Курс: *Архитектура компьютера и ОС*

Лекция 8: Эффективный ассемблер, LTO и динамическая компоновка

Лектор: Имя Фамилия

Дата: 08.11.2025

Содержание

1 Оптимизация ассемблерного кода

Завершающая лекция по ассемблеру посвящена методам его эффективного использования, взаимодействию с ядром и компоновщиком, а также созданию программ, полностью независимых от стандартной библиотеки.

1.1 Проблема раздельной компиляции

Ранее мы рассматривали вызов ассемблерной функции из C++ с использованием раздельной компиляции. Этот подход имеет существенные недостатки производительности.

Рассмотрим пример с подсчётом суммы арифметической прогрессии. Если функция подсчёта реализована в C++ (и компилируется Clang), компилятор может распознать паттерн и заменить цикл на формулу $O(1)$. Если же функция вынесена в отдельный `.S` файл, компилятор видит только её объявление и вынужден генерировать цикл $O(n)$.

Более того, сам вызов функции между единицами трансляции (translation units) — это не бесплатная операция. Он включает:

- Инструкцию `call`, которая сохраняет адрес возврата в стек и выполняет переход (jump).
- Инструкцию `ret`, которая извлекает адрес из стека и выполняет косвенный переход (косвенный переход) по нему.

Эти операции вносят накладные расходы, которых можно избежать.

1.2 Встроенный ассемблер (GNU Inline Assembly)

Для устранения накладных расходов на вызов функции можно использовать встроенный ассемблер. Эта конструкция позволяет компилятору вставить ассемблерный код непосредственно в тело C++ функции, избегая `call`/`ret`.

Определение: Синтаксис GNU Inline Assembly

Конструкция `asm` в C/C++ (GCC, Clang) имеет следующий расширенный синтаксис:

```
asm [volatile] ("assembly template"
                : output operands    /* optional outputs */
                : input operands     /* optional inputs */
                : clobber list       /* optional clobbers */
);
```

- **assembly template:** Строковый литерал с ассемблерным кодом. Входы и выходы подставляются как `%0`, `%1` и т.д.
- **output operands:** Список переменных C/C++, в которые нужно записать результат.
- **input operands:** Список переменных/выражений C/C++, которые нужно передать в ассемблер.
- **clobber list:** Список регистров или состояний, которые изменяются (портятся) внутри вставки, о чём компилятор должен знать.

Рассмотрим пример сложения двух чисел (листинг 1).

```
1 long add_asm(long a, long b) {
2     long res;
```

```
    asm (
        "mov %[a_reg], %[res_reg]\n\t" // res = a
        "add %[b_reg], %[res_reg]\n\t" // res += b
        : [res_reg] "=&r" (res) // Output: res, in any register (r)
                                // & = early clobber
        : [a_reg] "r" (a), // Input: a, in any register (r)
          [b_reg] "r" (b) // Input: b, in any register (r)
        : "cc" // Clobbers: "cc" (condition codes / flags)
    );
    return res;
}
```

Листинг 1 – Использование inline asm для сложения

1.2.1 Операнды и ограничения (Constraints)

Компилятор не понимает семантику ассемблерного кода; для него это просто шаблон. Мы должны явно описать интерфейс между C++ и ассемблером с помощью ограничений:

- `"r"`: Поместить переменную в регистр общего назначения (например, `eax`, `rdi`).
- `-r"`: Выходной операнд (`=`), который будет в регистре.
- `"&r"`: Ограничение **Early Clobber** (`&`). Оно сообщает компилятору, что этот выходной регистр (`res_reg`) будет перезаписан *до* того, как все входные операнды (`a_reg`, `b_reg`) будут использованы. Это запрещает компилятору выделять один и тот же физический регистр для `res` и, например, `a`.

1.2.2 Список порчи (Clobbers)

Ассемблерная вставка может иметь побочные эффекты. Инструкция `add` изменяет регистр флагов (EFLAGS/RFLAGS). Если мы не сообщим об этом компилятору, он может ошибочно предположить, что флаги, установленные *до* `asm`-вставки, останутся неизменными *после* неё.

- `"cc"`: Сообщает компилятору, что регистр флагов (condition codes) был изменён.
- `"memory"`: Сообщает, что вставка читает или пишет в память по адресам, неизвестным компилятору. (См. раздел 2.3).
- `"rax"`, `"rcx"` и т.д.: Сообщает, что конкретный регистр был изменён.

1.3 Оптимизация на этапе компоновки (LTO)

Встроенный ассемблер решает проблему вызова, но не проблему оптимизации. Если функция `add` находится в другом `.cpp` файле, компилятор всё ещё не видит её реализацию при компиляции `main.cpp` и не может, например, заинлайнить её.

