HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 09 – Condition variable

Глава 1

От активного ожидания к поддержке ядра: Проблема синхронизации

Одной из фундаментальных задач многопоточного программирования является координация действий между потоками. Часто возникает сценарий producer-consumer, где один поток готовит данные, а второй должен дождаться их готовности перед обработкой. В этой главе мы рассмотрим эволюцию подходов к ожиданию события: от наивного активного цикла до использования механизмов ядра операционной системы.

Активное ожидание (Polling)

Самый очевидный способ заставить поток ждать изменения состояния — это запустить бесконечный цикл проверки условия. Этот подход называется активным ожиданием (busy wait) или поллингом (polling).

```cpp
#include <atomic>
#include <chrono>
#include <thread>
#include <iostream>

std::atomic<bool> is_ready(false);

void producer() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    is_ready.store(true);
}

void consumer() {
    // Busy wait цикл
    while (!is_ready.load()) {
        // Поток бесконечно проверяет переменную
    }
    std::cout << "Data processed\n";
}

int main() {
    std::thread t(producer);
    consumer();
```

```
    t.join();
}
```

С точки зрения логики программы этот код корректен: поток `consumer` действительно дождется установки флага. Однако с точки зрения архитектуры системы это решение неприемлемо.

Цикл `while` в `consumer` представляет собой непрерывный поток инструкций для процессора. Планировщик операционной системы (OS Scheduler) видит, что поток готов к исполнению, и выделяет ему квант времени (time slice). Поток тратит этот квант на бесконечную проверку `false`, не выполняя полезной работы.

> **Важно!**
>
> Активное ожидание приводит к загрузке ядра процессора на 100% ("греет воздух"). Это увеличивает энергопотребление, нагрев CPU и может приводить к троттлингу частот. Кроме того, в условиях конкуренции за ресурсы (например, на одноядерной машине или при большом количестве потоков), активный поток отбирает время у потока-продюсера, замедляя наступление ожидаемого события (Starvation).

Попытка смягчить проблему с помощью `std::this_thread::yield()` не решает ее фундаментально:

```
while (!is_ready.load()) {
    std::this_thread::yield();
}
```

`yield()` сообщает планировщику, что поток готов отдать остаток своего кванта времени. Однако, если в очереди планировщика нет других готовых к исполнению потоков (или их приоритет ниже), управление немедленно вернется к текущему потоку, и цикл продолжится. В нагруженной системе это лишь увеличивает накладные расходы на переключение контекста (Context Switch Overhead).

Наивное решение: Сон (Sleep)

Следующая итерация попытки решить проблему без специальных примитивов синхронизации — добавление задержки в цикл проверки.

```
while (!is_ready.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(2));
}
```

Здесь поток добровольно уходит в состояние ожидания, и планировщик исключает его из очереди на исполнение на указанное время. Загрузка CPU падает почти до нуля. Однако появляются две критические проблемы:

1. Латентность (Latency)

Если событие (`is_ready = true`) произойдет через 0.1 мс после начала сна, поток-потребитель узнает об этом только через 1.9 мс, когда проснется. В высокопроизводительных системах

(HFT, Real-Time Systems) задержка в миллисекунды недопустима. Уменьшение времени сна приближает нас обратно к активному ожиданию и лишним переключениям контекста.

2. Зависимость от планировщика

Аргумент функции `sleep_for` — это *минимальное* время сна. Реальное время зависит от гранулярности системного таймера и загруженности планировщика.

- В Windows дефолтная гранулярность таймера может достигать 15.6 мс. Запрос на сон в 1 мс фактически усыпит поток на 15 мс.
- Операционная система не гарантирует мгновенного пробуждения. Поток проснется, попадет в очередь готовых к исполнению, и только когда до него дойдет очередь, он получит процессорное время.

Использование "магических констант" (например, 2 мс) — признак плохого дизайна. Время выполнения операций может варьироваться на порядки в зависимости от железа, и подобранная константа будет либо слишком большой (латентность), либо слишком маленькой (нагрузка на CPU).

