move(k), std::move(v)).second;
    }

```

storage.hpp (продолжение) C++

```cpp
    std::optional<std::int64_t> incr(std::string_view key);

    [[nodiscard]] std::size_t size() const {
        std::lock_guard lock(mutex_);
        return map_.size();
    }

private:
    mutable std::mutex mutex_;  // mutable: блокируем и в
        const-методах
    std::unordered_map<std::string, std::string> map_;  //
        GUARDED_BY(mutex_)
};

}  // namespace shardkv
```

Листинг 2.15 — хранилище shardkv, шаг 2 (storage.hpp)

storage.cpp

```cpp
#include "core/storage.hpp"

#include <charconv>
#include <limits>

namespace shardkv {

std::optional<std::int64_t> Storage::incr(std::string_view key) {
    std::string k(key);
    // Чтение, проверка и запись — под ОДНИМ захватом. Если бы мы сделали
    // get() + set() двумя вызовами, два клиента прочитали бы 5 и оба записали 6.
    std::lock_guard lock(mutex_);

    auto [it, inserted] = map_.try_emplace(std::move(k), "0");
    const std::string& text = it->second;

    std::int64_t value = 0;
    const auto [end, ec] = std::from_chars(text.data(), text.data() + text.size(), value);
    if (ec != std::errc{} || end != text.data() + text.size()) return std::nullopt;
    if (value == std::numeric_limits<std::int64_t>::max()) return std::nullopt;

```

storage.cpp (продолжение)

```cpp
    ++value;
    it->second = std::to_string(value);
    return value;
}

}  // namespace shardkv
```

Листинг 2.16 — реализация (storage.cpp)

Обратите внимание на три приёма из этой главы:
— мьютекс mutable, потому что захватывается и в const-методах;

— строки ключа и значения создаются до захвата лока, чтобы аллокации не выполнялись в критической секции;

— get() возвращает копию, а не ссылку.

В server.cpp из шага 1 меняется только комментарий над вызовом execute: гонки больше нет.

Стресс-тесты

test_storage_stress.cpp C++

```cpp
// Стресс-тесты хранилища. Запускать под TSan: ctest --preset tsan
    --repeat until-fail:10
#include <gtest/gtest.h>

#include <atomic>
#include <string>
#include <thread>
#include <vector>

#include "core/storage.hpp"
#include "net/socket.hpp"
#include "server/server.hpp"

TEST(StorageStress, ConcurrentIncrLosesNothing) {
    constexpr int kThreads = 8;
    constexpr int kIncrements = 5'000;
    shardkv::Storage storage;
    {
        std::vector<std::jthread> threads;
        for (int t = 0; t < kThreads; ++t) {
            threads.emplace_back([&storage] {
                for (int i = 0; i < kIncrements; ++i)
                    ASSERT_TRUE(storage.incr("hits").has_value());
            });
        }
    }
    EXPECT_EQ(storage.get("hits"), std::to_string(kThreads *
        kIncrements));
```

test_storage_stress.cpp (продолжение) C++

```cpp
}

TEST(StorageStress, SetnxHasExactlyOneWinner) {
    constexpr int kThreads = 8;
    for (int round = 0; round < 200; ++round) {
        shardkv::Storage storage;
        std::atomic<int> winners{0};
        {
            std::vector<std::jthread> threads;
            for (int t = 0; t < kThreads; ++t) {
                threads.emplace_back([&, t] {
                    if (storage.setnx("lock", std::to_string(t)))
                        winners.fetch_add(1);
                });
            }
        }
        ASSERT_EQ(winners.load(), 1) << "раунд " << round;
    }
}

TEST(StorageStress, ServerWithManyClients) {
    constexpr int kClients = 6;
    constexpr int kRequests = 300;
    shardkv::Storage storage;
    shardkv::Server server(storage, 0);
    std::jthread acceptor([&server](std::stop_token st) {
        server.run(st); });
    {
```

test_storage_stress.cpp (продолжение) C++

```
        std::vector<std::jthread> clients;
        for (int c = 0; c < kClients; ++c) {
            clients.emplace_back([&server] {
                auto conn = shardkv::net::connect_tcp(server.port());
                shardkv::net::LineReader reader(conn);
                for (int i = 0; i < kRequests; ++i) {
                    ASSERT_TRUE(shardkv::net::send_all(conn, "INCR
                        hits\n"));
                    ASSERT_TRUE(reader.next().has_value());
                }
            });
        }
    }
    acceptor.request_stop();
    acceptor.join();
    EXPECT_EQ(storage.get("hits"), std::to_string(kClients *
        kRequests));
}
```

Листинг 2.17 — тесты (test_storage_stress.cpp)

В тесте SetnxHasExactlyOneWinner используется std::atomic<int> для подсчёта победителей. Пока считайте его «int, который можно безопасно увеличивать из многих потоков»; подробно атомики разберём в главе 5.

kvbench, версия 1

kvbench.cpp C++

```cpp
// kvbench.cpp — нагрузочный клиент shardkv. Версия 1: простая и честная.
//
// Использование: kvbench <порт> <клиентов> <запросов_на_клиента> [get|set|incr|mix]
// Каждый клиент — отдельный поток со своим соединением. Запрос-ответ без конвейера.
// Печатает суммарную пропускную способность и среднюю задержку.
//
// Ограничение версии 1: клиенты стартуют не одновременно (первые успевают поработать,
// пока создаются последние). На тысячах запросов это шум; в главе 6 исправим через std::barrier.
#include <algorithm>
#include <charconv>
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <print>
#include <string>
#include <string_view>
#include <thread>
#include <vector>

#include "net/socket.hpp"

namespace {

```

kvbench.cpp (продолжение) C++

```cpp
std::string make_request(std::string_view mode, int client, int i) {
    const std::string key = "key:" + std::to_string((client * 7919 +
        i) % 1000);
    if (mode == "get") return "GET " + key + "\n";
    if (mode == "set") return "SET " + key + " value\n";
    if (mode == "incr") return "INCR counter\n";  // один горячий
        ключ — худший случай
    return (i % 10 == 0) ? "SET " + key + " value\n" : "GET " + key
        + "\n";  // mix: 90% чтений
}

template <class T>
bool parse(std::string_view s, T& out) {
    return std::from_chars(s.data(), s.data() + s.size(), out).ec ==
        std::errc{};
}

}  // namespace

int main(int argc, char** argv) {
    std::uint16_t port = 0;
    int clients = 0;
    int requests = 0;
    if (argc < 4 || !parse(argv[1], port) || !parse(argv[2],
        clients) || !parse(argv[3], requests) || clients <= 0 ||
        requests <= 0) {
        std::println(stderr, "Использование: {} <порт> <клиентов>
            <запросов> [get|set|incr|mix]", argv[0]);
```

kvbench.cpp (продолжение) C++

```cpp
        return 2;
    }
    const std::string mode = argc > 4 ? argv[4] : "mix";

    // Каждый клиент пишет свою статистику в свою ячейку: общих данных нет, мьютекс не нужен.
    std::vector<std::chrono::nanoseconds> busy(static_cast<std::size_t>(clients));
    std::vector<int> failures(static_cast<std::size_t>(clients), 0);

    const auto start = std::chrono::steady_clock::now();
    {
        std::vector<std::jthread> threads;
        for (int c = 0; c < clients; ++c) {
            threads.emplace_back([&, c] {
                try {
                    auto conn = shardkv::net::connect_tcp(port);
                    shardkv::net::LineReader reader(conn);
                    const auto t0 = std::chrono::steady_clock::now();
                    for (int i = 0; i < requests; ++i) {
                        if (!shardkv::net::send_all(conn, make_request(mode, c, i)) || !reader.next()) {
                            ++failures[static_cast<std::size_t>(c)];
                            break;
                        }
                    }
```

kvbench.cpp (продолжение) C++

```
                    busy[static_cast<std::size_t>(c)] =
                        std::chrono::steady_clock::now() - t0;
                } catch (const std::exception&) {
                    ++failures[static_cast<std::size_t>(c)];
                }
            });
        }
    }  // join всех клиентов: после этого их ячейки можно читать
    const auto wall = std::chrono::steady_clock::now() - start;

    const double seconds = std::chrono::duration<double>(wall).count(
        );
    const double total = static_cast<double>(clients) * requests;
    std::chrono::nanoseconds busy_sum{0};
    for (auto b : busy) busy_sum += b;
    const double avg_latency_us = std::chrono::duration<double,
        std::micro>(busy_sum).count() / total;
    const int failed = std::ranges::count_if(failures, [](int f) {
        return f > 0; });

    std::println("mode={} clients={} requests={} -> {:.0f} ops/s,
        avg latency {:.1f} us, failed clients {}", mode,
                 clients, requests, total / seconds, avg_latency_us,
                     failed);
    return failed == 0 ? 0 : 1;
}
```

Листинг 2.18 — нагрузочный клиент shardkv (kvbench.cpp)

Каждый клиент — отдельный поток со своим соединением. Обратите внимание: клиенты пишут статистику каждый в свою ячейку вектора, а читаем мы её только после join всех потоков. Общих изменяемых данных нет, синхронизация не нужна. Это лучший вид синхронизации.

Первые замеры

Сборка release, сервер и kvbench на одной машине, режим mix (90% GET, 10% SET), 20 000 запросов на клиента:

```
терминал                                                    SHELL
$ cmake --preset release && cmake --build --preset release
$ ./build/release/shardkv 6380 &
```

Листинг 2.19 — сборка release и запуск сервера

Результаты на моей машине:

```
терминал                                                    SHELL
$ for c in 1 2 4 8 16; do ./build/release/kvbench 6380 $c 20000 mix; done
clients=1   ->  17 829 ops/s, avg latency  56.1 us
clients=2   ->  63 412 ops/s, avg latency  21.9 us
clients=4   -> 107 909 ops/s, avg latency  32.3 us
clients=8   -> 111 313 ops/s, avg latency  67.5 us
clients=16  ->  80 786 ops/s, avg latency 186.2 us
```

Листинг 2.20 — kvbench версии 1: пропускная способность и средняя задержка

Что видно:

— до 4 клиентов пропускная способность растёт;

— дальше начинается плато, а на 16 клиентах она падает;

— задержка растёт быстрее, чем пропускная способность: с 8 до 16 клиентов задержка выросла почти втрое, а ops/s упали.

Честная оговорка, без которой замеры ничего не стоят: эти цифры сняты в контейнере с двумя ядрами, где сервер и kvbench делят одни и те же два ядра. На такой машине плато наступает уже из-за нехватки CPU, и отделить его от конкуренции за мьютекс по этим цифрам нельзя. Прогоните бенчмарк на своей машине с 8+ ядрами: там картина «пропускная способность упёрлась, хотя ядра простаивают» будет видна отчётливо. Как доказать, что виноват именно лок, а не CPU, мы научимся в главе 13 (off-CPU профилирование). А в главе 4 уберём глобальный мьютекс и сравним цифры.

Сохраните свои результаты в bench/results/step02.txt. Вся книга построена на сравнении «было/стало», и без «было» сравнивать будет не с чем.

Полный код шага 2

Остальные файлы не изменились по сравнению с шагом 1, кроме CMakeLists.txt (добавлены kvbench и стресс-тесты):

CMakeLists.txt CMAKE

