Курс: *Deep Dive into Systems*

Глава 1: Signals & Event Loops

Автор курса: Expert System

Дата: 19 декабря 2025 г.

Содержание

1 Асинхронная модель и Системные события: Signals, Timers, inotify

1.1 Введение в асинхронность и доставку сигналов

Работа с сигналами в UNIX-подобных системах представляет собой один из старейших механизмов межпроцессного взаимодействия (IPC) и обработки исключительных ситуаций. Ключевая особенность сигналов — их асинхронная природа.

Определение: Сигнал

Программное прерывание, доставляемое процессу операционной системой. Обработчик сигнала (signal handler) может быть вызван в произвольный момент времени, прерывая нормальное исполнение инструкций пользовательского кода (user-space).

В момент доставки сигнала ядро приостанавливает выполнение основного потока инструкций, сохраняет контекст процессора, модифицирует стек пользователя (или переключается на альтернативный стек сигналов) и передает управление функции-обработчику. Это накладывает строгие ограничения на код обработчика.

Предупреждение

Из-за асинхронности вызова, компилятор не может предсказать момент изменения переменных внутри обработчика. Если переменная используется в основном цикле и модифицируется в обработчике, она должна быть объявлена как `volatile sig_atomic_t`. Без спецификатора `volatile` оптимизатор может закэшировать значение в регистре и никогда не увидеть изменений («бесконечный цикл ожидания»).

1.1.1 Процесс Init и иммунитет к сигналам

В иерархии процессов Linux особую роль играет процесс с идентификатор процесса (PID) 1, известный как `init`. Он является корнем дерева процессов и имеет специфическую защиту на уровне ядра.

Процессу `init` нельзя отправить сигнал, на который у него явно не установлен обработчик. Это «костыль» ядра (kernel workaround), предназначенный для предотвращения случайного завершения системы. Из этого следует важный вывод: процессу `init` невозможно отправить сигналы `SIGKILL` (9) и `SIGSTOP`, так как эти сигналы технически невозможно перехватить (установить на них обработчик). Следовательно, ядро просто игнорирует их доставку для PID 1, если только сам `init` не был написан с учетом их обработки (что невозможно для данных сигналов).

1.2 Классификация сигналов: Синхронные и Асинхронные

Источники сигналов можно разделить на две фундаментальные категории в зависимости от их происхождения относительно исполняемого потока.

1.2.1 Асинхронные сигналы

Генерируются внешними событиями или другими процессами. Примером служит системный вызов `kill`, который отправляет сигнал от одного процесса другому (с проверкой прав доступа). Пользовательское действие `Ctrl+Z` в терминале отправляет `SIGSTOP`, который переводит процесс в состояние «stopped» (исключает из планировщика ОС), а `SIGCONT` возобновляет его исполнение.

1.2.2 Синхронные сигналы (Traps)

Являются прямым следствием выполнения конкретной инструкции процессора.

- **SIGSEGV (Segmentation Fault):** Доступ к невалидной области памяти (отсутствует маппинг в таблице страниц или нарушение прав доступа).
- **SIGBUS:** Ошибка шины, часто возникающая при невыровненном доступе к памяти на архитектурах, не поддерживающих его, или при доступе к файлу через `mmap`, если файл был усечен.
- **SIGFPE:** Ошибка арифметической операции (деление на ноль, переполнение).

Самый простой способ вызвать синхронный сигнал программно — функция `raise()`, которая, по сути, реализует `kill(getpid(), sig)`. Стандарт POSIX гарантирует, что сигнал от `raise` будет доставлен синхронно: обработчик вызовется до возврата из функции.

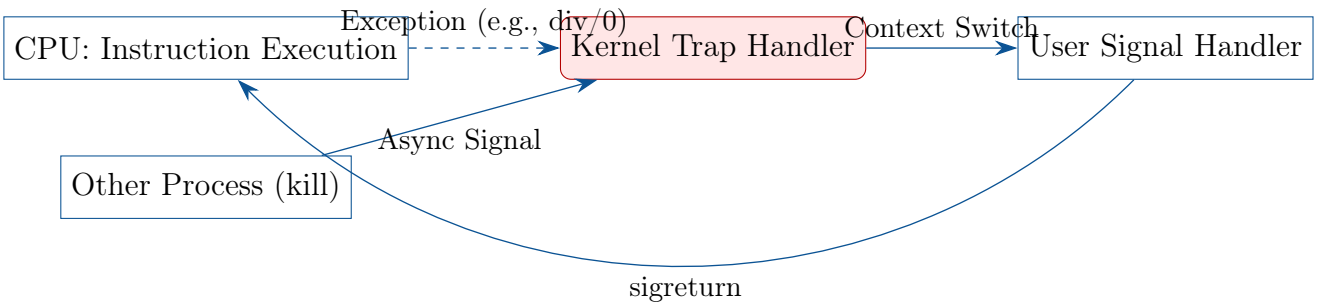

Рис. 1 – Пути доставки синхронных и асинхронных сигналов

1.3 Обработка синхронных ошибок памяти

Обычно обработка синхронных сигналов вроде `SIGSEGV` бессмысленна — нельзя просто вернуться к инструкции, вызвавшей сбой (она снова вызовет сбой). Однако существуют архитектурные паттерны, использующие это поведение.