Атомарные операции и Livelock

Иногда разработчики пытаются реализовать механизмы синхронизации на базе сложных атомарных операций, таких как `compare_exchange_weak` (CAS), надеясь избежать использования "тяжелых" мьютексов.

```
void wait_for_flag(std::atomic<int>& counter) {
    int expected = 1;
    // Пытаемся атомарно изменить значение, если оно равно expected
    while (!counter.compare_exchange_weak(expected, 0)) {
        expected = 1; // Сброс expected, так как CAS обновляет его при неудаче
        // Busy wait
    }
}
```

Хотя `std::atomic` обеспечивает корректность с точки зрения гонок данных (Data Race Free), с точки зрения использования ресурсов это тот же самый busy loop. Атомарные операции (особенно RMW — Read-Modify-Write) значительно дороже обычных операций чтения/записи, так как требуют блокировки шины памяти или использования протоколов когерентности кэшей (MESI) между ядрами.

В таком сценарии возможен **Livelock**. В отличие от Deadlock, где потоки заблокированы и не двигаются, в Livelock потоки активно выполняют инструкции, меняют свои локальные состояния, но глобальный прогресс системы отсутствует или крайне мал из-за постоянной конкуренции за кэш-линию (cache contention).

Необходимость поддержки ядра

Все рассмотренные выше методы работают в пространстве пользователя (User Space) и пытаются эмулировать ожидание. Эффективное решение невозможно без участия ядра операционной системы.

Нам нужен механизм, который позволяет:

1. Перевести поток в состояние блокировки (Blocked/Waiting), полностью убрав его из очереди планировщика.
2. Гарантировать, что поток будет разбужен **только** тогда, когда наступит определенное событие (или придет сигнал).
3. Выполнить пробуждение максимально быстро после наступления события.

В Linux для этого используется системный вызов `futex` (fast userspace mutex), в Windows — `WaitOnAddress` или объекты ядра (Event, Semaphore). В стандарте C++ эти механизмы абстрагированы в примитив **Condition Variable** (условная переменная), который мы рассмотрим в следующей главе.

Глава 2

Механизм Condition Variable: Корректное ожидание

Как мы выяснили в предыдущей главе, для эффективного ожидания событий необходима поддержка со стороны операционной системы. В стандартной библиотеке C++ таким механизмом является `std::condition_variable`. Это примитив синхронизации, который позволяет одному или нескольким потокам заблокироваться (уйти в сон) до тех пор, пока другой поток не отправит уведомление о том, что состояние разделяемых данных изменилось.

Триада синхронизации

Для корректной работы с условными переменными всегда требуются три компонента, работающие в связке:

1. **Разделяемое состояние** (Shared State) — данные, изменения которых мы ждем (например, флаг `bool is_ready` или очередь задач).
2. **Мьютекс** (`std::mutex`) — для защиты доступа к разделяемому состоянию.
3. **Условная переменная** (`std::condition_variable`) — механизм сигнализации.

Важная особенность интерфейса `std::condition_variable::wait` заключается в том, что он принимает не просто мьютекс, а `std::unique_lock<std::mutex>`. Использование `std::lock_gua` здесь невозможно. Это продиктовано механикой работы: `wait` должен иметь возможность программно разблокировать и снова заблокировать мьютекс, чем `lock_guard` (владеющий мьютексом до конца области видимости) не управляет.

Рассмотрим канонический пример взаимодействия Producer и Consumer:

```
#include <mutex>
#include <condition_variable>
#include <iostream>
#include <thread>

std::mutex m;
std::condition_variable cv;
bool ready = false; // Разделяемое состояние
int data = 0;

```

```cpp
void producer() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        // 1. Захватываем мьютекс для модификации состояния
        std::lock_guard<std::mutex> lk(m);
        data = 42;
        ready = true;
    } // Мьютекс освобождается здесь

    // 2. Уведомляем ожидающий поток
    cv.notify_one();
}

void consumer() {
    // 1. Захватываем мьютекс через unique_lock
    std::unique_lock<std::mutex> lk(m);

    // 2. Ждем выполнения условия
    // wait принимает блокировку и предикат
    cv.wait(lk, []{ return ready; });

    // 3. Здесь мьютекс снова захвачен, а ready == true
    std::cout << "Data received: " << data << "\n";
}
```

Механика метода wait

Что именно происходит внутри вызова `cv.wait(lk)`? Это сложная составная операция, которую можно разложить на следующие этапы:

1. **Проверка предиката (опционально):** Если передан предикат (лямбда `[]{ return ready; }`), он выполняется. Если он возвращает `true`, метод немедленно возвращает управление, поток продолжает работу.
2. **Атомарное освобождение и сон:** Если предикат вернул `false` (или не был передан), `condition_variable` атомарно выполняет два действия:
   - Разблокирует мьютекс (`lk.unlock()`).
   - Переводит текущий поток в список ожидания на данной условной переменной и усыпляет его (Syscall `futex wait` в Linux).