```cmake
cmake_minimum_required(VERSION 3.25)
project(shardkv LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Санитайзер выбирается пресетом: -DSHARDKV_SANITIZER=thread |
    address
set(SHARDKV_SANITIZER "" CACHE STRING "thread, address или пусто")

# Строгие предупреждения только для нашего кода, не для googletest
set(SHARDKV_WARNINGS -Wall -Wextra -Wpedantic -Werror)
if(SHARDKV_SANITIZER STREQUAL "thread")
    add_compile_options(-fsanitize=thread -g -O1)
    add_link_options(-fsanitize=thread)
elseif(SHARDKV_SANITIZER STREQUAL "address")
    add_compile_options(-fsanitize=address,undefined
        -fno-omit-frame-pointer -g -O1)
    add_link_options(-fsanitize=address,undefined)
endif()

find_package(Threads REQUIRED)

```

CMakeLists.txt (продолжение) CMAKE

```cmake
add_library(shardkv_lib
    src/core/storage.cpp
    src/core/protocol.cpp
    src/net/socket.cpp
    src/server/server.cpp)
target_include_directories(shardkv_lib PUBLIC src)
target_link_libraries(shardkv_lib PUBLIC Threads::Threads)
target_compile_options(shardkv_lib PRIVATE ${SHARDKV_WARNINGS})

add_executable(shardkv src/server/main.cpp)
target_link_libraries(shardkv PRIVATE shardkv_lib)
target_compile_options(shardkv PRIVATE ${SHARDKV_WARNINGS})

# --- Нагрузочный клиент ---
add_executable(kvbench bench/kvbench.cpp)
target_link_libraries(kvbench PRIVATE shardkv_lib)
target_compile_options(kvbench PRIVATE ${SHARDKV_WARNINGS})

# --- Тесты ---
include(FetchContent)
FetchContent_Declare(googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG v1.15.2)
FetchContent_MakeAvailable(googletest)

```

CMakeLists.txt (продолжение) CMAKE

```cmake
enable_testing()
add_executable(shardkv_tests tests/test_protocol.cpp
    tests/test_server.cpp tests/test_storage_stress.cpp)
target_link_libraries(shardkv_tests PRIVATE shardkv_lib
    GTest::gtest_main)
target_compile_options(shardkv_tests PRIVATE ${SHARDKV_WARNINGS})
include(GoogleTest)
gtest_discover_tests(shardkv_tests PROPERTIES TIMEOUT 30)
```

Листинг 2.21 — файл сборки CMake (CMakeLists.txt)

Задачки для разминки мозгов

Напоминаю правило самопроверки: решение засчитывается, только если тесты проходят под TSan и ASan десять раз подряд.

Задача 2.1 (лёгкая). Банк, который зависает

task2_1_bank.cpp

```cpp
// Задача 2.1. Банк, который зависает под нагрузкой. Найдите почему и почините.
#include <cstdint>
#include <mutex>
#include <print>
#include <thread>
#include <vector>

struct Account {
    std::uint64_t id;
    std::mutex m;
    std::int64_t balance;
};

bool transfer(Account& from, Account& to, std::int64_t amount) {
    std::lock_guard lock_from(from.m);
    std::lock_guard lock_to(to.m);
    if (from.balance < amount) return false;
    from.balance -= amount;
    to.balance += amount;
    return true;
}

int main() {
    Account alice{.id = 1, .m = {}, .balance = 1'000'000};
    Account bob{.id = 2, .m = {}, .balance = 1'000'000};
    {
        std::vector<std::jthread> tellers;
```

task2_1_bank.cpp (продолжение)

```cpp
        for (int i = 0; i < 4; ++i) {
            tellers.emplace_back([&] { for (int k = 0; k < 100'000; ++k) transfer(alice, bob, 1); });
            tellers.emplace_back([&] { for (int k = 0; k < 100'000; ++k) transfer(bob, alice, 1); });
        }
    }
    std::println("итого денег в банке: {}", alice.balance + bob.balance);  // должно быть 2000000
}
```

Листинг 2.22 — код к задаче 2.1 (task2_1_bank.cpp)

Эта программа зависает (у меня — каждый раз). Найдите дедлок и опишите конкретную последовательность событий, которая к нему приводит. Исправьте функцию transfer двумя способами:

1. через std::scoped_lock;
2. через захват мьютексов в порядке возрастания id счёта.

Объясните, почему оба способа корректны. Отдельно подумайте: что должно происходить при transfer(alice, alice, 10)?

Проверка: программа завершается за секунды, итоговая сумма денег в банке равна 2 000 000; TSan чист; перевод самому себе не приводит ни к дедлоку, ни к UB.

Задача 2.2 (средняя). ThreadSafeStack без гонок в интерфейсе

Напишите шаблон ThreadSafeStack<T> с интерфейсом:

фрагмент.cpp C++

```cpp
void push(T value);
std::optional<T> try_pop();
std::size_t size() const;  // только для статистики, не для принятия
    решений
```

Листинг 2.23 — интерфейс ThreadSafeStack<T>

Требования:

— никаких методов empty() + top() + pop(), которые нужно вызывать связкой;

— исключение в конструкторе копирования или перемещения T не должно оставить стек в некорректном состоянии и не должно потерять элемент;

— нельзя возвращать ссылки на внутренние данные.

Напишите стресс-тест: 4 потока кладут по 100 000 уникальных чисел, 4 потока одновременно забирают, пока не заберут все. Каждое число должно быть извлечено ровно один раз.

Проверка: стресс-тест проходит под TSan 10 раз подряд; вы можете объяснить, почему size() в многопоточном коде годится только для статистики.

Задача 2.3 (сложная). hierarchical_mutex

Напишите класс hierarchical_mutex: мьютекс с числовым уровнем. Правило: поток, который держит мьютекс уровня N, может захватить только мьютекс с уровнем строго меньше N. Попытка нарушить порядок бросает std::logic_error сразу, до захвата, а не приводит к дедлоку когда-нибудь потом.

Требования:

— методы lock(), unlock(), try_lock(), чтобы класс работал с std::lock_guard и std::unique_lock;

— текущий уровень потока хранится в thread_local переменной;

— после unlock() уровень потока восстанавливается до предыдущего (поток может держать несколько мьютексов по иерархии);

— подумайте, что делать, если мьютексы отпускают не в обратном порядке захвата.

Проверка: тесты на захват по правильной иерархии; на исключение при нарушении порядка; на корректное восстановление уровня после цепочки lock/unlock; тест с несколькими потоками под TSan.

Решения и разбор ваших неизбежных ошибок — в главе «Ответы к задачам» в конце книги.

Глава 3. Condition variables и потокобезопасная очередь

«Добавил sleep(100), и баг пропал». Баг не пропал. Он ушёл в отпуск и вернётся в пятницу.

Мьютекс отвечает на вопрос «как не мешать друг другу». Но потокам нужно ещё и **ждать** друг друга: потребитель ждёт, пока в очереди появится задача; фоновый поток ждёт ближайшего дедлайна; главный поток ждёт, пока загрузится конфиг. Делать это через while (!ready) sleep_for(…) — путь в никуда: либо реакция запаздывает на время сна, либо CPU сгорает на холостых проверках.

Правильный инструмент — condition variable (условная переменная). У неё самая коварная семантика из всех примитивов стандартной библиотеки: ошибиться в ней можно тремя разными способами, и все три проявляются раз в сто тысяч запусков. Сегодня мы разберём все три, напишем блокирующую очередь, которая не теряет элементы и умеет закрываться, и научим shardkv удалять ключи по времени жизни.

К концу главы вы:

— поймёте, почему condition variable всегда используется вместе с мьютексом и предикатом;

— научитесь распознавать spurious wakeup и lost wakeup;

— разберётесь, когда звать notify_one, когда notify_all и почему уведомлять лучше после освобождения лока;

— научитесь ждать с таймаутом и дедлайном и узнаете, почему дедлайн лучше;

— освоите condition_variable_any вместе с stop_token;

— напишете блокирующую очередь с закрытием и корректной остановкой;

— добавите в shardkv EXPIRE/TTL с фоновым потоком, который спит ровно до ближайшего дедлайна.

3.1. Как ждать правильно

Condition variable — это механизм «усни, пока тебя не разбудят». Сама по себе она ничего не хранит и ничего не знает о состоянии программы. Это ключевая мысль, из которой следует всё остальное: **состояние хранится в обычных переменных под мьютексом, а condition variable лишь будит тех, кто этого состояния ждёт.**

Минимальный правильный шаблон выглядит так:

ex01_event.cpp C++

```cpp
// ex01_event.cpp — «дождаться события» тремя способами: плохо,
    очень плохо и правильно.
#include <chrono>
#include <condition_variable>
#include <mutex>
#include <print>
#include <string>
#include <thread>

using namespace std::chrono_literals;

// ОЧЕНЬ ПЛОХО: активное ожидание на обычном bool — data race и UB
    (глава 2).
// ПЛОХО: опрос с sleep_for — либо задержка реакции, либо сожжённый
    CPU. Выбирайте.

// ПРАВИЛЬНО: condition_variable + мьютекс + предикат.
class Event {
public:
    void set() {
        {
            std::lock_guard lock(m_);
            ready_ = true;  // меняем состояние ПОД ЛОКОМ — даже
                если это просто bool
        }
        cv_.notify_all();  // будим ПОСЛЕ освобождения лока:
            проснувшимся не придётся ждать мьютекс
    }
```

ex01_event.cpp (продолжение) C++

```cpp

    void wait() {
        std::unique_lock lock(m_);                // wait требует
            unique_lock: он будет его отпускать
        cv_.wait(lock, [this] { return ready_; });  // предикат
            защищает от spurious и lost wakeup
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    bool ready_ = false;  // GUARDED_BY(m_)
};

int main() {
    Event config_loaded;
    std::string config;  // пишется до set(), читается после wait():
        синхронизировано через m_

    std::jthread loader([&] {
        std::this_thread::sleep_for(50ms);  // «читаем файл»
        config = "port=6380";
        config_loaded.set();
    });

    config_loaded.wait();  // спим, а не крутимся: CPU свободен
    std::println("конфиг получен: {}", config);
}
```

Листинг 3.1 — «дождаться события» тремя способами (ex01_event.cpp)

Три обязательных элемента:

1. **Состояние** (здесь ready_) — обычная переменная, которую читают и меняют только под мьютексом. Даже если это bool.
2. **Мьютекс** — один и тот же для изменения состояния и для ожидания.
3. **Предикат** в wait — функция, которая проверяет состояние.

Что происходит внутри cv.wait(lock, pred):

псевдокод C++

```cpp
while (!pred()) {
    // атомарно: отпустить мьютекс и уснуть
    // проснувшись: снова захватить мьютекс
}
```

Листинг 3.2 — что делает cv.wait(lock, pred)

Слово «атомарно» здесь критично: между освобождением мьютекса и засыпанием нет окна, в которое могло бы проскочить уведомление. Поэтому wait принимает именно unique_lock: ему нужно отпускать и снова захватывать мьютекс, а lock_guard так не умеет.

3.2. Три способа ошибиться

Ошибка 1: wait без предиката и spurious wakeup

У condition variable есть официально разрешённое право проснуться без всякого notify. Это называется spurious wakeup (ложное пробуждение). Причины — детали реализации ОС: так дешевле реализовать примитив. Стандарт явно разрешает такое поведение.

Если код выглядит так:

фрагмент.cpp C++