1.3.1 Lazy Process Migration

Один из продвинутых сценариев — «ленивая» миграция процессов между машинами.

1. На целевой машине создается процесс-пустышка.
2. При попытке исполнения кода или доступа к данным происходит `SIGSEGV` (память не выделена).
3. Установленный обработчик `SIGSEGV` перехватывает управление.
4. Обработчик определяет адрес сбоя, подгружает нужную страницу памяти по сети с исходной машины, выполняет `mmap`.
5. Обработчик завершается, и инструкция перезапускается — теперь уже успешно.

1.3.2 Блокировка синхронных сигналов

Возникает вопрос: что произойдет, если заблокировать `SIGSEGV` с помощью `sigprocmask`, а затем вызвать ошибку сегментации?

Примечание

Операционная система не может отложить доставку синхронного сигнала (в отличие от асинхронного), так как продолжение исполнения невозможно. Ядро Linux в такой ситуации принудительно завершает процесс («убивает»), даже если сигнал заблокирован. Это компромисс реализации: программа считается некорректной.

Если же попытаться разблокировать сигнал внутри его собственного обработчика и вызвать ошибку снова, возникнет бесконечная рекурсия. Каждый новый вызов обработчика создает новый стековый фрейм, что неизбежно приведет к исчерпанию стека (Stack Overflow) и аварийному завершению.

1.4 Эволюция API: signalfd и Event Loop

Асинхронные обработчики сигналов сложны в отладке и ограничены в возможностях (можно использовать только *async-signal-safe* функции). Современный подход в Linux — интеграция сигналов в общий цикл событий (Event Loop) через файловые дескрипторы.

Механизм `signalfd` позволяет принимать сигналы синхронно, вычитывая их как данные из специального файловый дескриптор (FD).

```c
#include <sys/signalfd.h>
#include <signal.h>
#include <unistd.h>

int main() {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);

    // 1. Block signals so they are not delivered via standard async handlers
    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        handle_error("sigprocmask");

    // 2. Create a file descriptor for reading signals
    int sfd = signalfd(-1, &mask, 0);
    if (sfd == -1)
        handle_error("signalfd");

    // The sfd can now be added to epoll or read via blocking read()
    struct signalfd_siginfo fdsi;
    ssize_t s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
    // ...
}
```

Листинг 1 – Использование signalfd с блокировкой

При чтении из `signalfd` возвращается структура `signalfd_siginfo`. В ней содержится детальная информация о сигнале: PID отправителя, `uid`, код завершения (для `SIGCHLD`) и т.д.

Любопытная деталь реализации структуры — наличие в конце поля-заполнителя (padding):

```
uint8_t __pad[28]; // Pad to 128 bytes
```

Размер 28 байт (вместе с остальными полями доводит размер структуры до 128 байт) оставлен для прямой совместимости (forward compatibility). Если в будущем ядру потребуется передавать дополнительные данные с сигналом, они займут место этого паддинга, и старые программы (скомпилированные с текущим размером структуры) продолжат корректно работать, просто не читая новые поля.

1.5 Управление процессами: pidfd

Традиционный системный вызов `waitpid` имеет архитектурный недостаток, связанный с переиспользованием PID. Поскольку идентификаторы процессов — это ограниченный ресурс (обычно 16-битное число), после завершения процесса и вызова `wait` его PID освобождается. Операционная система может сразу же назначить этот PID новому процессу.

Если родительский процесс "промедлит"с ожиданием, он может случайно вызвать `waitpid` или отправить сигнал (`kill`) уже новому, ни в чем не повинному процессу, который занял тот же номер. Это состояние гонки (Race Condition).

Для решения этой проблемы в Linux 5.3+ введен механизм `pidfd`.

Определение: pidfd

Файловый дескриптор, ссылающийся на конкретный экземпляр процесса, а не на его числовой идентификатор. Даже если процесс завершится и его PID будет переиспользован, `pidfd` будет гарантированно указывать на "старый"(уже мертвый) процесс, предотвращая ошибочную отправку сигналов.

`pidfd` также интегрируется с `epoll`: дескриптор становится доступным для чтения (readable), когда процесс завершается. Это позволяет унифицировать ожидание сетевых событий, сигналов и завершения дочерних процессов в одном цикле.

1.6 Мониторинг файловой системы: inotify

Для отслеживания изменений в файловой системе (создание, удаление, запись) вместо неэффективного поллинга (периодического вызова `stat`) используется механизм `inotify`.

При работе с `inotify` возникает сложность: события имеют переменный размер. Структура `inotify_event` содержит поле имени файла, длина которого заранее неизвестна.

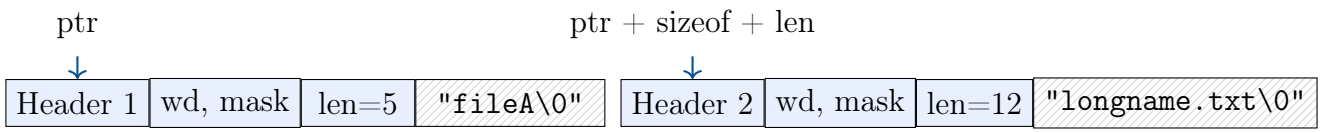

Рис. 2 – Буфер чтения inotify с записями переменной длины

Поле `name` в конце структуры является массивом переменной длины (Variable Length Array concept). При чтении из дескриптора `inotify` программа получает буфер с последовательностью таких структур. Итерирование по ним требует ручного смещения

указателя:

$$\text{next\_ptr} = \text{current\_ptr} + \text{sizeof(struct inotify\_event)} + \text{event->len}$$

Еще один нюанс `inotify` — атомарность переименования. Операция `mv A B` генерирует два события: `MOVED_FROM` и `MOVED_TO`. Чтобы связать их между собой (понять, что файл не просто исчез и появился, а был переименован), ядро заполняет поле `cookie` одинаковым уникальным значением для обоих событий.