   Атомарность здесь критически важна: между освобождением мьютекса и уходом в сон не может "вклиниться" другой поток и отправить уведомление, которое мы бы пропустили (проблема Lost Wakeup).
3. **Ожидание:** Поток спит. Он не потребляет CPU.
4. **Пробуждение:** Поток просыпается по сигналу (`notify`) или системному событию.
5. **Захват мьютекса:** Перед тем как вернуть управление пользовательскому коду, `wait` обязан снова захватить мьютекс (`lk.lock()`). Если мьютекс занят другим потоком, наш поток блокируется на мьютексе.

Ложные пробуждения (Spurious Wakeups)

Один из самых неочевидных и опасных аспектов работы с условными переменными — это феномен ложных пробуждений.

> **Важно!**
>
> Поток, заблокированный в `cv.wait()`, может проснуться **даже если никто не вызывал** `notify_one()` или `notify_all()`.

Это не баг стандартной библиотеки C++, а особенность реализации планировщиков операционных систем (в частности, POSIX Threads). Причины могут быть разными: обработка сигналов процессом, оптимизации реализации условных переменных (например, использование широковещательного пробуждения вместо точечного в некоторых гонках внутри ядра).

Из-за этого факта использование `if` для проверки условия является грубой ошибкой:

Некорректный код (Anti-pattern)

```
std::unique_lock<std::mutex> lk(m);
if (!ready) {
    cv.wait(lk); // ОШИБКА!
}
// Если произошло ложное пробуждение, мы попадем сюда,
// хотя ready все еще false.
process(data); // Обработка невалидных данных
```

Единственно верный паттерн использования — цикл `while`. Стандартный метод `cv.wait(lock, predicate)` является синтаксическим сахаром для следующего цикла:

Корректная реализация цикла ожидания

```
while (!ready) {
    cv.wait(lk);
}
```

Алгоритм работы потребителя с учетом ложных пробуждений выглядит как диаграмма состояний:

1. Захватить мьютекс.
2. Проверить условие. Если `true` — выйти из цикла.
3. Если `false` — освободить мьютекс и уснуть.
4. ... (сон) ...
5. Проснуться (по любой причине).
6. Захватить мьютекс.
7. Перейти к шагу 2.

Таким образом, даже если ОС разбудит поток "просто так", цикл снова проверит переменную `ready`, увидит `false` и снова отправит поток спать.

Проблема потерянного уведомления (Lost Wakeup)

Почему так важно изменять разделяемую переменную (`ready = true`) под мьютексом? Рассмотрим сценарий без мьютекса:

Сценарий гонки (Race Condition)

```
// Поток 1 (Producer)          // Поток 2 (Consumer)
                               while (!ready) {
ready = true;                      // 1. Прочитал ready (false)
cv.notify_one();                   // 2. Отправил сигнал (в пустоту)
                                   cv.wait(lk); // 3. Уснул навечно
                               }
```

Если уведомление отправляется в момент между проверкой условия (шаг 1) и входом в состояние сна (шаг 3), оно "теряется". Условная переменная не хранит состояние (в отличие от Семафора или Event в Windows), она работает как мгновенный сигнал. Если в момент свистка на платформе никого не было, никто этот свисток не услышит и позже не узнает о нем.

Мьютекс гарантирует, что проверка условия и переход в режим ожидания являются атомарной последовательностью относительно изменения условия.

Стратегии уведомления: notify_one vs notify_all

`std::condition_variable` предоставляет два метода пробуждения:

- `notify_one()`: Будит *один* из ожидающих потоков. Какой именно — не определено стандартом (зависит от планировщика). Используется, когда событие подразумевает эксклюзивную обработку (один элемент в очереди — один рабочий поток).
- `notify_all()`: Будит *все* ожидающие потоки. Используется, когда событие касается всех (например, сигнал остановки приложения или открытие "ворот" барьера).

Использование `notify_all()` в сценарии, когда задачу может выполнить только один поток, приводит к проблеме **Thundering Herd** ("Эффект разорвавшейся бомбы" или "Стадо бизонов"). Представьте 100 потоков, ждущих на одном мьютексе. При `notify_all()` все 100 просыпаются. Все 100 пытаются захватить один и тот же мьютекс. Один выигрывает, 99 блокируются снова (теперь уже на мьютексе, а не на CV). Это вызывает огромный всплеск переключений контекста и деградацию производительности.