Определение: Link Time Optimization (LTO)

Link Time Optimization (оптимизация на этапе компоновки) (LTO) — это техника, при которой компилятор генерирует объектные файлы не в виде машинного кода, а в виде Intermediate Representation (промежуточное представление) (IR). На этапе компоновки (линковки) компилятор снова запускается, считывает IR из *всех* объектных файлов и выполняет оптимизации (включая инлайнинг, удаление мёртвого кода, константное сворачивание) так, как если бы весь код находился в одной единице трансляции.

Инфраструктура Low Level Virtual Machine (инфраструктура для построения компиляторов) (LLVM) (используемая Clang) идеально для этого подходит.

- **Без LTO:** `clang++ -O2` → `main.o` (x86-64), `add.o` (x86-64). Линкер просто склеивает их.
- **С LTO (`-flto`):** `clang++ -flto -O2` → `main.o` (LLVM IR), `add.o` (LLVM IR). На этапе линковки `clang` видит IR обеих функций, инлайнит `add` в `main` и может применить оптимизацию (например, свернуть сумму арифметической прогрессии в константу).

Примечание

Объектные файлы LTO, сгенерированные Clang, не являются стандартными ELF-файлами с машинным кодом. Это архивы LLVM IR (LLVM-AR). Для их просмотра вместо `objdump` используется `llvm-dis`. GCC также поддерживает LTO, но обычно встраивает своё IR (GIMPLE) в специальные секции ELF-файлов.

Итоги раздела

- Вызовы функций между `.cpp` и `.S` файлами несут накладные расходы (`call`/`ret`).
- встроенный ассемблер позволяет встроить ассемблерный код в C++, устраняя эти расходы, но требует аккуратного описания интерфейса (входы, выходы, clobbers (список порчи)).
- LTO позволяет компилятору оптимизировать код *между* единицами трансляции, генерируя промежуточное IR вместо машинного кода.

2 Взаимодействие с ядром и побочные эффекты

2.1 Прямой вызов `syscall`

Ассемблерные вставки позволяют нам выполнять системный вызов напрямую, минуя обёртки стандартной библиотеки (libc).

Определение: Соглашение о `syscall` в Linux x86-64

- Инструкция: `syscall`.
- Номер системного вызова: передаётся в `RAX`.
- Аргументы (по порядку): `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9`.
- Возвращаемое значение: в `RAX`.
- **Порча:** Инструкция `syscall` *уничтожает* содержимое `RCX` и `R11`.

Примечание

Соглашение о `syscall` отличается от стандартного System V ABI в 4-м аргументе (`R10` вместо `RCX`). Это связано с тем, что `syscall` использует `RCX` для сохранения адреса возврата (RIP) и `R11` для сохранения RFLAGS, чтобы ядро могло вернуться в пользовательский процесс с помощью инструкции `sysret`.

В листинг 2 показан пример вызова `write` (номер 1) для печати "Hello".

```c
#include <sys/syscall.h> // for SYS_write
#include <unistd.h> // for STDOUT_FILENO

long write_syscall(int fd, const char* buf, size_t count) {
    long ret;
    asm volatile (
        "syscall"
        : "=a" (ret) // Output: in RAX (a)
        : "a" (SYS_write), // Input: syscall number in RAX (a)
          "D" (fd), // Input: arg1 in RDI (D)
          "S" (buf), // Input: arg2 in RSI (S)
          "d" (count) // Input: arg3 in RDX (d)
        : "rcx", "r11", "memory" // Clobbers: syscall clobbers rcx, r11
                                 // "memory" because 'buf' is read
    );
    return ret;
}

int main() {
    const char* msg = "Hello from syscall!\n";
    write_syscall(STDOUT_FILENO, msg, 20);
    return 0;
}
```

Листинг 2 – Системный вызов `write` через inline asm

2.2 `volatile` и побочные эффекты

Что произойдёт, если мы вызовем `write_syscall`, но не будем использовать возвращаемое значение (`ret`)?

```
int main() {
    const char* msg = "Hello...\n";
    // The compiler (with -O2) might DELETE this line!
    write_syscall(STDOUT_FILENO, msg, 10);
    return 0;
}
```

Листинг 3 – Проблема оптимизации

С точки зрения компилятора, функция `write_syscall` (листинг 2 без `volatile`) — это "чёрный ящик который принимает 4 аргумента и возвращает `long`. Если этот `long` не используется, компилятор вправе удалить вызов целиком, следуя правилу "as-if"(программа должна вести себя *так, как если бы* она выполнялась).

Проблема в том, что у `syscall` есть **побочный эффект** (вывод на экран), о котором компилятор не знает.

Определение: asm `volatile`

Ключевое слово volatile перед `asm` (`asm volatile (...)`) запрещает компилятору:

1. **Удалять** эту ассемблерную вставку, даже если её выходные операнды не используются.
2. **Переупорядочивать** её относительно других `volatile` операций (например, доступа к `volatile` переменным).

Это необходимо для всех ассемблерных вставок, имеющих побочные эффекты (side effects), такие как системные вызовы.

2.3 Использование `asm` для барьеров компиляции

Конструкцию `asm volatile` можно использовать для управления оптимизациями компилятора.