```
1  std::unique_lock lock(m);
2  if (!ready) cv.wait(lock);  // ПЛОХО: if вместо while
3  use(data);                  // а ready всё ещё false
```

Листинг 3.3 — if вместо while

то после ложного пробуждения он пойдёт дальше, считая, что событие произошло. Лечится просто: всегда используйте перегрузку wait с предикатом (или явный цикл while). Предикат перепроверит состояние после любого пробуждения.

Есть и вторая причина для цикла, совсем не экзотическая. Даже при честном пробуждении к моменту, когда поток снова захватил мьютекс, состояние могло измениться: другой потребитель успел забрать элемент из очереди. Проснулся — не значит, что условие выполняется.

Ошибка 2: lost wakeup — уведомление, которое никто не услышал

Condition variable не запоминает уведомления. Если notify случился, когда никто не ждал, он просто пропал. Поэтому ждущий поток обязан проверить состояние **до** засыпания, и проверка с засыпанием должны быть атомарны относительно изменения состояния. Это обеспечивает мьютекс — но только если и изменение состояния идёт под ним.

Вот как это ломается:

фрагмент.cpp C++

```
// Поток A (ждущий):                  // Поток B (уведомляющий):
std::unique_lock lock(m);
if (!ready)        // false
                                      ready = true;     // БЕЗ мьютекса
                                      cv.notify_one();  // никто ещё не
                                      ждёт — пропало
    cv.wait(lock); // спит вечно
```

Листинг 3.4 — потерянное пробуждение: два потока по времени

Поток B поменял флаг и уведомил в окне между проверкой и засыпанием потока A. Если бы B менял ready под тем же мьютексом, он не смог бы попасть в это окно: A держит мьютекс от проверки до момента, когда wait атомарно отпускает его и засыпает.

Самое неприятное: эта ошибка не лечится тем, что ready объявлен как std::atomic<bool>. Атомарность флага убирает data race, но не закрывает окно между проверкой и засыпанием. В задаче 3.1 вас ждёт именно такой код.

Правило: изменение состояния, которого ждут через condition variable, выполняется под тем же мьютексом, что и ожидание. Без исключений, даже для атомиков.

Ошибка 3: разные мьютексы или состояние вне мьютекса

Вариации предыдущей ошибки: ждут под одним мьютексом, а состояние меняют под другим; или предикат читает данные, которые защищены совсем другим локом. Результат тот же — окно для потерянного уведомления, плюс data race в предикате. Одна condition variable — один мьютекс — одно защищаемое состояние.

3.3. notify_one, notify_all и где их вызывать

notify_one будит одного ждущего (какого — не определено). Используйте, когда изменение состояния может обработать ровно один поток: в очередь положили один элемент.

notify_all будит всех. Используйте, когда изменение важно для всех: очередь закрыли, конфиг загружен, наступил момент «всем стоп». Не используйте его «на всякий случай» в горячем пути: при 32 ждущих потоках один элемент в очереди разбудит всех 32, один заберёт элемент, 31 проверит предикат и снова уснёт. Это называется thundering herd (паническое стадо), и на нагруженных системах оно стоит заметных процентов CPU.

Ошибка в обратную сторону опаснее: notify_one там, где нужен notify_all. Если очередь закрыли и разбудили одного из восьми ждущих потребителей, остальные семь будут ждать вечно.

Уведомлять под локом или после? Оба варианта корректны, если состояние менялось под локом. Но если вызвать notify под локом, проснувшийся поток сразу упрётся в мьютекс, который уведомляющий ещё держит, и уснёт снова, теперь уже на мьютексе. Поэтому обычно делают так:

фрагмент.cpp — C++

```cpp
{
    std::lock_guard lock(m);
    ready = true;      // изменение — под локом
}
cv.notify_one();       // уведомление — после
```

Листинг 3.5 — изменить состояние под локом, уведомить после

Есть одно исключение. Если сразу после уведомления condition variable может быть уничтожена (например, ждущий поток проснётся, увидит флаг и удалит объект, содержащий cv), то notify после освобождения лока может обратиться к уже уничтоженной cv. В таких сценариях уведомляют под локом. Это редкость, но это ровно тот тип бага, который находят в продакшене.

3.4. Ожидание с таймаутом: wait_for и wait_until

Ждать вечно можно не всегда. Для ожидания с ограничением есть две функции:
— **wait_for(lock, duration, pred)** — ждать не дольше заданной длительности;
— **wait_until(lock, time_point, pred)** — ждать до заданного момента.
Обе с предикатом возвращают bool: true, если предикат стал истинным, false, если время вышло, а предикат так и остался ложным.

ex03_deadline.cpp C++

```cpp
// ex03_deadline.cpp — ожидание с дедлайном: «ответ за 100 мс или ошибка».
#include <chrono>
#include <condition_variable>
#include <mutex>
#include <optional>
#include <print>
#include <string>
#include <thread>

using namespace std::chrono_literals;

class ResultSlot {
public:
    void put(int value) {
        {
            std::lock_guard lock(m_);
            value_ = value;
        }
        cv_.notify_all();
    }

    // Дедлайн, а не длительность: если нас разбудят ложно, мы не начнём отсчёт заново.
    std::optional<int> wait_until(std::chrono::steady_clock::time_point deadline) {
        std::unique_lock lock(m_);
```

ex03_deadline.cpp (продолжение) C++

```cpp
        if (!cv_.wait_until(lock, deadline, [this] { return
            value_.has_value(); })) {
            return std::nullopt;  // предикат так и не стал true до
                дедлайна
        }
        return value_;
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    std::optional<int> value_;  // GUARDED_BY(m_)
};

int main() {
    ResultSlot fast, slow;
    std::jthread a([&] { std::this_thread::sleep_for(10ms);
        fast.put(42); });
    std::jthread b([&] { std::this_thread::sleep_for(300ms);
        slow.put(7); });

    const auto deadline = std::chrono::steady_clock::now() + 100ms;
        // ОДИН дедлайн на оба ожидания
    const auto r1 = fast.wait_until(deadline);
    const auto r2 = slow.wait_until(deadline);
    std::println("fast: {}", r1 ? std::to_string(*r1) : "таймаут");
    std::println("slow: {}", r2 ? std::to_string(*r2) : "таймаут");
}
```

Листинг 3.6 — ожидание с дедлайном (ex03_deadline.cpp)

Почему дедлайн лучше длительности. Представьте функцию, которой дано 100 мс на всё, а она ждёт несколько событий подряд. Если каждое ожидание делать через wait_for(100ms), общая длительность может стать 300 мс. Правильно вычислить дедлайн один раз в начале и ждать до него во всех вызовах. Так и сделано в примере: один дедлайн на оба ожидания.

Та же логика внутри одного ожидания: если вы реализуете цикл ожидания сами и после ложного пробуждения снова зовёте wait_for с той же длительностью, отсчёт начинается заново, и под постоянными ложными пробуждениями поток может не выйти никогда. wait_until с фиксированным моментом от этого защищён. (Перегрузки с предикатом из стандартной библиотеки сделаны правильно, но если пишете цикл руками — помните.)

И, как всегда для таймаутов, используйте steady_clock (глава 2, раздел 2.8).

3.5. condition_variable_any и stop_token

В главе 1 мы увидели проблему: поток, заснувший в блокирующем вызове, не узнает о запросе остановки, пока его не разбудят. Для condition variable в C++20 есть готовое решение. std::condition_variable_any — более общая версия condition_variable, которая работает с любым типом лока и, главное, умеет ждать с stop_token:

фрагмент.cpp C++

```cpp
cv.wait(lock, stop_token, pred);
cv.wait_until(lock, stop_token, deadline, pred);
```

Листинг 3.7 — ожидание с stop_token

Такое ожидание просыпается, когда предикат стал истинным **или** запрошена остановка. Возвращаемое значение — результат предиката: если вернулось false, значит, вас разбудила остановка (или дедлайн), а не событие.

Внутри это сделано через stop_callback, который будит condition variable при request_stop() — ровно тот приём, который мы писали руками для сокетов в главе 1. Плата за удобство — condition_variable_any чуть тяжелее обычной condition_variable (у неё свой внутренний мьютекс). В большинстве случаев это несущественно, но в очень горячих путях имейте в виду.

3.6. Блокирующая очередь

Очередь «производитель — потребитель» — самый распространённый многопоточный паттерн: одни потоки создают задачи, другие их выполняют. Соберём её из всего, что знаем:

ex02_blocking_queue.cpp C++

```cpp
// ex02_blocking_queue.cpp — производитель-потребитель на неограниченной очереди с закрытием.
#include <condition_variable>
#include <deque>
#include <mutex>
#include <optional>
#include <print>
#include <stop_token>
#include <thread>
#include <utility>
#include <vector>

template <class T>
class BlockingQueue {
public:
    // false — очередь закрыта, элемент не принят.
    bool push(T value) {
        {
            std::lock_guard lock(m_);
            if (closed_) return false;
            items_.push_back(std::move(value));
        }
        cv_.notify_one();  // элемент один — будить всех незачем
        return true;
    }

```

ex02_blocking_queue.cpp (продолжение) C++

```cpp
    // Ждёт элемент. nullopt — очередь закрыта и пуста ИЛИ запрошена остановка через token.
    std::optional<T> pop(std::stop_token token) {
        std::unique_lock lock(m_);
        // condition_variable_any умеет просыпаться по stop_token: никаких «будильников» вручную.
        const bool has_item = cv_.wait(lock, token, [this] { return !items_.empty() || closed_; });
        if (!has_item || items_.empty()) return std::nullopt;
        T value = std::move(items_.front());
        items_.pop_front();
        return value;
    }

    // После close() новые элементы не принимаются, а оставшиеся можно дочитать.
    void close() {
        {
            std::lock_guard lock(m_);
            closed_ = true;
        }
        cv_.notify_all();  // а вот тут будим ВСЕХ: каждый ждущий должен узнать о закрытии
    }

```

ex02_blocking_queue.cpp (продолжение) C++

```cpp
private:
    std::mutex m_;
    std::condition_variable_any cv_;
    std::deque<T> items_;  // GUARDED_BY(m_)
    bool closed_ = false;  // GUARDED_BY(m_)
};

int main() {
    BlockingQueue<int> queue;
    std::vector<long> sums(3, 0);  // у каждого потребителя своя ячейка — никаких гонок