Оптимизация: Уведомление вне блокировки

Рассмотрим внимательно код продюсера:

```
{
    std::lock_guard<std::mutex> lk(m);
    ready = true;
    cv.notify_one(); // (A) Уведомление под мьютексом
}
```

В варианте (A) происходит следующее:

1. Продюсер будит консьюмера (сигнал внутри ядра).
2. Консьюмер просыпается в ядре, планировщик ставит его на исполнение.
3. Консьюмер пытается выйти из `wait`, для чего ему нужно захватить мьютекс `m`.
4. Но мьютекс `m` все еще захвачен продюсером!
5. Консьюмер снова блокируется, теперь уже ожидая освобождения мьютекса.
6. Продюсер выходит из скоупа, освобождает мьютекс.
7. Консьюмер снова просыпается и захватывает мьютекс.

Это называется "Hurry up and Wait". Чтобы избежать лишнего переключения контекста, можно вынести уведомление из-под мьютекса:

Оптимизированный Producer

```
{
    std::lock_guard<std::mutex> lk(m);
    ready = true;
} // Мьютекс освобожден
cv.notify_one(); // Уведомление
```

Теперь, когда консьюмер проснется, мьютекс будет свободен, и он сможет захватить его сразу.

На заметку

Эта оптимизация безопасна в большинстве случаев, но требует осторожности, если объект `cv` может быть уничтожен сразу после разблокировки мьютекса (например, если ожидающий поток отвечает за удаление структуры данных, содержащей CV). Однако в стандартных сценариях долгоживущих очередей или пулов потоков это рекомендуемый паттерн.

В следующей главе мы используем эти знания для построения более высокоуровневых примитивов синхронизации, таких как События и Семафоры.

Глава 3

Построение абстракций синхронизации: Event, Semaphore, Latch

Механизм `std::condition_variable`, рассмотренный ранее, является низкоуровневым строительным блоком. В прикладном коде прямое использование связки `mutex + cv + bool` часто приводит к дублированию кода и ошибкам.

Для решения типовых задач синхронизации строятся высокоуровневые абстракции, инкапсулирующие управление состоянием и блокировками. В этой главе мы реализуем три фундаментальных примитива: Событие (Event), Семафор (Semaphore) и Защелку (Latch), разбирая их внутреннюю механику и сценарии использования.

Событие (Event)

Абстракция **Event** (в терминологии Windows API) или "одноразовый сигнал" позволяет одному потоку уведомить другой (или несколько других) о наступлении определенного факта. В отличие от сырой условной переменной, класс `Event` хранит свое состояние внутри.

Существует два основных типа событий:

- **Manual Reset Event:** После перехода в сигнальное состояние остается в нем до явного сброса. Все потоки, пришедшие в `Wait`, проходят сквозь него немедленно.
- **Auto Reset Event:** После пробуждения одного потока событие автоматически сбрасывается в несигнальное состояние.

Реализуем вариант с ручным сбросом (Manual Reset), так как он наиболее наглядно демонстрирует работу с разделяемым флагом.

Реализация класса Event

```
class Event {
public:
    // Перевод события в сигнальное состояние
    void Signal() {
        {
            // Захват мьютекса обязателен для изменения флага
            std::lock_guard lock{mtx_};
            ready_ = true;
        }
```

```
        // Оптимизация: уведомление после разблокировки мьютекса
        // позволяет проснувшимся потокам сразу захватить его
        cv_.notify_all();
    }

    // Блокирующее ожидание события
    void Wait() {
        std::unique_lock lock{mtx_};
        // Стандартный паттерн ожидания с предикатом
        cv_.wait(lock, [this] { return ready_; });
    }

    // Неблокирующая проверка состояния
    bool Ready() {
        std::lock_guard lock{mtx_};
        return ready_;
    }

    // Сброс события (для повторного использования)
    void Reset() {
        std::lock_guard lock{mtx_};
        ready_ = false;
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    bool ready_ = false;
};
```

В методе `Signal` мы используем `notify_all`, так как событие подразумевает, что "информация стала доступна", и это может быть интересно множеству подписчиков.

Семафор (Semaphore)

Если мьютекс обеспечивает **эксклюзивный** доступ (в критической секции находится ровно 1 поток), то семафор управляет пулом ресурсов, допуская одновременный доступ для N потоков.