2.3.1 Барьер оптимизации (`DoNotOptimize`)

Иногда в бенчмарках нужно C++ значение, чтобы компилятор не "выкинул"всё вычисление этого значения.

```
// Helper function, similar to Google Benchmark
template <class T>
void DoNotOptimize(T const& value) {
    // The asm block does nothing, but "reads" 'value'
    // 'm' = memory operand
    asm volatile("" :: "m" (value) : "memory");
}

// ...
long result = complex_calculation();
DoNotOptimize(result); // Now the compiler MUST
                       // compute 'result'
```

Листинг 4 – Запрет оптимизации переменной

2.3.2 Барьер памяти (Compiler Fence)

```
asm volatile("" ::: "memory");
```

Листинг 5 – Барьер памяти компилятора

Список clobbers (список порчи), содержащий `"memory"`, сообщает компилятору, что эта вставка может читать или писать в *любую* ячейку памяти. Это заставляет компилятор:

- **Сбросить** (spill) все значения из регистров, которые были изменены, обратно в память *до* этой вставки.
- **Загрузить** (reload) значения из памяти *после* этой вставки, если они понадобятся, не полагаясь на кэшированные в регистрах значения.

Это барьер *только для компилятора*, он не генерирует инструкций барьера центральный процессор (CPU) (типа `mfence`).

Итоги раздела

- Системные вызовы в Linux x86-64 выполняются инструкцией `syscall`, используя регистры `RAX`, `RDI`, `RSI`, `RDX`, `R10`...
- Инструкция `syscall` портит `RCX` и `R11`.
- `asm volatile` необходимо использовать, когда вставка имеет побочные эффекты (как `syscall`), чтобы компилятор её не удалил.
- `asm volatile` с `memory` в clobbers служит барьером памяти для компилятора.

3 Указатели, функции и полиморфизм

Знание ассемблера позволяет понять, как реализованы высокоуровневые конструкции C++, такие как указатели на функции и виртуальные методы.

3.1 Указатели на функции и косвенные переходы

Указатель на функцию в C++ — это переменная, хранящая адрес.

```
int f1(int x) { return x; }
int f2(int x) { return x * 2; }
int f3(int x) { return x * 3; }

// Syntax: ret_type (*var_name)(arg_types)
int (*fn_array[])(int) = { f1, f2, f3 };

int main() {
    int index = 1; // Assume this came from user input
    // ...
    int result = fn_array[index](5); // calls f2(5)
}
```

Листинг 6 – Массив указателей на функции

Во что транслируется `fn_array[index](5)`?

1. Загрузка адреса из `fn_array[index]` в регистр (например, `rax`).
2. Загрузка аргумента `5` в `rdi`.
3. Выполнение косвенный переход: `call rax`.

Определение: Указатель на функцию и косвенный переход

Численное значение указателя на функцию — это, как правило, адрес первой инструкции этой функции в секции `.text`. Вызов по такому указателю реализуется CPU через **косвенный вызов** (`call <reg>`), адрес которого неизвестен на этапе компиляции.

3.2 Защита от атак: `endbr64`

Косвенные переходы — основной вектор атак (ROP/JOP), когда злоумышленник получает контроль над регистром (`rax`) и заставляет программу прыгнуть не на начало функции, а в середину другой функции (на "гаджет").

Для борьбы с этим в современных CPU (Intel CET) введена инструкция endbr64.

- Компиляторы (GCC/Clang) теперь вставляют `endbr64` в начало каждой функции.
- Если ОС и CPU включают защиту, любой косвенный переход (`call rax`), который приземляется *не* на инструкцию `endbr64`, вызовет аппаратное исключение (fault).
- Это гарантирует, что косвенные вызовы могут приземляться только на легитимные начала функций.

Примечание

На данный момент (в лекции) Linux использует эту защиту в основном для кода ядра, но не для пользовательских (userspace) приложений. Однако компиляторы всё

равно генерируют `endbr64` для совместимости в будущем.

3.3 Реализация виртуальных функций C++

Динамический полиморфизм в C++ (ключевое слово `virtual`) также построен на косвенных вызовах.

Определение: vptr и vtable

- **vtable (Таблица виртуальных методов):** Статический массив указателей на функции, создаваемый компилятором для *каждого класса*, имеющего виртуальные методы.
- **vptr (Указатель на vtable):** Скрытый указатель, добавляемый компилятором в *каждый объект* такого класса. vptr указывает на vtable, соответствующую реальному типу объекта.