    {
        std::vector<std::jthread> consumers;
        for (std::size_t c = 0; c < sums.size(); ++c) {
            consumers.emplace_back([&queue, &sum = sums[c]](std::stop_token st) {
                while (auto item = queue.pop(st)) sum += *item;
            });
        }

        for (int i = 1; i <= 1000; ++i) queue.push(i);
        queue.close();  // потребители дочитают остаток и выйдут сами
    }   // join потребителей

    long total = 0;
    for (long s : sums) total += s;
    std::println("сумма: {} (ожидали {})", total, 1000 * 1001 / 2);
}
```

Листинг 3.8 — производитель-потребитель на неограниченной очереди с закрытием (ex02_blocking_queue.cpp)

На что обратить внимание:

— **pop возвращает std::optional<T>.** Проверка «не пусто ли» и извлечение — одна операция под одним локом (урок главы 2).

— **push уведомляет одного, close — всех.** Новый элемент может забрать один потребитель; о закрытии должны узнать все.

— **После close элементы можно дочитать.** Предикат просыпается, если есть элементы ИЛИ очередь закрыта, а pop возвращает nullopt, только когда очередь закрыта **и** пуста. Ни одна принятая задача не теряется.

— **pop принимает stop_token.** Потребителя можно остановить, даже если очередь не закрыта и пуста.

— **Каждый потребитель пишет результат в свою ячейку.** Общий счётчик потребовал бы ещё одного мьютекса; отдельные ячейки, которые читаются после join, — нет.

У этой очереди один серьёзный недостаток: она **неограниченная**. Если производитель быстрее потребителей, очередь растёт, пока не закончится память. В продакшене это классический сценарий падения под нагрузкой: сервис не тормозит, а копит задачи в памяти, пока его не убьёт OOM killer. Ограниченную очередь, которая заставляет производителя ждать, вы напишете в задаче 3.2, а о стратегиях работы с переполнением (backpressure) поговорим в главе 12.

3.7. Graceful shutdown: как остановить систему потоков

Корректная остановка — одна из самых недооценённых задач. Сервис, который при деплое теряет запросы, зависает на остановке (и его убивают по таймауту через SIGKILL) или падает в деструкторах, — это сервис, который боятся деплоить. А сервис, который боятся деплоить, деплоят в пятницу вечером, когда «ну наконец-то всё готово».

Общая схема для системы «производители → очередь → потребители»:

1. **Перестать принимать новую работу.** Производители останавливаются (или закрываются входные соединения).

2. **Закрыть очередь.** close() будит всех потребителей.

3. **Дочитать остаток.** Потребители обрабатывают то, что уже в очереди, и выходят, когда pop вернул nullopt.

4. **Дождаться потребителей.** join (в нашем случае — деструкторы jthread).

5. **Только после этого разрушать общие ресурсы**, которыми пользовались потоки.

Если нужно остановиться быстро и потерять остаток допустимо, вместо шага 3 используется stop_token: потребители выходят сразу, не дочитывая. Какая стратегия правильная, решает бизнес: потерять задачу «отправить письмо» неприятно, потерять задачу «списать деньги» недопустимо.

Порядок шагов важен. Закрыть очередь до остановки производителей — значит, часть задач отвергнется (push вернёт false), и производители должны уметь это обработать. Разрушить ресурсы до join потребителей — use-after-free, о котором мы говорили в главе 1.

3.8. Грабли с ревью

— **wait без предиката** (или через if, а не while).
— **Изменение состояния без мьютекса**, «потому что оно атомарное».
— **notify_one при закрытии или остановке**, когда ждущих может быть несколько.
— **notify_all на каждый элемент** в горячем пути.
— **Предикат, который читает данные под другим мьютексом** или вообще без него.
— **wait_for в цикле** с той же длительностью вместо дедлайна.
— **Ожидание, которое нельзя прервать** при остановке сервиса.
— **Неограниченная очередь** между компонентами с разной скоростью.
— **sleep_for как синхронизация**: «подождём, пока поток успеет».

3.9. Что спросят на собеседовании

Зачем condition variable нужен мьютекс? (middle) Чтобы проверка состояния и засыпание были атомарны относительно изменения состояния. Без этого возможен lost wakeup. Мьютекс защищает состояние, cv только будит.

Что такое spurious wakeup и как с ним бороться? (middle) Пробуждение без notify, разрешённое стандартом. Бороться — всегда проверять предикат в цикле (перегрузка wait с предикатом).

Что такое lost wakeup? (middle/senior) Уведомление, отправленное, когда ещё никто не ждал. Возникает, если состояние меняется без того мьютекса, под которым ждут. Хороший ответ: «атомарный флаг не спасает» — и объяснение, почему.

notify_one или notify_all? (middle) notify_one, когда событие может обработать один поток; notify_all, когда событие важно всем (закрытие, остановка). Бонус: thundering herd.

Уведомлять под локом или после? (senior) Оба корректны; после — эффективнее (проснувшийся не упрётся в мьютекс); под локом — когда cv может быть уничтожена сразу после пробуждения ждущего.

Почему wait_until лучше wait_for? (senior) Дедлайн не сдвигается при повторных ожиданиях и ложных пробуждениях. Общий бюджет времени на операцию соблюдается.

Как корректно остановить пул потребителей? (senior) Перестать принимать работу, закрыть очередь, разбудить всех, дочитать остаток (или выйти по stop_token), дождаться потоков, потом освобождать ресурсы. Сильный ответ обсуждает, какие задачи допустимо потерять.

3.10. Сквозной проект shardkv, шаг 3: TTL и поток, который спит ровно сколько нужно

Что делаем:

— команды EXPIRE и PEXPIRE (время жизни в секундах и миллисекундах), TTL и PTTL (сколько осталось) с семантикой Redis: −2 — ключа нет, −1 — ключ без времени жизни;

— SET снимает время жизни, INCR сохраняет;

— ленивое истечение: любая операция с ключом сначала проверяет, не истёк ли он;

— активное истечение: фоновый поток удаляет истёкшие ключи, даже если к ним никто не обращается. Без этого ключи с TTL, которые больше никто не читает, копились бы в памяти вечно.

Изначально в плане книги этот шаг включал и ограниченную очередь между приёмом соединений и обработчиками. Я перенёс её в главу 7, где появится пул потоков: сейчас она потребовала бы переделать модель «поток на соединение», а заодно выдала бы решение задачи 3.2.

Как устроен фоновый поток

Наивное решение — проверять все ключи раз в секунду. Оно плохое по двум причинам: при миллионе ключей каждая проверка дорогая, а точность истечения — до секунды. Правильное решение:

— дедлайны хранятся в мин-куче (std::priority_queue), на вершине — ближайший;

— поток спит через wait_until ровно до ближайшего дедлайна;

— если появился дедлайн раньше текущего ближайшего, expire() будит поток через notify, и тот пересчитывает время сна;

— если дедлайн стал не ближайшим, будить поток не нужно: он и так проснётся вовремя;

— остановка через stop_token работает, даже если поток спит до дедлайна через час.

Тонкость: запись в куче может устареть. Ключу назначили TTL, а потом его перезаписали через SET (TTL снят) или назначили новый TTL. Удалять запись из середины priority_queue нельзя, поэтому мы оставляем её и при извлечении сверяем: совпадает ли дедлайн в куче с текущим expires_at ключа. Не совпадает — запись просто выбрасывается.

Код хранилища

storage.hpp C++

```cpp
// storage.hpp — хранилище shardkv, шаг 3: потокобезопасное, с временем жизни ключей (TTL).
//
// Истечение ключей работает двумя способами одновременно:
//  1. Лениво: любая операция с ключом сначала проверяет, не истёк ли он.
//  2. Активно: фоновый поток run_expirer() спит до ближайшего дедлайна и удаляет
//     истёкшие ключи, даже если к ним больше никто не обращается (иначе память утечёт).
//
// Инвариант: map_, deadlines_ читаются и меняются только под mutex_.
#pragma once

#include <chrono>
#include <condition_variable>
#include <cstdint>
#include <mutex>
#include <optional>
#include <queue>
#include <stop_token>
#include <string>
#include <string_view>
#include <unordered_map>
#include <vector>

namespace shardkv {

```

storage.hpp (продолжение) C++

```cpp
class Storage {
public:
    using Clock = std::chrono::steady_clock;  // для TTL только монотонные часы

    [[nodiscard]] std::optional<std::string> get(std::string_view key);
    void set(std::string_view key, std::string_view value);  // SET снимает TTL, как в Redis
    bool del(std::string_view key);
    bool setnx(std::string_view key, std::string_view value);
    std::optional<std::int64_t> incr(std::string_view key);

    // Назначает время жизни существующему ключу. false — ключа нет.
    bool expire(std::string_view key, std::chrono::milliseconds ttl);

    // Оставшееся время жизни: nullopt — ключа нет; milliseconds::max() — ключ без TTL.
    [[nodiscard]] std::optional<std::chrono::milliseconds>
        ttl(std::string_view key);

    [[nodiscard]] std::size_t size() const;

    // Тело фонового потока активного истечения. Возвращается, когда запрошена остановка.
    void run_expirer(std::stop_token token);

```

storage.hpp (продолжение) C++

```
46  private:
47      struct Entry {
48          std::string value;
49          std::optional<Clock::time_point> expires_at;
50      };
51      struct Deadline {
52          Clock::time_point when;
53          std::string key;
54          bool operator>(const Deadline& other) const { return when >
    ↪             other.when; }
55      };
56
57      // *_locked: вызывающий уже держит mutex_.
58      using Map = std::unordered_map<std::string, Entry>;
59      Map::iterator find_alive_locked(const std::string& key,
    ↪         Clock::time_point now);
60      std::size_t purge_expired_locked(Clock::time_point now);
61
62      mutable std::mutex mutex_;
63      std::condition_variable_any expirer_cv_;  // будит фоновый
    ↪         поток, если появился более ранний дедлайн
64      Map map_;                                 // GUARDED_BY(mutex_)
65      // Мин-куча дедлайнов. Может содержать устаревшие записи (ключ
    ↪         удалён или TTL изменён):
66      // при удалении сверяемся с Entry::expires_at, лишнее просто
    ↪         выбрасываем.
67      std::priority_queue<Deadline, std::vector<Deadline>,
    ↪         std::greater<>> deadlines_;  // GUARDED_BY(mutex_)
```

storage.hpp (продолжение) C++

```
68  };
69
70  }  // namespace shardkv
```

Листинг 3.9 — хранилище shardkv, шаг 3 (storage.hpp)

storage.cpp C++

```cpp
#include "core/storage.hpp"

#include <charconv>
#include <limits>

namespace shardkv {

Storage::Map::iterator Storage::find_alive_locked(const std::string&
    key, Clock::time_point now) {
    auto it = map_.find(key);
    if (it != map_.end() && it->second.expires_at &&
        *it->second.expires_at <= now) {
        map_.erase(it);  // ленивое истечение: ключ умер, просто
            никто ещё не заметил
        return map_.end();
    }
    return it;
}

std::optional<std::string> Storage::get(std::string_view key) {
    const std::string k(key);
    const auto now = Clock::now();
    std::lock_guard lock(mutex_);
    if (auto it = find_alive_locked(k, now); it != map_.end())
        return it->second.value;
    return std::nullopt;
}

```

storage.cpp (продолжение) C++