Итоги раздела

- Сигналы обрабатываются асинхронно, что требует защиты общих данных (`volatile sig_atomic_t`).
- Синхронные сигналы (Traps) нельзя отложить; их блокировка может привести к аварийному завершению ядра.
- Современные Linux API (`signalfd`, `pidfd`, `inotify`) уходят от концепции callbacks/interrupts к концепции дескрипторов, интегрируемых в единый Event Loop (`epoll`).
- `pidfd` решает критическую проблему безопасности (гонки PID) при управлении процессами.

2 Семантика памяти в C++: Pointer Provenance и Абстрактная машина

В низкоуровневом системном программировании существует фундаментальный разрыв между тем, как память видит процессор (CPU), и тем, как её представляет компилятор языка C/C++. Для процессора память — это плоский линейный массив байтов, а адрес — простое целое число, с которым можно производить любые арифметические операции. Однако для оптимизирующего компилятора C++ (опирающегося на стандарт Abstract Machine) указатель — это гораздо более сложная сущность.

Игнорирование этой разницы приводит к тому, что код, выглядящий абсолютно корректным с точки зрения ассемблера, становится некорректным (Undefined Behavior) с точки зрения стандарта языка, позволяя компилятору удалять проверки безопасности или генерировать нерабочий код.

2.1 Концепция Pointer Provenance

Ключевым понятием, объясняющим поведение современных компиляторов при работе с памятью, является *происхождение указателя* (Provenance).

Определение: Pointer Provenance (Происхождение указателя)

Абстрактное свойство указателя, связывающее его с конкретным объектом аллокации (allocation site). Указатель в семантике C++ можно представить не как число `uint64_t address`, а как кортеж:

$$\text{ptr} = (\text{Allocation\_ID}, \text{Offset})$$

Любая арифметика над указателем изменяет только `Offset`, но не `Allocation_ID`. Доступ к памяти валиден тогда и только тогда, когда адрес физически попадает в диапазон аллокации **и** `Allocation_ID` совпадает с ID объекта по этому адресу.

Рассмотрим классический пример, демонстрирующий эту концепцию. Пусть у нас есть два массива, расположенных в памяти друг за другом (что часто случается на стеке).

```
int* f() {
    int a[3] = {1, 1, 1};
    int b[3] = {2, 2, 2};

    // Assume stack grows downwards and 'b' is placed right after 'a'
    // Address-wise: &a[3] == &b[0]
    int* p = &a[0];
    p += 3; // Formally this is &a[3], a past-the-end iterator

    // Attempting dereference
    // Asm: reads b[0] (value 2)
    // C++: Undefined Behavior
    return *p;
}
```

Листинг 2 – Выход за границы массива с точки зрения ассемблера и C++

Компилятор, анализируя этот код, рассуждает следующим образом: указатель `p` происходит от объекта `a` (Provenance: `a`). Арифметика `p += 3` создает указатель со смещением 3, но с тем же происхождением. Поскольку доступ `*p` выходит за границы объекта `a`, компилятор вправе считать этот код «мертвым» или невозможным (unreachable), даже если физически по этому адресу расположены данные массива `b`.

Примечание

Стандарт разрешает вычислять указатель на элемент, следующий за последним (*past-the-end pointer*), например `&arr[size]`, для использования в итераторах и сравнениях. Однако **разыменовывать** такой указатель запрещено, даже если за массивом есть валидная память.

2.2 Оптимизация аллокаций: Dead Allocation Elimination

Понимание provenance позволяет объяснить агрессивные оптимизации. Рассмотрим функцию, которая выделяет память, но использует её специфическим образом.

```
int example() {
    int* p = new int(42); // Allocation A
    int* q = new int(42); // Allocation B

    // Comparison of pointers from different allocations
    bool equal = (p == q);

    delete p;
    delete q;

    return equal; // Always false
}
```

Листинг 3 – Удаление "неиспользуемой"аллокации

В этом примере компилятор видит, что `p` и `q` имеют разный provenance (разные вызовы `new`). Стандарт гласит, что указатели на разные живые объекты не могут быть равны. Следовательно, выражение `p == q` всегда ложно.

Далее вступает в силу *Dead Allocation Elimination*: раз содержимое памяти по адресам `p` и `q` никак не влияет на наблюдаемое поведение программы (кроме самого факта их существования, который мы только что оптимизировали до `false`), вызовы `new` и `delete` можно полностью удалить.

В результирующем ассемблерном коде не будет вызовов аллокатора `malloc/new`, функция просто вернет 0. Это контринтуитивно, так как `new` имеет побочный эффект (выделение памяти), но компилятор имеет право его удалить, если этот эффект не наблюдаем в рамках абстрактной машины.