Рассмотрим вызов `a->foo()`:

```
struct A {
    virtual void foo() { /* A's foo */ }
};
struct B : A {
    virtual void foo() override { /* B's foo */ }
};

int main() {
    A* a = new B();
    a->foo(); // <-- How does this work?
}
```

Листинг 7 – Виртуальный вызов

Вызов `a->foo()` (где `a` в `rdi`) транслируется в:

1. `mov rax, [rdi]` ; Загрузить vptr из объекта (this) в `rax`
2. `call [rax]` ; Вызвать функцию по первому адресу в vtable

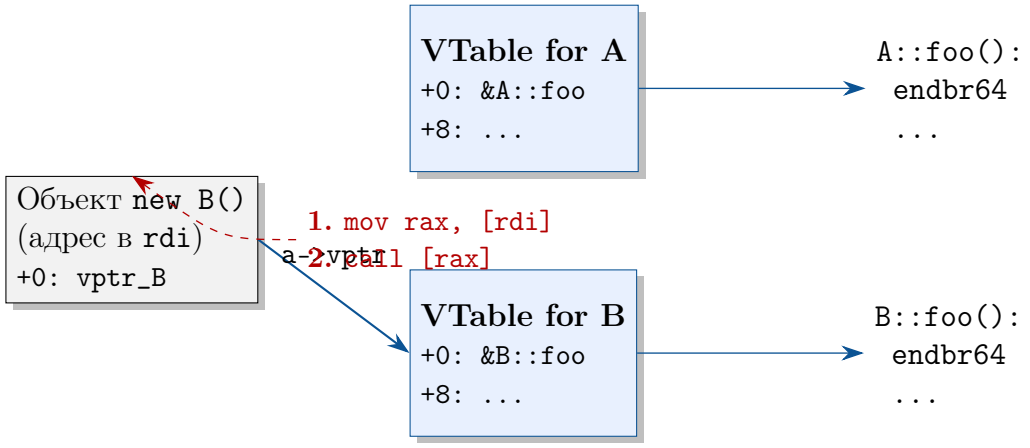

Рис. 1 – Схема виртуального вызова через vptr и vtable

3.3.1 Цена виртуализации

- **Цена по памяти:** +8 байт на *каждый* объект (vptr). Это нарушает принцип C++ "платишь только за то, что используешь т.к. вы платите за vptr, даже если никогда не делаете виртуальных вызовов.
- **Цена по времени:** Виртуальный вызов требует двух обращений к памяти (чтение vptr, чтение адреса из vtable) и косвенный переход, что медленнее прямого `call`.

3.4 JIT-компиляция (Just-in-Time)

Зная, что код — это просто байты в памяти, мы можем генерировать его во время выполнения.

Определение: JIT-компиляция

Just-in-Time (компиляция «на лету») (JIT) — это техника, при которой машинный код генерируется не на этапе компиляции, а во время выполнения программы. Это позволяет создавать код, оптимизированный под конкретные данные (например, SQL-запрос в ClickHouse) или под конкретное железо (например, используя AVX-инструкции, если они доступны на CPU пользователя).

Процесс JIT в Linux:

1. Выделить память с помощью `mmap` с правами `PROT_READ | PROT_WRITE`.
2. Записать в эту память байты машинного кода.
3. Изменить права памяти с помощью `mprotect` на `PROT_READ | PROT_EXEC` (W⊕X).
4. Преобразовать указатель на эту память в указатель на функцию.
5. Вызвать сгенерированную функцию.

```
#include <sys/mman.h> // mmap, mprotect
#include <string.h> // memcpy

// Bytes for the function:
// mov rax, rdi (48 89 f8)
// add rax, rsi (48 01 f0)
// ret (c3)
unsigned char code[] = { 0x48, 0x89, 0xf8, 0x48, 0x01, 0xf0, 0xc3 };

typedef long (*add_func_t)(long, long);

int main() {
    void* mem = mmap(NULL, sizeof(code),
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    memcpy(mem, code, sizeof(code));

    // Important: make the memory executable
    mprotect(mem, sizeof(code), PROT_READ | PROT_EXEC);

    add_func_t fn = (add_func_t)mem;
    long result = fn(10, 20); // result == 30
```

```
24
25      munmap(mem, sizeof(code));
26      return 0;
27  }
```

Листинг 8 – Ручная JIT-компиляция функции `add(a, b)`

Примечание

Приложения вроде ClickHouse или JVM не пишут байты вручную. Они встраивают в себя бэкенд компилятора (например, LLVM) и используют его API для генерации оптимизированного кода "на лету".

Итоги раздела

- Указатели на функции реализуются через косвенный переход (`call rax`).
- Инструкция endbr64 защищает от атак, помечая легитимные цели для таких переходов.
- Виртуальные вызовы C++ используют vptr (в объекте) и vtable (на класс) для реализации косвенный переход, что несёт расходы памяти и времени.
- JIT-компиляция позволяет генерировать машинный код во время выполнения с помощью `mmap` и `mprotect`.

4 Динамическая компоновка

динамическая библиотека (.so) — это код, который компонуется с программой не при сборке, а при запуске.