```cpp
void Storage::set(std::string_view key, std::string_view value) {
    std::string k(key), v(value);
    std::lock_guard lock(mutex_);
    // Старый дедлайн в куче останется, но станет устаревшим:
        expires_at больше не совпадёт.
    map_.insert_or_assign(std::move(k), Entry{std::move(v),
        std::nullopt});
}

bool Storage::del(std::string_view key) {
    const std::string k(key);
    const auto now = Clock::now();
    std::lock_guard lock(mutex_);
    auto it = find_alive_locked(k, now);
    if (it == map_.end()) return false;
    map_.erase(it);
    return true;
}

bool Storage::setnx(std::string_view key, std::string_view value) {
    std::string k(key), v(value);
    const auto now = Clock::now();
    std::lock_guard lock(mutex_);
    if (find_alive_locked(k, now) != map_.end()) return false;
    map_.emplace(std::move(k), Entry{std::move(v), std::nullopt});
    return true;
}

```

storage.cpp (продолжение) C++

```cpp
std::optional<std::int64_t> Storage::incr(std::string_view key) {
    std::string k(key);
    const auto now = Clock::now();
    std::lock_guard lock(mutex_);

    auto it = find_alive_locked(k, now);
    if (it == map_.end()) it = map_.emplace(std::move(k), Entry{"0",
        std::nullopt}).first;
    std::string& text = it->second.value;  // INCR сохраняет TTL
        ключа, как в Redis

    std::int64_t value = 0;
    const auto [end, ec] = std::from_chars(text.data(), text.data()
        + text.size(), value);
    if (ec != std::errc{} || end != text.data() + text.size())
        return std::nullopt;
    if (value == std::numeric_limits<std::int64_t>::max()) return
        std::nullopt;

    ++value;
    text = std::to_string(value);
    return value;
}

```

storage.cpp (продолжение) C++

```cpp
bool Storage::expire(std::string_view key, std::chrono::milliseconds ttl) {
    const std::string k(key);
    const auto now = Clock::now();
    const auto when = now + ttl;
    bool earliest = false;
    {
        std::lock_guard lock(mutex_);
        auto it = find_alive_locked(k, now);
        if (it == map_.end()) return false;
        it->second.expires_at = when;
        earliest = deadlines_.empty() || when < deadlines_.top().when;
        deadlines_.push(Deadline{when, k});
    }
    // Будим фоновый поток, только если новый дедлайн раньше всех известных:
    // иначе он и так проснётся вовремя. Лишние пробуждения — лишняя конкуренция за мьютекс.
    if (earliest) expirer_cv_.notify_one();
    return true;
}

std::optional<std::chrono::milliseconds> Storage::ttl(std::string_view key) {
    const std::string k(key);
    const auto now = Clock::now();
    std::lock_guard lock(mutex_);
```

storage.cpp (продолжение) C++

```cpp
    auto it = find_alive_locked(k, now);
    if (it == map_.end()) return std::nullopt;
    if (!it->second.expires_at) return std::chrono::milliseconds::max
        ();
    return std::chrono::ceil<std::chrono::milliseconds>(*it->second.e
        xpires_at - now);
}

std::size_t Storage::size() const {
    std::lock_guard lock(mutex_);
    return map_.size();
}

std::size_t Storage::purge_expired_locked(Clock::time_point now) {
    std::size_t removed = 0;
    while (!deadlines_.empty() && deadlines_.top().when <= now) {
        const Deadline& d = deadlines_.top();
        auto it = map_.find(d.key);
        // Удаляем, только если запись в куче актуальна: TTL ключа
            не меняли и ключ не пересоздавали.
        if (it != map_.end() && it->second.expires_at == d.when) {
            map_.erase(it);
            ++removed;
        }
        deadlines_.pop();
    }
    return removed;
}
```

storage.cpp (продолжение) C++

```cpp

void Storage::run_expirer(std::stop_token token) {
    std::unique_lock lock(mutex_);
    while (!token.stop_requested()) {
        purge_expired_locked(Clock::now());

        if (deadlines_.empty()) {
            // Дедлайнов нет: спим, пока не появится первый или не
                попросят остановиться.
            expirer_cv_.wait(lock, token, [this] { return
                !deadlines_.empty(); });
        } else {
            // Спим до ближайшего дедлайна. Проснёмся раньше, если
                появится дедлайн ещё раньше
            // (предикат) или запросят остановку (token). Ложные
                пробуждения безвредны:
            // цикл просто пересчитает всё заново.
            const auto next = deadlines_.top().when;
            expirer_cv_.wait_until(lock, token, next, [this, next] {
                return !deadlines_.empty() && deadlines_.top().when
                    < next;
            });
        }
    }
}

}  // namespace shardkv
```

Листинг 3.10 — реализация (storage.cpp)

Что стоит разобрать подробно:

— **run_expirer держит мьютекс почти всё время, кроме сна.** wait_until отпускает его на время сна, поэтому клиенты не блокируются. Но очистка (purge_expired_locked) идёт под мьютексом, и если одновременно истечёт миллион ключей, все клиенты подождут. В Redis эту проблему решают, ограничивая работу за один проход; для нашего масштаба это преждевременная оптимизация, но держите её в голове.

— **expire будит поток только при новом ближайшем дедлайне** и делает notify после освобождения лока.

— **Предикат в wait_until** — «появился дедлайн раньше того, до которого я сплю». Если поток проснулся ложно или по таймауту, цикл просто пересчитает всё заново: purge, затем новое ожидание.

— **get теперь не const.** Ленивое истечение удаляет ключ, то есть get может менять хранилище. Делать map_ mutable ради сохранения const было бы обманом: const-метод, который меняет видимое состояние, сбивает с толку и тех, кто читает код, и тех, кто рассуждает о потокобезопасности.

Протокол и запуск

В протокол добавлены команды EXPIRE, PEXPIRE, TTL, PTTL (полный код — в архиве главы). В main.cpp появился фоновый поток:

main.cpp C++

```cpp
shardkv::Storage storage;
// Объявлен ПОСЛЕ storage: разрушится раньше и успеет остановиться,
    пока storage жив.
std::jthread expirer([&storage](std::stop_token st) {
    storage.run_expirer(st); });
shardkv::Server server(storage, port);
```

Листинг 3.11 — порядок объявления storage и expirer (main.cpp)

Порядок объявления — опять тот же урок, что в главе 1: поток должен разрушиться раньше данных, которыми пользуется.

Тесты

test_ttl.cpp C++

```cpp
// Тесты TTL. Используют реальное время с большим запасом (десятки миллисекунд TTL
// против сотен миллисекунд ожидания), чтобы не быть flaky на нагруженной CI-машине.
#include <gtest/gtest.h>

#include <chrono>
#include <thread>

#include "core/protocol.hpp"
#include "core/storage.hpp"

using namespace std::chrono_literals;
using shardkv::execute;
using shardkv::Storage;

namespace {

// Ждём условия с дедлайном вместо одного sleep: тест быстрый, когда всё хорошо,
// и не падает зря, когда машина медленная.
template <class Pred>
bool eventually(Pred pred, std::chrono::milliseconds limit = 2s) {
    const auto deadline = std::chrono::steady_clock::now() + limit;
    while (std::chrono::steady_clock::now() < deadline) {
        if (pred()) return true;
        std::this_thread::sleep_for(5ms);
    }
```

test_ttl.cpp (продолжение) C++

```cpp
    return pred();
}

}  // namespace

TEST(Ttl, ProtocolSemantics) {
    Storage s;
    EXPECT_EQ(execute(s, "TTL nokey"), ":-2");
    EXPECT_EQ(execute(s, "SET k v"), "+OK");
    EXPECT_EQ(execute(s, "TTL k"), ":-1");
    EXPECT_EQ(execute(s, "EXPIRE k 100"), ":1");
    EXPECT_EQ(execute(s, "TTL k"), ":100");
    EXPECT_EQ(execute(s, "SET k v2"), "+OK");  // SET снимает TTL
    EXPECT_EQ(execute(s, "TTL k"), ":-1");
    EXPECT_EQ(execute(s, "EXPIRE nokey 10"), ":0");
    EXPECT_EQ(execute(s, "EXPIRE k -1"), "-ERR invalid expire time");
}

TEST(Ttl, LazyExpiryWithoutBackgroundThread) {
    Storage s;
    s.set("k", "v");
    ASSERT_TRUE(s.expire("k", 20ms));
    EXPECT_TRUE(eventually([&] { return !s.get("k").has_value(); }));
    EXPECT_EQ(s.size(), 0u);  // ленивое удаление при обращении
}

```

test_ttl.cpp (продолжение) C++

```cpp
TEST(Ttl, BackgroundExpirerRemovesUntouchedKeys) {
    Storage s;
    std::jthread expirer([&s](std::stop_token st) {
        s.run_expirer(st); });
    for (int i = 0; i < 100; ++i) {
        const auto key = "k" + std::to_string(i);
        s.set(key, "v");
        ASSERT_TRUE(s.expire(key, std::chrono::milliseconds(10 +
            i)));
    }
    s.set("forever", "v");
    // К ключам больше никто не обращается: удалить их может только
        фоновый поток.
    EXPECT_TRUE(eventually([&] { return s.size() == 1; }));
}

TEST(Ttl, EarlierDeadlineWakesExpirer) {
    Storage s;
    std::jthread expirer([&s](std::stop_token st) {
        s.run_expirer(st); });
    s.set("late", "v");
    ASSERT_TRUE(s.expire("late", 1h));  // поток заснёт на час...
    std::this_thread::sleep_for(20ms);
    s.set("soon", "v");
    ASSERT_TRUE(s.expire("soon", 10ms));  // ...и должен проснуться
        ради этого ключа
    EXPECT_TRUE(eventually([&] { return s.size() == 1; }));
}
```

test_ttl.cpp (продолжение) C++

```cpp

TEST(Ttl, StaleDeadlineDoesNotDeleteRenewedKey) {
    Storage s;
    std::jthread expirer([&s](std::stop_token st) {
        s.run_expirer(st); });
    s.set("k", "v");
    ASSERT_TRUE(s.expire("k", 20ms));
    s.set("k", "v2");  // TTL снят; старый дедлайн в куче остался
    std::this_thread::sleep_for(100ms);
    EXPECT_EQ(s.get("k"), "v2");
}