2.3 Проблема Roundtrip Casting и XOR Linked List

Преобразование указателя в целое число (`reinterpret_cast<uintptr_t>`) и обратно — это операция, поддерживаемая компиляторами, но с нюансами.

Стандарт гарантирует, что `ptr -> int -> ptr` вернет исходный указатель с исходным provenance. Однако, если над числом была произведена арифметика, provenance теряется.

$$\text{ptr} \xrightarrow{\text{cast}} \text{int} \xrightarrow{\text{math}} \text{new\_int} \xrightarrow{\text{cast}} \text{new\_ptr (Provenance: ???)}$$

Это ставит под угрозу такие классические структуры данных, как **XOR Linked List**.

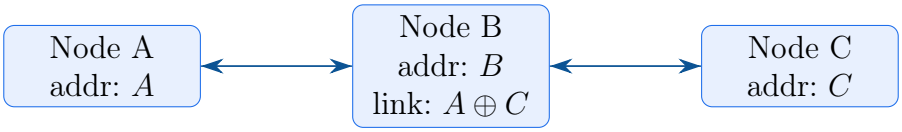

Рис. 3 – Концепция XOR Linked List

В XOR-списке (рис. 3) вместо хранения двух указателей `prev` и `next`, хранится их побитовое исключающее ИЛИ: `link = prev ^ next`. Чтобы перейти к следующему элементу, зная предыдущий, выполняется операция:

$$\text{next} = \text{link} \oplus \text{prev}$$

С точки зрения C++, мы берем целое число, выполняем над ним операцию XOR и кастуем результат в указатель. Полученный указатель не имеет provenance (он «создан из воздуха» с точки зрения компилятора). Доступ по такому указателю формально является Undefined Behavior, хотя на большинстве платформ это работает. Тем не менее, теоретически компилятор может сломать этот код, решив, что раз указатель не получен от аллокатора, то он не может указывать ни на один валидный объект.

2.4 Конфликт оптимизаций LLVM: Case Study (2018)

В 2018 году исследователи обнаружили, что в компиляторе LLVM (используется в Clang, Rust, Swift) комбинация трех корректных по отдельности оптимизаций приводила к некорректной генерации кода.

Рассмотрим следующий код (упрощенная модель):

```
void bug_demo(int *p, int *q) {
    // p and q point to different objects (different provenance)
    // But physically they might be equal after arithmetic logic

    uintptr_t ip = (uintptr_t)p;
    uintptr_t iq = (uintptr_t)q;

```

```
    if (ip == iq) {
        // If addresses match numerically, write to p
        *p = 10;
    }
}
```

Листинг 4 – Код, ломающий оптимизатор LLVM

Цепочка преобразований, которую выполнял компилятор:

1. **Gvn (Global Value Numbering) / Constant Propagation:** Компилятор видит условие `if (ip == iq)`. Внутри ветки `then` он знает, что `ip` равно `iq`. Следовательно, он может заменить использование `p` на `q` (или наоборот), так как их числовые представления равны. *Результат:* замена `*p = 10` на `*q = 10` (или создание эквивалентного указателя из `iq`).
2. **IntToPtr Cast Folding:** Если мы привели `q` к `int`, а потом обратно, это эквивалентно исходному `q`.
3. **Dead Store Elimination (на основе Provenance):** Это критический шаг. Компилятор анализирует запись `*q = 10`. Он знает, что в этой функции (по условиям вызова) мы работаем с объектом `p`. Указатель `q` имеет другой provenance. Стандарт говорит, что доступ к объекту `p` через указатель с provenance `q` невозможен (они не алиасятся). *Вывод компилятора:* запись `*q = 10` недостижима или не влияет на `p`. Инструкция удаления записи удаляется.

Итог: В исходном коде, если адреса совпадали, запись должна была произойти. В скомпилированном коде запись исчезла. Корректная программа сломалась. Это привело к пересмотру модели памяти в LLVM и ограничению оптимизаций при работе с `inttoptr` кастами.

2.5 Практический пример: Hazard Pointers и Placement New

В высокопроизводительных базах данных (например, YDB) часто используются кастомные аллокаторы. Рассмотрим структуру, где заголовок и данные аллоцируются одним куском памяти:

```
struct Chunk {
    int header;
    // Data starts right here, after the header
};

void* mem = malloc(sizeof(Chunk) + sizeof(Data));
Chunk* chunk = new (mem) Chunk(); // Placement new

// Attempting to access data via pointer arithmetic from the header
char* data_ptr = reinterpret_cast<char*>(chunk) + sizeof(Chunk);
Data* data = reinterpret_cast<Data*>(data_ptr);

// Constructing data in place
new (data) Data();
```

Листинг 5 – Опасная арифметика с placement new

Проблема здесь заключается в том, как компилятор видит объект `chunk`. Он был создан как объект типа `Chunk`. Получение указателя на `Data`, который лежит за пределами

`sizeof(Chunk)`, формально является выходом за границы объекта `Chunk`. Хотя физически память выделена одним блоком `malloc`, с точки зрения C++ `chunk` — это указатель на объект конкретного размера. Попытка "шагнуть"за его пределы и обратиться к памяти может быть расценена как UB, если компилятор не увидит связь с исходным `malloc`.