4.1 Мотивация и основы (.so)

Две основные причины для использования динамическая библиотека (.so):

1. **Экономия памяти:** Множество программ (bash, ls, g++) используют одну и ту же стандартную библиотеку (libc, libstdc++). Вместо того чтобы каждый процесс загружал свою копию, ОС загружает `.so` в память один раз и отображает её в адресные пространства всех процессов.
2. **Оптимизация под платформу:** Можно иметь несколько реализаций `memcpy` (обычную, SSE, AVX) и при запуске загрузчик выберет ту `.so`, которая оптимизирована под текущий CPU.

Загрузчик (`ld.so` в Linux) отвечает за поиск и загрузку всех зависимостей (их можно посмотреть командой `ldd a.out`) перед запуском `_start`.

4.1.1 Позиционно-независимый код (PIC)

Динамическая библиотека не знает, по какому адресу она будет загружена в виртуальную память. Поэтому её код не может использовать абсолютную адресацию.

Определение: Position Independent Code (PIC)

Position Independent Code (позиционно-независимый код) (PIC) — это код, который использует **относительную адресацию** (в x86-64 — относительно регистра `RIP`) для всех переходов и доступа к данным. Это позволяет загружать `.so` в любое место в памяти без необходимости её модификации. Для сборки PIC используется флаг `-fPIC`.

4.2 Механизмы PLT и GOT

Как `main` (скомпилированный) может вызвать `printf` (адрес которой станет известен только при запуске)?

Определение: GOT и PLT

- Global Offset Table (глобальная таблица смещений) (GOT) **(Global Offset Table):** Глобальная таблица смещений. Это массив в секции данных, хранящий *реальные адреса* внешних функций и переменных.
- Procedure Linkage Table (таблица компоновки процедур) (PLT) **(Procedure Linkage Table):** Таблица компоновки процедур. Это секция *исполняемого* кода, содержащая "трамплины"(stubs) — по одному на каждую внешнюю функцию.

По умолчанию в Linux используется ленивое связывание (ленивое связывание).

Процесс первого вызова `printf`:

1. `main` вызывает не `printf`, а трамплин `printf@plt`.
2. Трамплин `printf@plt` прыгает на адрес, указанный в GOT для `printf`.
3. *Изначально* GOT указывает не на `printf`, а обратно на код в PLT.

4. Этот код в PLT кладёт ID функции (`printf`) в стек и прыгает на **динамический резолвер** (часть `ld.so`).
5. Резолвер находит реальный адрес `printf` в загруженной `libc.so`.
6. **(Патчинг)** Резолвер *перезаписывает* запись `printf` в GOT, указывая на реальный адрес.
7. Резолвер прыгает на реальный `printf`.

Второй и последующие вызовы `printf`:

1. `main` вызывает `printf@plt`.
2. Трамплин `printf@plt` прыгает на адрес, указанный в GOT.
3. GOT *уже* содержит реальный адрес `printf`. Происходит прямой переход к `printf`, минуя резолвер.

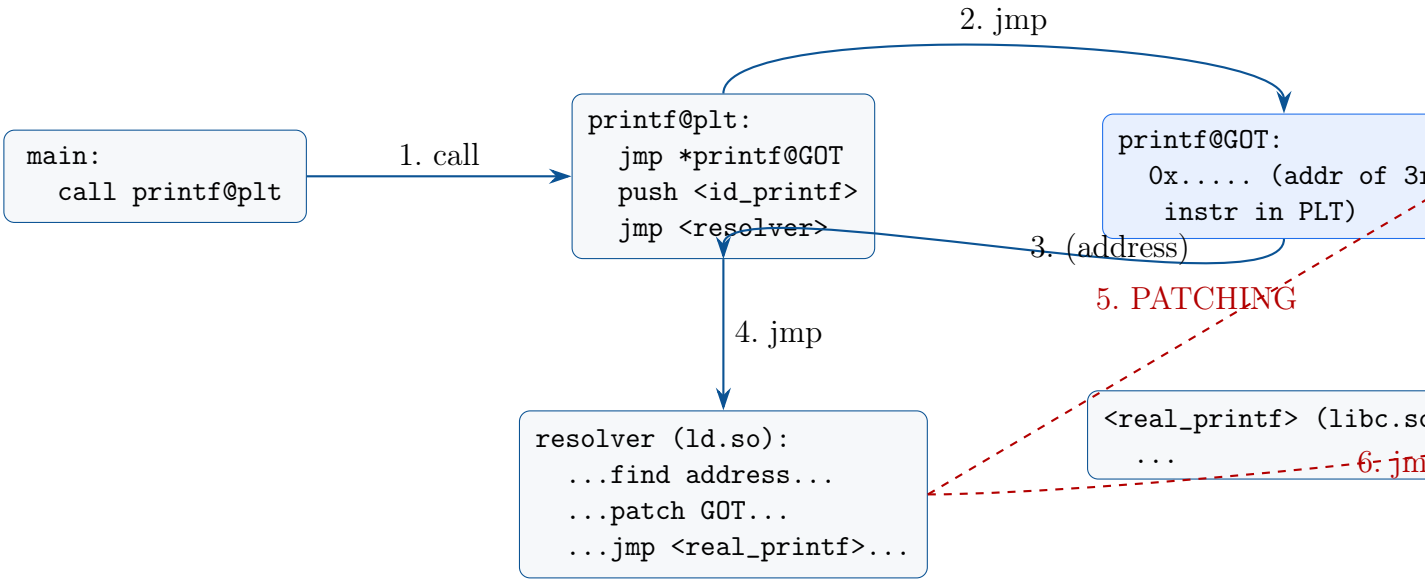

Схема ленивое связывание при *первом* вызове

Рис. 2 – Процесс ленивого связывания (Lazy Binding)