TEST(Ttl, ExpirerStopsPromptly) {
    Storage s;
    s.set("k", "v");
    ASSERT_TRUE(s.expire("k", 1h));
    const auto start = std::chrono::steady_clock::now();
    {
        std::jthread expirer([&s](std::stop_token st) {
            s.run_expirer(st); });
        std::this_thread::sleep_for(20ms);
    }  // ~jthread: request_stop + join. Поток спит до дедлайна
        через час — stop_token обязан его разбудить.
    EXPECT_LT(std::chrono::steady_clock::now() - start, 1s);
}
```

Листинг 3.12 — тесты (test_ttl.cpp)

Обратите внимание на функцию eventually. Тесты, завязанные на время, — главный источник flaky-тестов. Плохой вариант: sleep_for(100ms), потом проверка. На медленной CI-машине 100 мс не хватит, и тест упадёт. Хороший вариант: проверять условие в цикле до щедрого дедлайна. Когда всё работает, тест заканчивается за миллисекунды; когда машина медленная, тест подождёт, а не упадёт зря.

Тест EarlierDeadlineWakesExpirer проверяет главную многопоточную логику: поток заснул на час, а потом появился ключ, который истекает через 10 мс. Если expire не разбудит поток, тест не дождётся удаления. А ExpirerStopsPromptly проверяет, что stop_token будит поток, спящий до далёкого дедлайна.

Все тесты проходят под TSan и ASan 10 раз подряд.

Замеры

Добавили ли TTL накладные расходы на обычные GET и SET? Сравним шаг 2 и шаг 3 тем же kvbench (4 клиента, режим mix, три прогона подряд):

	прогон 1	прогон 2	прогон 3
шаг 2	105 762	107 375	134 780
шаг 3	141 367	109 063	102 325

Таблица 3.1 — накладные расходы TTL: шаг 2 и шаг 3, ops/s

Вывод: разница между шагами меньше, чем разброс между прогонами одной и той же версии (от 102 до 141 тысячи). Значит, по этим данным мы **не можем** сказать, что TTL что-то замедлил. Это нормальный и важный результат. Типичная ошибка — запустить бенчмарк по разу, увидеть 141 против 106 и написать в отчёте «оптимизация дала +33%». Без нескольких прогонов и оценки разброса любые цифры бенчмарка — гадание на кофейной гуще.

Задачки для разминки мозгов

Задача 3.1 (лёгкая). Потерянное пробуждение

task3_1_lost_wakeup.cpp C++

```cpp
// Задача 3.1. Этот код иногда зависает навсегда. Не часто. Но навсегда.
#include <atomic>
#include <condition_variable>
#include <mutex>
#include <print>
#include <thread>

std::mutex m;
std::condition_variable cv;
std::atomic<bool> ready{false};  // «атомик же, значит потокобезопасно»

void waiter() {
    std::unique_lock lock(m);
    if (!ready) {
        cv.wait(lock);
    }
}

void notifier() {
    ready = true;
    cv.notify_one();
}

```

task3_1_lost_wakeup.cpp (продолжение) C++

```cpp
int main() {
    for (int i = 0; i < 100'000; ++i) {
        ready = false;
        std::jthread w(waiter);
        std::jthread n(notifier);
    }
    std::println("100000 итераций без зависаний. Или нет?");
}
```

Листинг 3.13 — код к задаче 3.1 (task3_1_lost_wakeup.cpp)

Эта программа зависает. У меня — каждый запуск, но на вашей машине может понадобиться несколько. Флаг атомарный, notify вызывается — почему поток спит вечно?

Нарисуйте трассу событий двух потоков (как в разделе 3.2), которая приводит к зависанию. Найдите все проблемы в коде (их больше одной) и исправьте.

Проверка: исправленная программа проходит 100 000 итераций без зависаний, 10 запусков подряд, под TSan; трасса показывает конкретное окно, в которое попадает notify.

Задача 3.2 (средняя). BoundedQueue

Напишите шаблон ограниченной блокирующей очереди BoundedQueue<T> с ёмкостью, заданной в конструкторе:

фрагмент.cpp C++

```cpp
explicit BoundedQueue(std::size_t capacity);
bool push(T value);  // ждёт свободного места; false — очередь
    закрыта
std::optional<T> pop();  // ждёт элемента; nullopt — закрыта и пуста
std::optional<T> try_pop_for(std::chrono::milliseconds timeout);
void close();
```

Листинг 3.14 — интерфейс BoundedQueue<T>

Требования:
— push блокируется, когда очередь заполнена, и просыпается, когда место освободилось;
— после close() оставшиеся элементы можно дочитать, затем pop возвращает nullopt;
— производители, ждущие свободного места, при close() просыпаются и получают false;
— подумайте, сколько condition variable вам нужно и почему.

Проверка: тесты на порядок элементов (FIFO для одного производителя), на блокировку при заполнении, на таймаут try_pop_for, на закрытие при ждущих производителях и потребителях; стресс-тест 4 на 4 потока, ёмкость 8, каждый элемент получен ровно один раз; всё под TSan 10 раз подряд.

Задача 3.3 (средняя). Пинг-понг и хоровод

Два потока по очереди печатают ping и pong, N раз каждый, строго чередуясь: ping, pong, ping, pong…

Затем обобщите: K потоков с номерами 0..K−1 печатают свои номера строго по кругу (0, 1, 2, …, K−1, 0, 1, …), всего N кругов.

Требования: никакого активного ожидания (sleep в цикле или пустой цикл не годятся), никаких глобальных переменных.

Проверка: вывод собирается в вектор и сравнивается в тесте с ожидаемой последовательностью; при K = 8, N = 10 000 загрузка CPU во время работы низкая (потоки спят, а не крутятся); TSan чист. Подумайте, что произойдёт с производительностью при notify_all и при notify_one, и объясните, почему в одном из этих вариантов программа может зависнуть.

Решения и разбор ваших неизбежных ошибок — в главе «Ответы к задачам» в конце книги.

Глава 4. Железо и ОС: что на самом деле происходит под std::mutex

«Гонок нет, локов нет, а работает в шесть раз медленнее». Добро пожаловать в мир кеш-линий.

Первые три главы мы работали с абстракциями: поток, мьютекс, условная переменная. Абстракции хороши, пока не начинаешь мерить. А когда начинаешь, выясняется, что два потока, которые не делят ни одной переменной, мешают друг другу в шесть раз; что незахваченный мьютекс стоит наносекунды, а захваченный — микросекунды; что сигнал SIGTERM прилетает в случайный поток и может прервать его посреди malloc.

Эта глава — экскурсия на один уровень вниз: процессор, кеши, планировщик, сигналы. Не ради эрудиции, а потому что без этой модели в голове невозможно ни объяснить модель памяти C++ (глава 5), ни понять результаты бенчмарков, ни ответить на senior-собеседовании на вопрос «почему ваш шардированный кеш не масштабируется дальше 8 ядер».

К концу главы вы:

— поймёте, как устроены кеши и почему процессор работает с кеш-линиями, а не байтами;

— научитесь находить и устранять false sharing;

— узнаете, что такое store buffer и почему процессор «переставляет» операции с памятью;

— поймёте, как устроен мьютекс в Linux (futex) и сколько он стоит;

— освоите законы Амдала и Густафсона — главные «отрезвители» в разговорах о масштабировании;

— научитесь привязывать потоки к ядрам и узнаете, когда это вредно;

— поймёте, чем опасен fork() в многопоточной программе;

— научитесь правильно обрабатывать сигналы в многопоточном процессе;

— разобьёте хранилище shardkv на шарды и научите сервер останавливаться по SIGTERM.

4.1. Кеши и кеш-линии

Обращение к оперативной памяти занимает порядка 100 нс. За это время современное ядро могло бы выполнить сотни инструкций. Поэтому между ядром и памятью стоит иерархия кешей:

— **L1** — свой у каждого ядра, десятки килобайт, доступ за несколько тактов;

— **L2** — обычно свой у каждого ядра, от сотен килобайт до пары мегабайт, около десятка тактов;

— **L3** — общий для всех ядер процессора, десятки мегабайт, несколько десятков тактов;

— **память** — гигабайты, сотни тактов.

Главное, что нужно запомнить: **кеш работает блоками фиксированного размера — кеш-линиями.** На x86-64 и большинстве ARM это 64 байта (на Apple M-серии — 128). Когда вы читаете один байт, процессор загружает всю линию, в которой он лежит. Когда вы пишете один байт, процессор должен получить всю линию в своё эксклюзивное владение.

Это легко увидеть:

ex02_cache_stride.cpp C++

```cpp
// ex02_cache_stride.cpp — процессор читает память кеш-линиями, а не байтами.
// Трогаем каждый 1-й, 2-й, 4-й ... 64-й байт большого массива. Работы всё меньше, а время?
#include <chrono>
#include <cstdint>
#include <print>
#include <vector>

int main() {
    constexpr std::size_t kSize = 64 * 1024 * 1024;  // 64 МБ: заведомо больше кеша
    std::vector<std::uint8_t> data(kSize, 1);

    for (std::size_t stride : {1, 2, 4, 8, 16, 32, 64, 128, 256}) {
        const auto start = std::chrono::steady_clock::now();
        std::uint64_t sum = 0;
        for (std::size_t i = 0; i < kSize; i += stride) sum += data[i];
        const double ms = std::chrono::duration<double, std::milli>(std::chrono::steady_clock::now() - start).count();
        std::println("шаг {:3} байт: {:7.2f} мс, обращений {:9}, sum={}", stride, ms, kSize / stride, sum);
    }
    // От шага 1 до шага 64 обращений становится в 64 раза меньше, а время падает в разы меньше:
```

ex02_cache_stride.cpp (продолжение) C++

```
    // мы платим за загрузку кеш-линий из памяти, а их приходится
        грузить те же самые.
}
```

Листинг 4.1 — процессор читает память кеш-линиями, а не байтами (ex02_cache_stride.cpp)

На моей машине:

терминал SHELL

```
$ ./ex02_cache_stride
шаг   1 байт:   30.93 мс, обращений  67108864
шаг   8 байт:    9.72 мс, обращений   8388608
шаг  64 байт:    7.42 мс, обращений   1048576
шаг 256 байт:    3.54 мс, обращений    262144
# … вывод сокращён: шаги 2, 4, 16, 32, 128 и контрольная сумма опущены
```

Листинг 4.2 — стоимость обращений с разным шагом

От шага 1 до шага 64 обращений стало в 64 раза меньше, а время упало только в 4 раза. Потому что мы платим не за обращения, а за загрузку кеш-линий, и их приходится загружать те же самые. (После шага 64 время продолжает падать, но не пропорционально: в игру вступает аппаратный префетчер, который угадывает следующие линии заранее. Железо сложнее любой модели, поэтому мы и меряем.)

Отсюда первое правило производительности: **данные, которые используются вместе, должны лежать рядом** (массив структур, std::vector вместо std::list). А в многопоточном коде появляется второе правило, и оно хитрее.

4.2. Когерентность кешей и false sharing

Если у каждого ядра свой кеш, что будет, когда два ядра работают с одной и той же линией? Процессор гарантирует **когерентность**: все ядра видят согласованное содержимое каждой линии. На x86 для этого используется протокол семейства MESI. Каждая линия в каждом кеше находится в одном из состояний:

— **Modified** — линия изменена этим ядром и есть только у него;
— **Exclusive** — линия есть только у этого ядра, не изменена;
— **Shared** — линия есть у нескольких ядер, только для чтения;
— **Invalid** — копия недействительна.

Чтобы **записать** в линию, ядро должно перевести её в Modified, а для этого — инвалидировать копии во всех остальных кешах. Если два ядра поочерёдно пишут в одну линию, она постоянно «перепрыгивает» между ними: каждая запись требует обмена сообщениями между ядрами. Это называют ping-pong кеш-линии.

Самое коварное: ядрам не обязательно писать в одну и ту же **переменную**. Достаточно писать в разные переменные, которые лежат в одной **линии**. Это и есть false sharing (ложное разделение):

ex01_false_sharing.cpp C++

```cpp
// ex01_false_sharing.cpp — у каждого потока СВОЙ счётчик. Гонок
    нет. А тормозит.
#include <atomic>
#include <chrono>
#include <cstdint>
#include <print>
#include <thread>
#include <vector>