Итоги раздела

- Указатель в C++ — это **адрес + provenance** (информация о происхождении).
- Арифметическое равенство адресов (`0x1000 == 0x1000`) не гарантирует возможность взаимозаменяемости указателей, если они имеют разный provenance.
- Компилятор имеет право удалять «мертвые» аллокации (`new` без использования), даже если это меняет наблюдаемые побочные эффекты.
- Преобразование указателей в целые числа и обратно стирает provenance, что может мешать оптимизатору отслеживать зависимости (alias analysis) и приводить к удалению «лишних» записей в память.
- Для написания корректных аллокаторов и низкоуровневых структур данных необходимо использовать `std::launder` (C++17) или специальные атрибуты компилятора, чтобы «отмыть» указатель и начать новый цикл жизни объекта.

3 Каталог Undefined Behavior и Агрессивные Оптимизации

В основе философии языков C и C++ лежит принцип «Trust the Programmer» (Доверяй программисту). Компилятор исходит из предположения, что программист никогда не пишет некорректный код, не допускает переполнений и не выходит за границы массивов. Это позволяет отказаться от дорогостоящих проверок в рантайме (bounds checking, overflow checking) и применять агрессивные оптимизации.

Однако обратной стороной этой медали является *неопределенное поведение* (Undefined Behavior, UB). Если программа нарушает правила абстрактной машины, стандарт перестает гарантировать что-либо, и компилятор получает право трансформировать код любым образом, полагая, что данная ситуация недостижима.

3.1 Нарушение потока управления (Control Flow UB)

Одним из самых опасных видов UB является нарушение контрактов функций, возвращающих значение.

3.1.1 Отсутствие return в не-void функции

Согласно стандарту, если поток исполнения достигает конца функции, возвращающей значение (не `void`), и не встречает инструкции `return`, возникает Undefined Behavior.

```
int process_data(bool cond) {
    if (cond) {
        return 42;
    }
    // Missing return for the case cond == false
}

int main() {
    process_data(false);
}
```

Листинг 6 – Отсутствие return и генерация кода

При компиляции с оптимизациями (например, `-O2`) компилятор может рассуждать так:

1. Вызов `process_data(false)` приведет к исполнению пути без `return`.
2. Этот путь является UB.
3. Программа, содержащая UB, некорректна.
4. Следовательно, случай `cond == false` невозможен.
5. Можно удалить проверку `if (cond)` и считать, что `cond` всегда истинно, или просто сгенерировать пустую функцию.

На уровне ассемблера это часто приводит к тому, что функция не содержит инструкции возврата (`ret`) или восстановления стека для "невозможной"ветки. Процессор продолжает исполнение инструкций, следующих сразу за телом функции в памяти (fallthrough). Это может быть код следующей функции, данные (интерпретируемые как инструкции) или невыровненный мусор. Часто это заканчивается сигналом `SIGILL` (Illegal Instruction) или `SIGSEGV`.

Примечание

Единственным исключением является функция `main`. Стандарт C++ разрешает не писать `return 0;` в конце `main` — в этом случае компилятор подставляет его автоматически. Для всех остальных функций это строгое UB.

3.1.2 Бесконечные циклы без побочных эффектов

Еще один контринтуитивный аспект оптимизации связан с завершаемостью программ. Стандарт C++ (до C++11 и в определенных контекстах после) гласит, что бесконечный цикл без побочных эффектов (side effects) является UB. Под побочными эффектами понимаются операции ввода-вывода, доступ к `volatile` переменным или атомикам.

Если компилятор видит цикл, который просто вычисляет что-то в регистрах и никогда не завершается, он имеет право удалить этот цикл целиком, считая, что программа не должна застревать навечно.

```
bool check_fermat() {
    // Brute-force a, b, c to infinity or overflow
    for (int a = 1; ; ++a) {
        for (int b = 1; b <= a; ++b) {
            for (int c = 1; c <= a + b; ++c) {
                if (a*a*a + b*b*b == c*c*c) {
                    return true; // Counter-example found!
                }
            }
        }
    }
    // Execution never reaches here
    return false;
}
```

Листинг 7 – «Доказательство» Великой теоремы Ферма через UB

Компилятор анализирует этот код:

1. В цикле нет побочных эффектов (IO, volatile).
2. Если цикл бесконечен, это UB. Значит, цикл **должен** завершиться.
3. Единственный штатный выход из цикла — `return true`.
4. Следовательно, функция всегда возвращает `true`.

Оптимизатор заменяет все тело функции на инструкцию `mov eax, 1; ret`, тем самым "опровергая"теорему Ферма.

Примечание

В языке Rust конструкция `loop {}` является легитимным способом остановить программу или организовать вечный цикл. Однако, так как Rust использует бэкенд LLVM (общий с Clang), в прошлом возникали баги, когда LLVM удалял такие циклы, считая их UB по правилам C++. Это приводило к крашам в абсолютно безопасном (memory safe) коде на Rust.

3.2 Арифметика и Типы данных

3.2.1 Переполнение знаковых целых (Signed Integer Overflow)

В C++ переполнение знаковых типов (`int`, `long`) является UB. Переполнение беззнаковых (`unsigned`) определено как арифметика по модулю 2^N.