4.3 Перехват вызовов (LD_PRELOAD)

`LD_PRELOAD` — это переменная окружения Linux, которая указывает загрузчику `ld.so`, какую динамическая библиотека (.so) загрузить *в первую очередь*, до `libc` и всех остальных.

Если мы создадим свою `libmyhack.so`, в которой определим функцию `printf`, и запустим программу:

```
$ LD_PRELOAD=./libmyhack.so ./a.out
```

Когда резолвер `ld.so` будет искать `printf`, он сначала найдёт *нашу* реализацию в `libmyhack.so` и использует её.

Примечание

Компилятор может заменять небезопасные функции (как `printf`) на их "проверяющие"аналоги (например, `__printf_chk`) для защиты от переполнения буфера. Если вы хотите перехватить `printf`, вам, возможно, придётся перехватывать `__printf_chk`.

4.4 Ручная загрузка библиотек (dlopen)

Программа может сама загружать `.so` во время выполнения, используя API из `libdl`.

- `dlopen(const char* path, int mode)`: Загружает `.so`. Возвращает `void*` "handle".
- `dlsym(void* handle, const char* symbol)`: Ищет символ (функцию или переменную) по имени в загруженной библиотеке. Возвращает `void*`.
- `dlclose(void* handle)`: Выгружает библиотеку.
- `dlerror()`: Возвращает строку с описанием последней ошибки.

```c
#include <dlfcn.h>
#include <stdio.h>

// 1. Define the signature of the function we're looking for
typedef double (*sin_func_t)(double);

int main() {
    // 2. Load the library
    void* handle = dlopen("libm.so.6", RTLD_LAZY);
    if (!handle) { /* error handling */ }

    // 3. Find the symbol (function)
    void* sym = dlsym(handle, "sin");
    if (!sym) { /* error handling */ }

    // 4. Cast void* to the correct FUNCTION POINTER type
    sin_func_t my_sin = (sin_func_t)sym;

    // 5. Use it
    double result = my_sin(1.0); // ~0.841
    printf("sin(1.0) = %f\n", result);

    // 6. Close it
    dlclose(handle);
    return 0;
}
```

Листинг 9 – Ручная загрузка `libm.so` для вызова `sin`

Примечание

[title=Внимание!] Ошибка в сигнатуре функции при касте `dlsym` (листинг 9, шаг 4) — это тяжёлое **Undefined Behavior**. Если `sin` ожидает `double`, а вы вызовете его с `int`, это почти гарантированно приведёт к падению из-за нарушения соглашения о вызовах (аргументы будут лежать не в тех регистрах, `XMM0` vs `RDI`).

Итоги раздела

- Динамические библиотеки (`.so`) экономят память и позволяют подменять реализации.
- Они должны быть скомпилированы как PIC (`-fPIC`) для относительной адресации.
- PLT и GOT — механизмы, позволяющие вызывать функции, адреса которых неизвестны до запуска.
- ленивое связывание (по умолчанию) разрешает адрес функции при первом вызове через PLT.
- `LD_PRELOAD` позволяет перехватывать вызовы, подгружая свою `.so` первой.
- `dlopen` и `dlsym` позволяют программе вручную загружать плагины (`.so`) во время работы.

5 Freestanding: Программы без `stdlib`

Мы научились делать системный вызов сами. Теперь мы можем полностью отказаться от стандартной библиотеки C/C++.

5.1 `hosted` vs `freestanding`

- **Hosted:** Стандартный режим. Компилятор предполагает наличие ОС и `stdlib`. Доступны `main`, `malloc`, `printf`, `std::vector` и т.д.
- **Freestanding:** Режим, в котором не предполагается наличие `stdlib`. Нельзя использовать `malloc`, I/O, исключения, RTTI (если они требуют поддержки `stdlib`). Это окружение для ядер ОС, драйверов, микроконтроллеров.

Чтобы собрать программу в `freestanding` режиме, используются флаги:

```
$ g++ -ffreestanding -nostdlib my_program.cpp -o my_program
```

5.2 Точка входа `_start`

`main` — это *не* точка входа в программу. Это просто функция, которую вызывает код *инициализации* из `stdlib` (например, `crt0.o`).

Настоящая точка входа, куда ядро Linux передаёт управление, — это метка *_start*. Мы должны определить её сами, обычно на ассемблере.