#include "cacheline.hpp"

constexpr int kThreads = 2;
constexpr std::int64_t kIters = 50'000'000;

// Плотно: счётчики соседних потоков лежат в одной кеш-линии.
struct Packed {
    std::atomic<std::int64_t> value{0};
};

// С выравниванием: каждый счётчик занимает свою кеш-линию целиком.
struct alignas(kCacheLine) Padded {
    std::atomic<std::int64_t> value{0};
};

```

ex01_false_sharing.cpp (продолжение) C++

```cpp
template <class Counter>
double run() {
    std::vector<Counter> counters(kThreads);  // C++17: vector
        уважает alignas при аллокации
    const auto start = std::chrono::steady_clock::now();
    {
        std::vector<std::jthread> threads;
        for (int t = 0; t < kThreads; ++t) {
            threads.emplace_back([&c = counters[static_cast<std::size
                _t>(t)]] {
                for (std::int64_t i = 0; i < kIters; ++i) {
                    c.value.fetch_add(1, std::memory_order_relaxed);
                        // атомик — чтобы компилятор не свернул цикл
                }
            });
        }
    }
    return std::chrono::duration<double, std::milli>(std::chrono::ste
        ady_clock::now() - start).count();
}

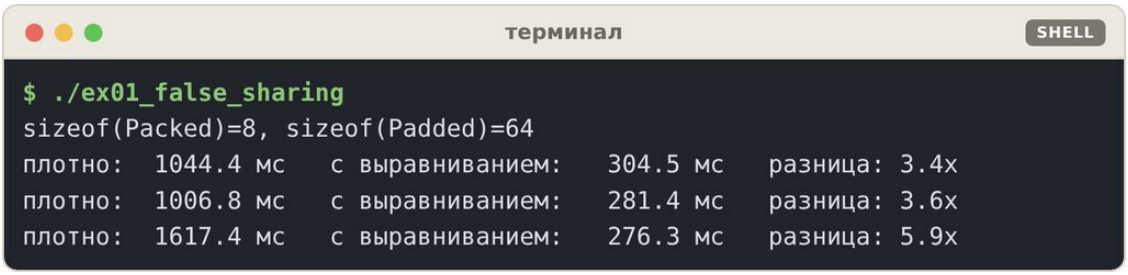
int main() {
    std::println("sizeof(Packed)={}, sizeof(Padded)={}",
        sizeof(Packed), sizeof(Padded));
    for (int round = 0; round < 3; ++round) {
        const double packed = run<Packed>();
        const double padded = run<Padded>();
```

ex01_false_sharing.cpp (продолжение) C++

```cpp
        std::println("плотно: {:7.1f} мс   с выравниванием: {:7.1f}
            мс   разница: {:.1f}x", packed, padded,
                packed / padded);
    }
}
```

Листинг 4.3 — у каждого потока свой счётчик (ex01_false_sharing.cpp)

Результат на моей машине:



Листинг 4.4 — false sharing: плотные и выровненные счётчики

Два потока, у каждого свой счётчик, никаких общих данных с точки зрения программы, — и разница в 3,4–6 раз. Лечение — разложить данные, которые пишут разные потоки, по разным кеш-линиям: alignas(64) или явный padding.

hardware_destructive_interference_size

В C++17 появилась константа std::hardware_destructive_interference_size из <new> — минимальное расстояние между объектами, чтобы избежать false sharing. Две оговорки:

— не все стандартные библиотеки её предоставляют (проверяйте макрос __cpp_lib_hardware_interference_size);

— GCC предупреждает при её использовании (-Winterference-size): значение зависит от флагов -mtune/-mcpu, и если оно попадёт в ABI (например, в размер структуры в заголовке библиотеки), разные единицы трансляции могут разойтись.

Практичное решение — своя константа с запасным значением:

cacheline.hpp C++

```cpp
// cacheline.hpp — размер кеш-линии для выравнивания, с запасным вариантом.
#pragma once

#include <cstddef>
#include <new>

#ifdef __cpp_lib_hardware_interference_size
// GCC предупреждает (-Winterference-size), что значение может отличаться между -march.
// Для нас это нормально: мы используем его только для раскладки в одной сборке.
#if defined(__GNUC__) && !defined(__clang__)
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Winterference-size"
#endif
inline constexpr std::size_t kCacheLine =
    std::hardware_destructive_interference_size;
#if defined(__GNUC__) && !defined(__clang__)
#pragma GCC diagnostic pop
#endif
#else
// Стандартная библиотека не дала значения — берём типичное для x86-64 и большинства ARM.
// (На Apple M-серии линия 128 байт; если целитесь туда — поставьте 128.)
inline constexpr std::size_t kCacheLine = 64;
#endif
```

Листинг 4.5 — размер кеш-линии для выравнивания, с запасным вариантом (cacheline.hpp)

Где искать false sharing

— массивы структур «по одной на поток»: счётчики, статистика, локальные буферы;

— мьютекс и защищаемые им данные рядом с мьютексом **другого** объекта (массив шардов, у каждого свой мьютекс — наш случай в шаге 4);

— флаг остановки рядом с часто изменяемыми полями;

— атомарный счётчик ссылок рядом с данными, которые читают все (это одна из причин, почему std::shared_ptr под нагрузкой бывает медленным).

И не нужно выравнивать всё подряд: каждая структура размером 64 байта вместо 8 — это в 8 раз больше памяти и меньше данных в кеше. Выравнивают только то, что **разные потоки часто пишут**.

4.3. Store buffer: почему процессор «переставляет» операции

Запись в память даже через кеш — дорогая операция: нужно получить линию в эксклюзивное владение. Чтобы не ждать, каждое ядро складывает записи в свой **store buffer** — маленькую очередь, — и продолжает выполнять следующие инструкции. Запись попадает в кеш (и становится видна другим ядрам) чуть позже.

Последствие: другое ядро может увидеть ваши операции не в том порядке, в котором они написаны в программе. Классический пример (store-load reordering):

псевдокод C++

```
// изначально x = 0, y = 0
// Ядро 1:          // Ядро 2:
x = 1;              y = 1;
r1 = y;             r2 = x;
```

Листинг 4.6 — переупорядочивание записи и чтения на двух ядрах

Интуиция говорит: хотя бы одно из r1, r2 будет равно 1. Но на реальном x86 возможен результат r1 == 0 и r2 == 0: обе записи ещё лежат в store buffer'ах, а чтения уже выполнились из кеша. Вы поймаете этот эффект своими руками в задаче 4.2.

Разные архитектуры разрешают разные перестановки. x86 довольно строгий: он переставляет только «запись, потом чтение другого адреса». ARM и POWER разрешают почти всё. Поэтому код, который «годами работал на x86», ломается при переезде на ARM-сервера или Apple Silicon.

Но писать код под конкретный процессор не нужно. Модель памяти C++ абстрагирует всё это: вы описываете, какие гарантии упорядочивания нужны (memory_order), а компилятор вставляет нужные барьеры для конкретной архитектуры. Мьютексы делают это за вас автоматически. Подробно — в главе 5.

4.4. Как устроен мьютекс в Linux: futex

Как можно сделать мьютекс, который почти ничего не стоит, когда свободен? В Linux ответ — futex (fast userspace mutex). Идея:

— состояние мьютекса — обычное целое число в памяти процесса (0 — свободен, 1 — захвачен, 2 — захвачен и есть ждущие);

— **захват без конкуренции:** одна атомарная операция compare-and-swap 0 → 1, полностью в пространстве пользователя. Никаких системных вызовов;

— **захват при конкуренции:** если CAS не удался, поток (обычно после короткого активного ожидания) вызывает системный вызов futex(FUTEX_WAIT), и ядро усыпляет его;

— **освобождение:** атомарная запись 0; если были ждущие — системный вызов futex(FUTEX_WAKE), чтобы разбудить одного.

Померим:

ex03_mutex_cost.cpp C++

```cpp
// ex03_mutex_cost.cpp — сколько стоит мьютекс без конкуренции и с ней.
#include <chrono>
#include <cstdint>
#include <mutex>
#include <print>
#include <thread>
#include <vector>

constexpr std::int64_t kIters = 5'000'000;

double ns_per_op(int threads) {
    std::mutex m;
    std::int64_t counter = 0;  // GUARDED_BY(m)
    const auto start = std::chrono::steady_clock::now();
    {
        std::vector<std::jthread> pool;
        for (int t = 0; t < threads; ++t) {
            pool.emplace_back([&] {
                for (std::int64_t i = 0; i < kIters; ++i) {
                    std::lock_guard lock(m);
                    ++counter;
                }
            });
        }
    }
    const auto ns = std::chrono::duration<double, std::nano>(std::chrono::steady_clock::now() - start).count();
```

ex03_mutex_cost.cpp (продолжение) C++

```cpp
    return ns / static_cast<double>(kIters * threads);
}

int main() {
    std::println("1 поток (без конкуренции): {:5.1f} нс на
        lock+unlock", ns_per_op(1));
    std::println("2 потока (конкуренция):   {:5.1f} нс на
        lock+unlock", ns_per_op(2));
    std::println("4 потока (конкуренция):   {:5.1f} нс на
        lock+unlock", ns_per_op(4));
}
```

```
$ ./ex03_mutex_cost
1 поток (без конкуренции):  21.2 нс на lock+unlock
2 потока (конкуренция):    66.5 нс на lock+unlock
4 потока (конкуренция):    43.2 нс на lock+unlock
```

Листинг 4.7 — сколько стоит мьютекс без конкуренции и с ней (ex03_mutex_cost.cpp)

(Цифры с виртуальной машины; на голом железе захват без конкуренции обычно в разы быстрее.) Даже на этом простом примере видно: при конкуренции каждая операция дороже в 2–3 раза, а ведь критическая секция здесь — один инкремент. Когда секции длиннее, а потоки засыпают в ядре, разница достигает микросекунд: переключение контекста, холодные кеши после пробуждения.

Отсюда вывод для проектирования: **проблема не в мьютексах, а в конкуренции за них.** Сотня мьютексов, на которых никто не сталкивается, дешевле одного, на котором толкаются все. Это ровно та идея, на которой построен шаг 4 проекта.

4.5. Законы Амдала и Густафсона

Закон Амдала. Если доля работы p распараллеливается идеально, а доля (1 − p) выполняется последовательно, то максимальное ускорение на N ядрах:

S(N) = 1 / ((1 — p) + p / N)

Если 95% работы параллельны, то на бесконечном числе ядер ускорение не превысит 1 / 0,05 = 20. Если параллельно 90% — не больше 10. Последовательная часть программы (в том числе критические секции под глобальным мьютексом!) задаёт потолок, который не пробить никаким количеством ядер.

Практический смысл: глобальный мьютекс в shardkv из шага 2 — это последовательная часть. Все операции с хранилищем выполнялись по одной. Сколько ни добавляй ядер, быстрее одного ядра хранилище работать не будет.

Закон Густафсона — оптимистичный взгляд с другой стороны: на практике с ростом числа ядер растёт и объём задачи (больше пользователей, больше данных), а последовательная часть часто остаётся прежней. Тогда ускорение на «большей задаче» почти линейно.

Оба закона верны, они отвечают на разные вопросы. Амдал: «насколько быстрее станет та же задача?» Густафсон: «насколько бóльшую задачу я решу за то же время?» На собеседовании полезно знать оба и уметь сказать, какой из них применим в обсуждаемой ситуации.

4.6. NUMA

На серверах с несколькими процессорными сокетами память физически разделена: у каждого сокета своя «ближняя» память. Обращение к «дальней» памяти (подключённой к другому сокету) заметно дороже. Это называется NUMA (non-uniform memory access).

Что важно знать:

— Linux по умолчанию выделяет физическую память на том узле, где поток впервые к ней **обратился** (first touch). Если главный поток инициализировал огромный массив, вся память оказалась на его узле, и рабочие потоки на другом сокете будут постоянно ходить в дальнюю память;

— поток, который планировщик перенёс на другой сокет, теряет и кеши, и близость к своей памяти;

— false sharing между сокетами стоит ещё дороже, чем внутри одного процессора.

Инструменты: numactl для запуска с политикой размещения, lscpu и numactl --hardware для просмотра топологии. На ноутбуке и в большинстве облачных виртуалок NUMA-узел один (у меня тоже), и эти эффекты не видны. На двухсокетном сервере базы данных — ещё как видны.

4.7. Планировщик, приоритеты и привязка к ядрам

Стандартный C++ ничего не знает о приоритетах потоков и ядрах. Когда это нужно, используется native_handle() — нативный дескриптор потока (в Linux — pthread_t), с которым работают системные API:

ex04_affinity.cpp C++