Компилятор полагается на то, что переполнения не происходит. Это позволяет делать следующие упрощения:

- `if (x + 1 > x)` → всегда `true`.
- `for (int i = 0; i < N; ++i)` → можно использовать 64-битный счетчик или векторизовать цикл, не беспокоясь о том, что `i` станет отрицательным после `INT_MAX`.

Это может приводить к неожиданным бесконечным циклам:

```
void check(int n) {
    // If n = INT_MAX, theoretically loop should be infinite upon overflow
    for (int i = 0; i < n + 100; ++i) {
        printf("%d\n", i);
    }
}
```

Листинг 8 – Оптимизация цикла с переполнением

Компилятор может удалить проверку `i < n + 100`, если решит, что `n + 100` не может переполниться, или наоборот, превратить цикл в бесконечный, игнорируя условие остановки.

3.2.2 Неинициализированные переменные

Чтение неинициализированной памяти — это не просто получение случайного "мусора". Это получение значения, которое может вести себя нестабильно (quantum state). Переменная `bool b`, которая не была инициализирована, может одновременно оцениваться как `true` в одной ветке оптимизации и как `false` в другой, приводя к выполнению взаимоисключающих блоков кода.

3.3 Strict Aliasing и Type-Based Alias Analysis (TBAA)

Одним из важнейших механизмов оптимизации работы с памятью является анализ алиасинга (Alias Analysis). Компилятору необходимо знать, могут ли два указателя адресовать одну и ту же ячейку памяти.

Определение: Strict Aliasing Rule

Правило строгого алиасинга гласит, что доступ к объекту в памяти может осуществляться только через указатель (или ссылку) совместимого типа.

- Нельзя читать `float` через `int*`.
- Нельзя читать `struct A` через `struct B*`.
- **Исключение:** Тип `char*` (и `unsigned char*`, `std::byte*`) может алиасить любые данные. Это необходимо для реализации `memcpy` и побайтового копирования.

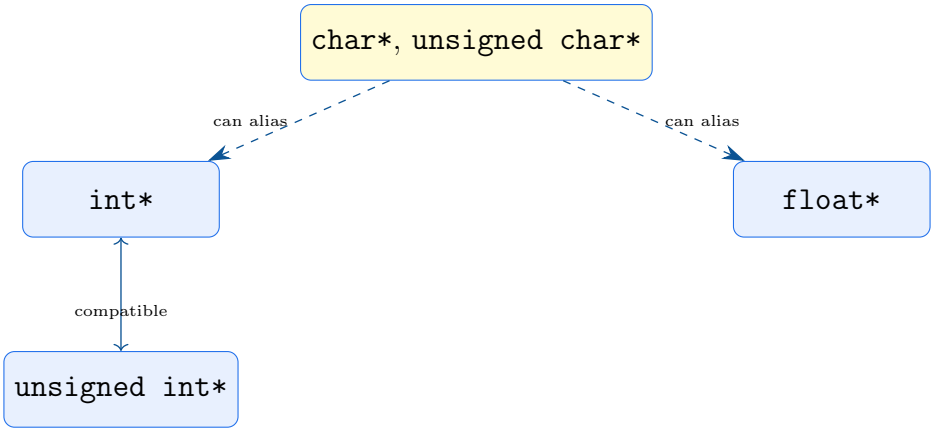

Рис. 4 – Иерархия совместимости указателей (упрощенная)

Нарушение этого правила позволяет компилятору предполагать, что записи в память через указатели разных типов **не влияют** друг на друга. Это критически важно для векторизации и регистрового кэширования.

3.3.1 Пример: std::vector<int> против std::vector<size_t>

Рассмотрим реализацию заполнения вектора значениями.

```
template <typename T>
void fill_vector(std::vector<T>& v, const T& val) {
    for (size_t i = 0; i < v.size(); ++i) {
        v[i] = val;
    }
}
```

Листинг 9 – Влияние типов на оптимизацию цикла

Компилятор генерирует принципиально разный код для `T=int` и `T=size_t`.

Случай 1: `std::vector<int>` Тип элементов — `int`, тип размера вектора — `size_t` (обычно `unsigned long`). Правила Strict Aliasing гарантируют, что запись в `v[i]` (типа `int`) не может изменить поле `v.size()` (типа `size_t`) внутри структуры вектора. *Оптимизация:* Компилятор загружает `v.size()` в регистр один раз перед циклом.

Случай 2: `std::vector<size_t>` Тип элементов — `size_t`, тип размера — `size_t`. Указатель на данные (`v.data()`) имеет тип `size_t*`. Компилятор не может доказать, что массив данных вектора не перекрывается с памятью, где лежит само поле `size` структуры вектора (теоретически, вы могли создать вектор поверх его же заголовка через placement new и злые касты). *Результат:* Компилятор вынужден перезагружать значение `v.size()` из памяти на **каждой** итерации цикла, так как запись `v[i] = val` потенциально могла изменить размер. Это блокирует авто-векторизацию и снижает производительность.