5.2.1 Состояние при запуске

Когда ядро запускает *_start*, CPU находится в следующем состоянии:

- `RSP` (указатель стека) 16-байтно выровнен.
- `RBP`, по соглашению, должен быть обнулён (`xor rbp, rbp`). Это используется дебаггерами и бэктрейсерами как маркер конца цепочки стековых кадров.
- Стек содержит аргументы и переменные окружения (рис. 3).

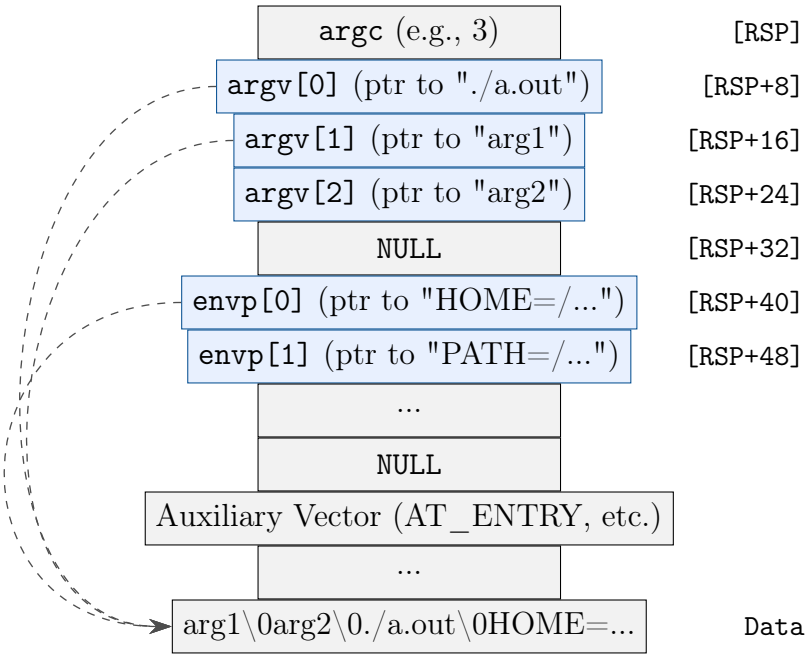

Рис. 3 – Содержимое стека при вызове `_start`

5.2.2 Пример _start

листинг 10 показывает минимальную `freestanding` программу.

```
.section .rodata
msg:
    .string "Hello, freestanding!\n"
msg_end:
    .equ msg_len, msg_end - msg

.section .text
.global _start

_start:
    # Convention: zero out RBP for backtracing
    xor %rbp, %rbp

    # syscall: write(1, msg, msg_len)
    mov $1, %rax # SYS_write
    mov $1, %rdi # fd (stdout)
    mov $msg, %rsi # buf
    mov $msg_len, %rdx # count
    syscall

    # syscall: exit(123)
    mov $60, %rax # SYS_exit
    mov $123, %rdi # exit_code
    syscall
```

Листинг 10 – Freestanding "Hello World"(AT&T синтаксис)

Примечание

Компилятор всё ещё может генерировать вызовы `memcpy`, `memset` и т.д. для оптимизации C++ кода (например, копирования структур). В настоящей `freestanding` среде вам пришлось бы предоставить реализации и этих функций.

5.3 Загрузка и расширение знака

При работе с ассемблером важно помнить о размерах данных. Ошибка из прошлой лекции: чтение 32-битного `int` в 64-битный регистр.

- `mov %eax, [%rsp]` (AT&T): Загружает 32 бита из памяти в `EAX`. *При этом старшие 32 бита* ***RAX*** *обнуляются.*
- `mov %rax, [%rsp]`: Загружает 64 бита.

Проблема: если мы читаем 32-битное отрицательное число (`0xFFFFFFFF`, т.е. -1) с помощью `mov %eax, ...`, `RAX` станет `0x00000000FFFFFFFF` (положительное число $\sim$ 4 млрд).

Определение: Инструкции расширения знака

- movsx / movzx (Move with Sign Extend) / `movslq` (Move Sign-extend Long to Quad): Копирует знаковый бит (старший бит) источника во все старшие биты приёмника.
- `movzx` (Move with Zero Extend) / `movzb/w...`: Заполняет старшие биты приёмника

нулями.

```
# Load a 32-bit dword from [rsp] into 64-bit rax
# with sign extension.
# Intel: movsx rax, dword ptr [rsp]
# AT&T:
movslq (%rsp), %rax
```

Итоги раздела

- `freestanding` режим позволяет писать код без `stdlib`.
- Точка входа в ELF — это *_start*, а не `main`.
- Ядро передаёт `argc`, `argv` и `envp` через стек.
- `_start` должна обнулить `RBP` (`xor %rbp, %rbp`).
- При загрузке 32-битных знаковых чисел в 64-битные регистры необходимо использовать movsx / movzx (`movslq`) для сохранения знака.

6 Введение в архитектуру процессора

Мы научились генерировать инструкции, но почему они выполняются так быстро? Современные CPU — это сложные системы, скрывающие огромную задержку (latency) доступа к памяти.