```cpp
// ex04_affinity.cpp — привязка потока к ядру через native_handle(). Только Linux.
// Здесь заканчивается стандартный C++ и начинается pthreads.
#include <pthread.h>
#include <sched.h>

#include <cstring>
#include <print>
#include <thread>

// Возвращает 0 при успехе, иначе код ошибки pthread.
int pin_to_cpu(std::jthread& t, int cpu) {
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(cpu, &set);
    return pthread_setaffinity_np(t.native_handle(), sizeof set, &set);
}

int main() {
    const int cpus = static_cast<int>(std::thread::hardware_concurrency());
    std::println("доступно аппаратных потоков: {}", cpus);

```

ex04_affinity.cpp (продолжение) C++

```
    std::jthread worker([](std::stop_token st) {
        while (!st.stop_requested()) std::this_thread::yield();
    });
    const int target = cpus > 1 ? cpus - 1 : 0;
    if (const int rc = pin_to_cpu(worker, target); rc != 0) {
        std::println("не удалось привязать: {}", std::strerror(rc));
    } else {
        std::println("поток привязан к ядру {}", target);
    }
    // Не забудьте: привязка — это обещание планировщику «не двигай
        меня».
    // Если это ядро занято чем-то ещё, поток будет ждать, а не
        уйдёт на свободное.
}
```

Листинг 4.8 — привязка потока к ядру через native_handle() (ex04_affinity.cpp)

Когда привязка (affinity) полезна:

— задачи с жёсткими требованиями к задержкам: поток на изолированном ядре не вытесняется и не теряет кеши (подробнее о real-time — в главе 12);

— архитектура thread-per-core, где каждое ядро обслуживает свою часть данных (тоже глава 12);

— NUMA: держать поток рядом с его памятью.

Когда вредна — почти во всех остальных случаях. Планировщик Linux неплохо балансирует нагрузку сам. Привязанный поток не может уйти на свободное ядро, если его ядро занято. Привязка, которая дала +10% на бенчмарке, может дать −50% в продакшене, где рядом крутятся другие процессы.

Про приоритеты: обычные политики планирования Linux (SCHED_OTHER) почти не дают гарантий; real-time политики (SCHED_FIFO, SCHED_RR) дают, но требуют прав и аккуратности — поток с real-time приоритетом в бесконечном цикле может заморозить систему. Это тема главы 12.

4.8. fork() в многопоточном процессе

fork() создаёт копию процесса. Но в дочернем процессе оказывается **только один поток** — тот, который вызвал fork(). Все остальные потоки просто исчезают. А память, включая состояние всех мьютексов, копируется как есть.

Сценарий катастрофы:

1. Поток A захватил мьютекс внутри malloc (у аллокатора есть внутренние блокировки).
2. Поток B в этот момент вызывает fork().
3. В дочернем процессе мьютекс аллокатора «захвачен», но потока A, который должен его отпустить, там нет.
4. Дочерний процесс вызывает malloc и зависает навсегда.

То же самое с любыми мьютексами: логгер, пул соединений, ваш собственный кеш.

Правило: **в многопоточном процессе после fork() в дочернем процессе можно вызывать только async-signal-safe функции до вызова exec().** Фактически — только сразу заменить процесс через exec. Если нужно запустить внешнюю программу, используйте posix_spawn() — он сделан именно для этого. Если нужен fork без exec (например, для снапшота памяти, как в Redis), делайте его до создания потоков или проектируйте систему так, чтобы дочерний процесс не трогал ничего, что могло быть захвачено.

4.9. Сигналы и потоки

Сигналы — механизм UNIX, придуманный задолго до потоков. В многопоточной программе с ними три проблемы.

Кому достанется сигнал. Процессно-направленный сигнал (например, SIGTERM от kill) доставляется **любому** потоку, у которого он не заблокирован. Какому именно — решает ядро.

Обработчик прерывает поток в произвольной точке. Например, посреди malloc или посреди критической секции с захваченным мьютексом. Поэтому внутри обработчика сигнала можно вызывать только async-signal-safe функции (их список есть в man signal-safety): никакого malloc, никакого printf, никаких мьютексов. Если обработчик попытается захватить мьютекс, который уже держит прерванный поток, это мгновенный дедлок: прерванный поток не продолжится, пока не закончится обработчик.

Что можно сделать в обработчике. Почти ничего: записать в переменную типа volatile std::sig_atomic_t или в lock-free std::atomic, и выйти. Это единственный законный случай для volatile в многопоточной теме (подробнее в главе 5).

Правильный шаблон: выделенный поток сигналов

Вместо асинхронных обработчиков используем синхронное ожидание:

1. в самом начале main, **до создания любых потоков**, блокируем нужные сигналы через pthread_sigmask. Маска сигналов наследуется, поэтому все потоки, созданные дальше, тоже их блокируют;

2. создаём выделенный поток, который ждёт сигналы через sigwait (или sigtimedwait);

3. sigwait возвращает номер сигнала как обычное значение функции, и дальше это **обычный код**: можно брать мьютексы, логировать, аллоцировать;

4. поток сигналов переводит сигнал в понятный остальной программе механизм — например, request_stop() на общем stop_source.

ex05_signal_thread.cpp C++

```cpp
// ex05_signal_thread.cpp — сигналы в многопоточной программе:
    выделенный поток с sigwait.
// Только POSIX. Запустите и нажмите Ctrl+C или пошлите kill -TERM
    <pid>.
#include <signal.h>
#include <unistd.h>

#include <cerrno>
#include <chrono>
#include <ctime>
#include <print>
#include <stop_token>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

int main() {
    // 1. Блокируем SIGINT и SIGTERM ДО создания любых потоков.
    //    Маска сигналов наследуется: все потоки, созданные дальше,
        тоже их блокируют.
    sigset_t signals;
    sigemptyset(&signals);
    sigaddset(&signals, SIGINT);
    sigaddset(&signals, SIGTERM);
    pthread_sigmask(SIG_BLOCK, &signals, nullptr);

```

ex05_signal_thread.cpp (продолжение) C++

```cpp
    std::stop_source stop;

    // 2. Выделенный поток синхронно ждёт сигнал. Это ОБЫЧНЫЙ код, не обработчик сигнала:
    //    здесь можно брать мьютексы, аллоцировать, логировать — все ограничения сняты.
    std::jthread signal_thread([&stop, signals](std::stop_token own) {
        const timespec tick{.tv_sec = 0, .tv_nsec = 200'000'000};
        while (!own.stop_requested()) {
            // sigtimedwait вместо sigwait: раз в 200 мс проверяем, не пора ли выйти самим.
            const int sig = sigtimedwait(&signals, nullptr, &tick);
            if (sig < 0) continue;  // EAGAIN (таймаут) или EINTR
            std::println("получен сигнал {}, начинаем остановку", sig);
            stop.request_stop();
            return;
        }
    });

    // 3. Рабочие потоки ничего не знают о сигналах. Они знают только stop_token.
    std::vector<std::jthread> workers;
    for (int i = 0; i < 3; ++i) {
        workers.emplace_back([token = stop.get_token(), i] {
            while (!token.stop_requested())
                std::this_thread::sleep_for(50ms);
```

ex05_signal_thread.cpp (продолжение) C++

```cpp
            std::println("воркер {} остановлен", i);
        });
    }

    std::println("pid {}: работаем, пошлите SIGINT или SIGTERM", getpid());
    workers.clear();          // ждём воркеров: они выйдут после сигнала
    signal_thread.request_stop();
    std::println("корректное завершение");
}
```

Листинг 4.9 — сигналы в многопоточной программе (ex05_signal_thread.cpp)

Почему sigtimedwait, а не sigwait: чтобы поток сигналов мог выйти сам, когда программа завершается по другой причине (например, из-за фатальной ошибки). sigwait ждал бы вечно, и деструктор jthread повис бы на join.

Рабочие потоки при этом ничего не знают о сигналах — только о stop_token. Это правильное разделение: обработка ОС-специфичных событий в одном месте, остальной код переносим.

4.10. Грабли с ревью

— Массив счётчиков или структур «по одной на поток» без выравнивания.

— Выравнивание всего подряд «на всякий случай» — память и кеш тоже не бесплатные.

— Код, который полагается на порядок операций с памятью без синхронизации («на x86 же работает»).

— Один горячий мьютекс на весь сервис и надежда, что ядра помогут.

— Привязка потоков к ядрам без замеров в реальном окружении.

— fork() без немедленного exec() в многопоточной программе.

— printf, malloc, мьютексы или логирование внутри обработчика сигнала.

— Обработка сигналов через signal() в многопоточной программе вместо выделенного потока.

4.11. Что спросят на собеседовании

Что такое false sharing и как его найти? (middle/senior) Разные потоки пишут в разные переменные в одной кеш-линии, линия прыгает между ядрами. Найти — профилировщиком (perf c2c на Linux) или по симптому «нет общих данных, а не масштабируется». Лечить — разносить по линиям.

Что такое кеш-линия и почему это важно для многопоточности? (middle) Единица обмена между памятью и кешем (обычно 64 байта); когерентность работает на уровне линий, поэтому записи соседних данных разными ядрами конфликтуют.

Почему процессор может переставить операции с памятью? (senior) Store buffer, внеочередное исполнение, оптимизации компилятора. x86 переставляет store-load, ARM почти всё. Модель памяти C++ абстрагирует это.

Сколько стоит мьютекс? (senior) Без конкуренции — атомарная операция в пространстве пользователя (futex), при конкуренции — системные вызовы и переключение контекста. Проблема в конкуренции, а не в мьютексе.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.