Семафор содержит внутренний счетчик доступных "токенов" (разрешений).

- **Acquire (P-операция):** Если счетчик > 0, уменьшает его и продолжает работу. Если счетчик $== 0$, поток блокируется.
- **Release (V-операция):** Увеличивает счетчик и будит один из ожидающих потоков.

Сценарий использования: Ограничение количества одновременных подключений к базе данных или внешнему API (Rate Limiting). Например, если внешний сервис разрешает не более 10 параллельных запросов, семафор с начальным значением 10 обеспечит соблюдение этого лимита на уровне архитектуры приложения.

Реализация Counting Semaphore

```
class Semaphore {
public:
    explicit Semaphore(int initial_count) : counter_{initial_count} {}

    // Захват ресурса (аналог lock)
    void acquire() {
        std::unique_lock lock{mutex_};
        // Ждем, пока появятся свободные слоты (counter > 0)
        not_empty_.wait(lock, [this] { return counter_ > 0; });
        --counter_;
    }

    // Освобождение ресурса (аналог unlock)
    void release() {
        {
            std::unique_lock lock{mutex_};
            ++counter_;
        }
        // Будим только один поток, так как освободился всего один слот
        not_empty_.notify_one();
    }

private:
    int counter_;
    std::mutex mutex_;
    std::condition_variable not_empty_;
};
```

Важное отличие от мьютекса: у семафора нет концепции "владельца". Поток, вызвавший `acquire`, не обязан вызывать `release`. Разрешение может быть возвращено любым другим потоком (хотя в классическом RAII-подходе это происходит в деструкторе гарда).

Начиная с C++20, в стандарте доступны `std::counting_semaphore` и `std::binary_semaphore`, которые могут быть реализованы более эффективно (через атомики и футексы без тяжелых мьютексов), но приведенная реализация демонстрирует их логическую суть.

Защелка (Latch)

Защелка (Latch) — это примитив синхронизации обратного отсчета. Она инициализируется значением N. Потоки могут уменьшать это значение (`Arrive`). Потоки могут ждать, пока значение не станет равным нулю (`Wait`). В отличие от `Barrier`, защелка одноразовая: после достижения нуля она не сбрасывается.

Сценарий использования: Ожидание завершения инициализации N сервисов перед запуском основного цикла обработки, или ожидание завершения N параллельных подзадач в алгоритме MapReduce.

Наивная реализация

Latch на базе мьютекса

```cpp
class Latch {
public:
    explicit Latch(int count) : count_{count} {}

    void Arrive() {
        std::lock_guard lock{mtx_};
        if (--count_ == 0) {
            cv_.notify_all(); // Будим всех ожидающих
        }
    }

    void Wait() {
        std::unique_lock lock{mtx_};
        cv_.wait(lock, [this] { return count_ == 0; });
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    int count_;
};
```

В этой реализации есть проблема производительности. Если 100 рабочих потоков завершают работу почти одновременно и вызывают `Arrive`, они создают высокую конкуренцию (contention) на одном мьютексе `mtx_`. Каждое уменьшение счетчика требует захвата и освобождения мьютекса, что дорого.

Оптимизация с использованием std::atomic

Заметим, что мьютекс нам нужен по факту только для одного события: уведомления ожидающего потока (когда счетчик переходит из 1 в 0). Все остальные декременты (из 100 в 99, из 99 в 98 и т.д.) можно выполнить атомарно без полной блокировки.

Оптимизированный Latch

```cpp
#include <atomic>

class Latch {
public:
    explicit Latch(int count) : count_{count} {}

    void Arrive() {
        // Атомарный декремент. fetch_sub возвращает СТАРОЕ значение.
        // Блокировка не захватывается для декремента.
        if (count_.fetch_sub(1) == 1) {
            // Если старое значение было 1, значит теперь стало 0.
            // Именно этот поток "выключил свет" и должен уведомить остальных.

            // Формально notify_all можно вызывать без лока, но
            // в некоторых реализациях CV это может быть безопасно
```

```
            // только если мы уверены в времени жизни объекта.
            // Для строгости стандарта захватим лок или используем atomic notify
            ↪ (C++20).
            std::lock_guard lock{mtx_};
            cv_.notify_all();
        }
    }

    void Wait() {
        std::unique_lock lock{mtx_};
        // Предикат читает атомик
        cv_.wait(lock, [this] { return count_.load() == 0; });
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    std::atomic<int> count_; // Теперь это atomic
};
```

В оптимизированном варианте 99 из 100 потоков выполнят лишь одну процессорную инструкцию `lock xadd` (на x86) и продолжат выполнение, не затрагивая мьютекс и планировщик ОС. Лишь последний поток заплатит цену за синхронизацию. Это существенно снижает накладные расходы при большом количестве воркеров.