6.1 Проблема доступа к памяти и кэши

1. **Виртуальная память:** Разыменование указателя (виртуального адреса) требует 4-5 обращений к памяти для прохода по таблицам страниц (Page Tables).
2. **DRAM:** Обращение к основной памяти (DRAM) занимает ~100 наносекунд.

Итого ~500 нс (тысячи тактов CPU) на *каждый* доступ к памяти.

6.1.1 Решение 1: TLB

Translation Lookaside Buffer (буфер ассоциативной трансляции) (TLB) — это маленький, очень быстрый кэш внутри CPU, который хранит недавние отображения "виртуальная страница → физическая страница". Если в TLB есть попадание (hit), CPU избегает 4-5 обращений к памяти.

6.1.2 Решение 2: Иерархия кэшей L1/L2/L3

TLB решает проблему трансляции, но кэш решает проблему медленной DRAM.

- **L1 (32-64KB):** ~1 нс (несколько тактов). Обычно разделён на L1i (инструкции) и L1d (данные).
- **L2 (256KB-4MB):** ~4-12 нс.
- **L3 (8MB+):** ~30-50 нс. (Общий для всех ядер).
- **DRAM:** ~100+ нс.

6.1.3 Организация кэша

- **кэш-линия:** Данные передаются не побайтно, а блоками по 64 байта.
- **ассоциативный кэш:** Кэш организован как хэш-таблица.

Физический адрес разбивается на три части: `[ TAG (36 бит) | SET_INDEX (6-10 бит) | OFFSET (6 бит) ]`

1. **OFFSET (0-5):** Байт внутри 64-байтной кэш-линия.
2. **SET_INDEX (6-11):** Индекс "корзины"(set) в хэш-таблице.
3. **TAG (12-47):** Уникальный идентификатор кэш-линия.

Примечание

Простая хэш-функция (средние биты адреса) — это проблема. Если программа часто обращается к адресам с шагом, кратным большой степени двойки (например, `arr[i * 1024]`), все эти обращения могут попасть в *один и тот же set*, "выбивая"друг друга из кэша, даже если кэш L1 в целом пуст.

6.2 Конвейер инструкций (Pipeline)

Для сокрытия задержек (даже L1) CPU исполняет инструкции в конвейер (например: Fetch → Decode → Execute → Memory → Writeback). Исполнение нескольких инструкций перекрывается во времени.

6.2.1 Конфликты (Hazards)

- **конфликт по данным:** Инструкция N (напр. `add`) ждёт результат инструкции N-1 (напр. `mov`). Пример: итерация по связному списку (`node = node->next`) — это чистый конфликт по данным, т.к. следующий `mov` зависит от предыдущего `mov`.
- **конфликт по управлению:** Условный переход (`je`, `jne`). CPU не знает, какую инструкцию загружать (Fetch) следующей, пока не выполнится (Execute) `cmp`.

6.2.2 Продвинутые оптимизации CPU

1. **Out-of-Order Execution (внеочередное исполнение) (OoOE):** CPU может исполнять инструкции не по порядку, если они не зависят друг от друга, чтобы "заполнить"простои (например, во время конфликт по данным или промаха кэша).
2. **Переименование регистров:** CPU имеет сотни *физических* регистров, но только 16 *архитектурных* (`rax`...). CPU динамически переименовывает `rax` в `phys_reg_5` в одной инструкции и в `phys_reg_28` в другой, чтобы разорвать ложные зависимости по данным.
3. **предсказание ветвлений:** Для решения конфликт по управлению, CPU *угадывает* результат `je` и спекулятивно исполняет код.

6.2.3 Пример: Предсказатель ветвлений

Рассмотрим код (даже на Python) для подсчёта элементов в массиве:

```python
import numpy as np
data = np.random.randint(0, 256, size=100000)
# data.sort() # <--- The key line

count = 0
for x in data:
    if x < 128: # <--- The conditional branch
        count += 1
```

Листинг 11 – Тест предсказателя ветвлений

- **Несортированный массив:** `if x < 128` непредсказуем. Предсказатель ошибается в ~50% случаев.
- **Сортированный массив:** `if` всегда `True` для первой половины, всегда `False` для второй. Предсказатель ошибается *только один раз* (когда `True` меняется на `False`).

Результат: код на сортированном массиве работает *значительно* (в 5-10 раз) быстрее из-за почти 100% точности предсказание ветвлений.

Итоги раздела

- Доступ к DRAM очень медленный (~100 нс).
- TLB кэширует трансляцию виртуальных адресов в физические.
- Кэши L1/L2/L3 кэшируют сами данные из DRAM.
- Данные ходят кэш-линия по 64 байта.
- конвейер перекрывает исполнение инструкций.

- OoOE, переименование регистров и предсказание ветвлений — ключевые техники CPU для сокрытия задержек и решения конфликт по данным и конфликт по управлению.