Для решения таких проблем в языке C (и как расширение в C++) существует ключевое слово `restrict`, которое обещает компилятору, что указатель не алиасится ни с чем другим. В стандартном C++ часто приходится копировать размер в локальную переменную перед циклом, чтобы "подсказать"оптимизатору:

```
size_t n = v.size(); // Local copy to avoid reloading
for (size_t i = 0; i < n; ++i) v[i] = val;
```

Итоги раздела

- Undefined Behavior — это не ошибка, а контракт: компилятор обещает быстрый код, если программист обещает соблюдать правила.
- Отсутствие `return` в функции (кроме main) ломает поток управления и может привести к выполнению произвольного кода.
- Бесконечные циклы без побочных эффектов могут быть полностью удалены.
- Strict Aliasing Rule позволяет компилятору эффективно работать с памятью, но требует строгого соблюдения типов. Использование `reinterpret_cast` между несовместимыми типами (например, `int*` в `float*`) ведет к UB.
- При работе с однотипными данными (как в примере с `vector<size_t>`) возможна пессимизация кода из-за страха компилятора перед алиасингом.

4 Многопоточность, Линковка и ODR: Где ломаются абстракции

В предыдущих главах мы рассматривали код преимущественно в контексте одного потока исполнения и одного модуля трансляции. Однако реальные системные приложения работают в многопоточной среде и собираются из сотен объектных файлов. В этих условиях оптимизации, корректные локально, могут приводить к катастрофическим последствиям глобально.

Компиляторы C++ традиционно оптимизируют код, исходя из модели «as-if single-threaded». Это означает, что любое преобразование кода допустимо, если оно сохраняет наблюдаемое поведение *текущего* потока. Однако, когда память становится разделяемым ресурсом, это предположение вступает в конфликт с моделями согласованности памяти.

4.1 Проблема спекулятивных записей (Write Invention)

Одной из самых коварных проблем, возникающих на стыке оптимизации и многопоточности, является «изобретение записи» (Write Invention или Speculative Store).

Рассмотрим функцию, которая инкрементирует глобальную переменную только при выполнении определенного условия.

```
int global_counter = 0;

```

```
void process(bool condition) {
    if (condition) {
        global_counter++;
    } else {
        // Heavy computation that doesn't touch global_counter
        heavy_computation();
    }
}
```

Листинг 10 – Исходный код с условной записью

С точки зрения однопоточной логики, компилятор может попытаться избавиться от условного перехода (который может портить предсказание ветвлений) и заменить его на безусловную арифметику с последующей компенсацией.

```
void process_optimized(bool condition) {
    // Speculative write: we always increment the counter
    int temp = global_counter;
    global_counter = temp + 1;

    if (!condition) {
        // If condition was false, revert the change
        global_counter = temp;
        heavy_computation();
    }
}
```

Листинг 11 – Некорректная оптимизация (Псевдокод)

Для однопоточной программы эти два варианта эквивалентны. Но в многопоточной среде вариант с оптимизацией вносит *состояние гонки* (Data Race).

Представьте, что параллельно работает второй поток, который читает `global_counter`, когда `condition` ложно.

1. **Исходный код:** Второй поток всегда видит старое значение (записи нет).
2. **Оптимизированный код:** Второй поток может увидеть промежуточное значение (инкрементированное), которое через мгновение будет «откачено» первым потоком.

Примечание

Современные стандарты C++ и модели памяти (например, LLVM Memory Model) явно запрещают компиляторам создавать записи в память (store) на тех путях исполнения, где их не было в исходном коде. Это правило известно как «Do not invent stores». Тем не менее, баги такого рода периодически всплывают в старых версиях компиляторов или на экзотических архитектурах.

4.2 One Definition Rule (ODR) и процесс линковки

Еще одна зона риска находится на этапе сборки программы. C++ использует модель раздельной компиляции: каждый `.cpp` файл компилируется в отдельный модуль трансляции (Translation Unit, TU), ничего не зная о других. Связывание этих модулей в единый исполняемый файл выполняет линкер.

Определение: One Definition Rule (ODR)

Правило одного определения гласит:

1. В пределах одного модуля трансляции сущность (переменная, функция, класс) может быть определена только один раз.
2. В всей программе глобальная сущность может быть определена только один раз.
3. **Исключение:** Inline-функции, шаблоны и классы могут быть определены в нескольких модулях трансляции, но эти определения должны быть **побитово идентичны**.

4.2.1 Механизм Weak Symbols

Когда вы объявляете функцию `inline` (или определяете метод прямо в теле класса в хедере), компилятор помечает этот символ как «слабый» (weak symbol) в объектном файле. Это инструкция для линкера: «Если встретишь несколько определений этого символа, выбери любое одно и отбрось остальные». Линкер не проверяет, что тела функций идентичны — он просто верит программисту на слово.

Это открывает дорогу к нарушению ODR, которое не ловится ни компилятором, ни линкером, но приводит к Runtime-ошибкам.

```
// File: module_a.cpp
// Definition 1: sum
inline int logic(int a, int b) { return a + b; }

void check_a() {
    if (logic(1, 2) != 3) abort(); // Expect 3
}

// File: module_b.cpp
// Definition 2: sum + 1 (ERROR: same name, different body)
inline int logic(int a, int b) { return a + b + 1; }

void check_b() {
    if (logic(1, 2) != 4) abort(); // Expect 4
}
```

Листинг 12 – ODR Violation: Разные определения одной inline-функции

Если собрать этот код с оптимизацией `-O2`, компилятор может подставить (заинлайнить) тела функций прямо в места вызова `check_a` и `check_b` *до* этапа линковки. В этом случае программа может «случайно» заработать корректно. Однако при сборке `-O0` вызовы останутся, линкер выберет одну реализацию, и один из модулей сломается.