> **Важно!**
>
> C++20 вводит `std::latch` и `std::barrier`, которые реализованы максимально эффективно (часто вообще без использования `std::mutex` и `std::condition_variable`, опираясь только на атомики и системные вызовы типа futex).

Глава 4

Управление потоками: Blocking Queue и Thread Pool

В предыдущих главах мы рассматривали механизмы синхронизации на примере взаимодействия небольшого количества потоков. Однако в реальных высоконагруженных системах создание нового потока (`std::thread`) на каждую задачу является архитектурной ошибкой.

Стоимость создания потока в современных операционных системах не является нулевой. Она включает в себя:

- Аллокацию стека (обычно 1–8 МБ адресного пространства).
- Создание структур данных ядра (Kernel Thread Control Block).
- Системные вызовы для регистрации потока в планировщике.

Замеры показывают, что запуск потока может занимать от десятков до сотен микросекунд. Если полезная нагрузка задачи сопоставима с этим временем (например, вычисление квадратного корня или короткий сетевой запрос), система будет тратить больше времени на администрирование потоков, чем на вычисления.

Решением является паттерн **Thread Pool** (Пул потоков): создание фиксированного набора потоков при старте приложения и переиспользование их для выполнения множества задач. Центральным элементом этой архитектуры выступает **Blocking Queue** (Блокирующая очередь).

Блокирующая очередь (Blocking Queue)

Блокирующая очередь — это потокобезопасный буфер, реализующий паттерн Producer-Consumer.

- Метод `Put` добавляет элемент в очередь и уведомляет ожидающие потоки.
- Метод `Take` извлекает элемент. Если очередь пуста, вызывающий поток блокируется до появления данных.

Ключевой сложностью при реализации очереди является не сама синхронизация доступа (она тривиальна через мьютекс), а механизм корректного завершения работы (Graceful Shutdown).

Проблема остановки (Shutdown)

Рассмотрим ситуацию, когда приложение завершается. Пул потоков должен остановиться. Если мы просто вызовем деструкторы потоков или `std::terminate`, мы рискуем прервать задачи на середине или получить утечки ресурсов. Нам нужен механизм, который сообщает потокам-потребителям: "Больше данных не будет, доделывайте текущую работу и выходите".

Этот механизм реализуется через внутренний флаг `stopped_` и изменение сигнатуры метода `Take`. Вместо `T` он должен возвращать `std::optional<T>`:

- `std::nullopt` — сигнал о том, что очередь закрыта и пуста.
- `Value` — валидная задача для исполнения.

Реализация UnboundedBlockingQueue

Ниже представлена реализация неограниченной блокирующей очереди с поддержкой механизма остановки.

Реализация блокирующей очереди

```
template <typename T>
class UnboundedBlockingQueue {
public:
    bool Put(T value) {
        std::lock_guard<std::mutex> guard(mutex_);
        if (stopped_) {
            return false; // Нельзя добавлять в закрытую очередь
        }
        buffer_.push_back(std::move(value));
        not_empty_.notify_one(); // Будим одного воркера
        return true;
    }

    std::optional<T> Take() {
        std::unique_lock<std::mutex> guard(mutex_);

        // КРИТИЧЕСКИЙ МОМЕНТ: Предикат ожидания
        // Мы ждем, пока:
        // 1. Очередь остановлена (stopped_ == true)
        // ИЛИ
        // 2. В очереди есть данные (!buffer_.empty())
        not_empty_.wait(guard, [this] {
            return stopped_ || !buffer_.empty();
        });

        // После пробуждения проверяем:
        // Если очередь остановлена И пуста -> возвращаем nullopt
        if (stopped_ && buffer_.empty()) {
            return std::nullopt;
        }

        // Иначе забираем элемент
        T result = std::move(buffer_.front());
        buffer_.pop_front();
```

```
        return result;
    }

    // Graceful Shutdown: доработать оставшиеся задачи
    void Close() {
        CloseImpl(/*clear=*/false);
    }

    // Hard Shutdown: выбросить все задачи и остановиться
    void Cancel() {
        CloseImpl(/*clear=*/true);
    }

private:
    void CloseImpl(bool clear) {
        std::lock_guard<std::mutex> guard(mutex_);
        stopped_ = true;
        if (clear) {
            buffer_.clear();
        }
        // Важно: будим ВСЕХ, чтобы они проснулись, увидели stopped_
        // и корректно завершили свои циклы
        not_empty_.notify_all();
    }

private:
    std::mutex mutex_;
    std::condition_variable not_empty_;
    bool stopped_{false};
    std::deque<T> buffer_; // std::deque эффективнее vector для FIFO
};
```