4.2.2 Static vs Inline

Чтобы избежать таких проблем для вспомогательных функций, следует использовать ключевое слово `static` (в C) или безымянные пространства имен (в C++). Это сообщает линкеру, что символ имеет *внутреннее связывание* (internal linkage) и не должен быть виден за пределами текущего объектного файла. В таком случае в каждом модуле будет своя независимая копия функции.

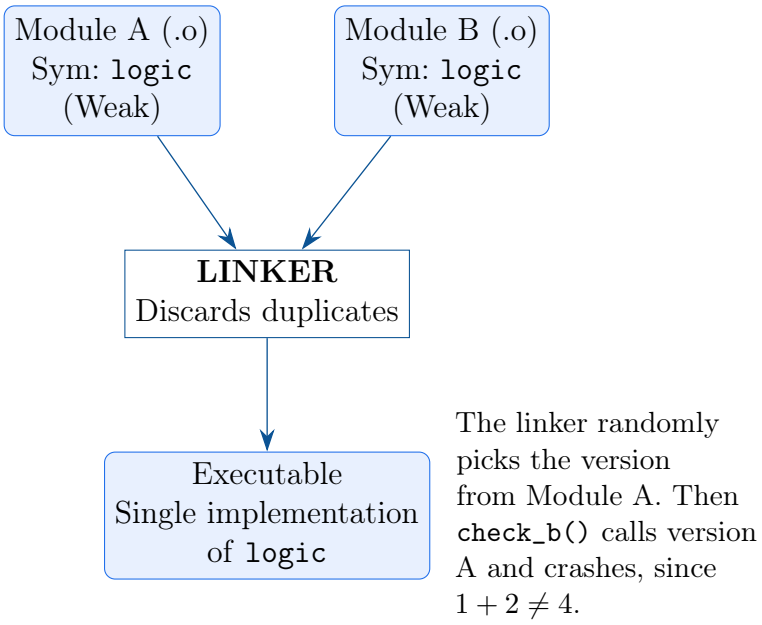

Рис. 5 – Процесс выбора реализации inline-функции линкером

4.3 Нарушение ABI: Кейс библиотеки Abseil

Еще более сложный случай нарушения ODR происходит не из-за кода, а из-за несовпадения настроек препроцессора и компилятора в разных модулях. Это классическая проблема нарушения Application Binary Interface (ABI).

В библиотеке Google Abseil (и многих других, включая STL) часто используется условная компиляция для добавления отладочной информации.

```
// header.h
struct Container {
    int size;
    void* data;

#ifdef DEBUG_BUILD
    // Additional debug fields
    // This changes sizeof(Container) and offsets below!
    int debug_magic;
    const char* creation_stacktrace;
#endif

    int capacity; // Offset depends on DEBUG_BUILD
};
```

Листинг 13 – Условное изменение структуры данных

Представьте, что вы используете предварительно скомпилированную библиотеку, собранную в режиме `Release` (без поля `debug_magic`). Ваше приложение вы собираете в режиме `Debug` (или с включенным Address Sanitizer), где этот макрос определен.

Когда ваше приложение передает указатель на `Container` в библиотечную функцию, библиотека ожидает найти поле `capacity` по смещению `0xC`. Однако в вашей версии структуры по этому смещению находится поле `debug_magic`. Библиотека запишет данные в `debug_magic`, думая, что пишет в `capacity`, или наоборот. Это приведет к порче памяти (Memory Corruption), которую крайне сложно отладить, так как ошибка возникает не в

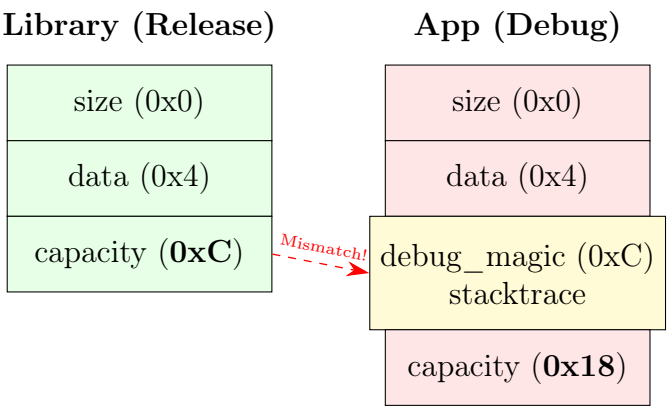

Рис. 6 – Несовпадение раскладки памяти (Layout Mismatch)

момент записи, а спустя долгое время при использовании испорченных данных.

Итоги раздела

- Однопоточные оптимизации могут вносить гонки данных (Data Races) в многопоточный код, если компилятор "изобретает"записи в память.
- One Definition Rule — фундаментальное требование C++. Нарушение ODR (разные тела inline-функций) приводит к неопределенному поведению, так как линкер произвольно выбирает одну из реализаций.
- Слабые символы (Weak Symbols) — механизм, позволяющий множественные определения, но возлагающий ответственность за их идентичность на программиста.
- Нарушение ABI через несовпадение флагов препроцессора (`-DDEBUG`, `-fsanitize`) в разных модулях приводит к несовпадению раскладки структур в памяти. Всегда следите за тем, чтобы все статические библиотеки и основное приложение собирались с идентичными фундаментальными флагами.