Важно!

Обратите внимание на вызов `not_empty_.notify_all()` в методе `CloseImpl`. Если 10 потоков спят в методе `Take`, и мы закроем очередь, нам нужно разбудить их всех, чтобы они могли проверить флаг `stopped_` и вернуть `std::nullopt`. Если использовать `notify_one`, 9 потоков останутся спать навечно, и программа зависнет при выходе (deadlock при `join`).

Пул потоков (Thread Pool)

Пул потоков представляет собой владельца рабочих потоков (Workers) и очереди задач. Для хранения произвольных задач используется `std::function<void()>`. Это техника стирания типа (Type Erasure), позволяющая хранить в одном контейнере лямбда-выражения, функторы и указатели на функции с разным контекстом захвата, при условии, что они имеют сигнатуру вызова `void()`.

Реализация ThreadPool

```cpp
#include <vector>
#include <thread>
#include <functional>

class ThreadPool {
public:
    using Task = std::function<void()>;

    explicit ThreadPool(size_t num_threads) {
        workers_.reserve(num_threads);
        for (size_t i = 0; i < num_threads; ++i) {
            // Запускаем потоки, каждый из которых исполняет метод Run
            workers_.emplace_back([this] { Run(); });
        }
    }

    // Добавление задачи
    void Submit(Task task) {
        queue_.Put(std::move(task));
    }

    // Явное ожидание завершения
    void Join() {
        queue_.Close(); // Сообщаем потокам о закрытии
        for (auto& t : workers_) {
            if (t.joinable()) {
                t.join(); // Ждем завершения каждого потока
            }
        }
    }

    // Деструктор обеспечивает безопасность (RAII)
    ~ThreadPool() {
        Join();
    }

private:
    // Цикл рабочего потока
    void Run() {
        // Цикл работает, пока Take() возвращает значение
        while (auto task = queue_.Take()) {
            try {
                (*task)(); // Выполняем задачу
            } catch (...) {
                // В простейшем пуле исключения в задачах
                // не должны убивать рабочий поток.
                // В реальном коде здесь нужно логирование.
            }
        }
        // Выход из цикла означает, что очередь закрыта и пуста
    }

private:
```

```
54      UnboundedBlockingQueue<Task> queue_;
55      std::vector<std::thread> workers_;
56  };
```

Анализ архитектуры

Данная реализация демонстрирует классическую схему "One Queue - Multiple Workers".

Достоинства

1. **Балансировка нагрузки:** Свободный поток сразу забирает следующую задачу. Нет простоя, пока есть работа.
2. **Простота:** Вся синхронизация инкапсулирована в `BlockingQueue`. Код пула минималистичен.

Недостатки и ограничения

1. **Contention (Конкуренция):** Все потоки борются за один мьютекс внутри очереди. При очень большом количестве потоков (>32-64) и коротких задачах мьютекс становится узким местом. В таких случаях используют *Work Stealing Queue* (очередь на каждый поток + кража задач).
2. **Исключения:** Если задача выбросит исключение, которое не будет поймано внутри `Run`, функция потока завершится аварийно, и `std::terminate` убьет все приложение. Поэтому вызов задачи обернут в `try-catch`.
3. **Блокирующие задачи:** Если все потоки пула заняты задачами, которые ожидают ввода-вывода или других событий (например, спят), пул встанет (Thread Starvation). Для блокирующих операций рекомендуется использовать отдельные пулы или асинхронный I/O.

Резюме раздела

Пул потоков с блокирующей очередью — это стандарт де-факто для параллельной обработки CPU-bound задач в C++. Он позволяет амортизировать накладные расходы на создание потоков и контролировать уровень параллелизма в системе.