Среда исполнения

МФТИ
Осень 2024

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Что скрывается за этим кодом?

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Функция `main`, являющаяся точкой входа в пользовательский код

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Аргументы функции `main`: `argc` и `argv`

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Функция `printf`, которая *каким-то* магическим образом *что-то* печатает

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

Весь код это набор закодированных инструкций, находящихся *где-то* в памяти

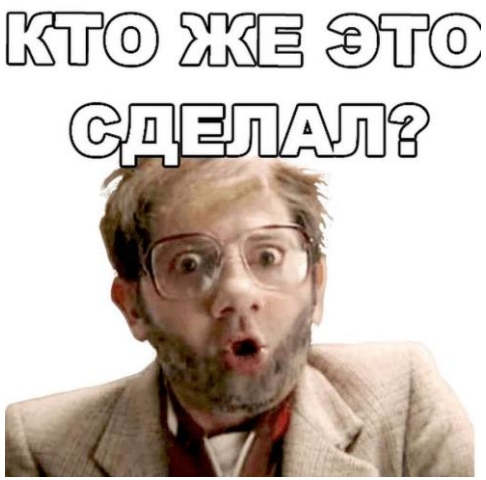
Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
}
```

А еще *где-то* в памяти лежат строковые литералы, про которые мы конечно же не забыли

Рассмотрим пример программы

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“Specified args:\n”);
  for (int i = 0; i < argc; ++i)
    printf(“%s\n”, argv[i]);
  return 0;
}
```



А еще наш любимый компилятор вставляет код возврата *куда-то* из `main`

Обсуждение: что определяет реальность?

- Что из рассмотренного определяется стандартом?

Обсуждение: что определяет реальность?

- Что из рассмотренного определяется стандартом?
  - `main` – точка входа
  - `main` возвращает `int`, `0` по умолчанию
  - Компилятор должен позаботиться о передаче аргументов в `main`
  - Строковые литералы должны лежать в особой памяти

Обсуждение: что определяет реальность?

- Что из рассмотренного определяется стандартом?
  - `main` – точка входа
  - `main` возвращает `int`, `0` по умолчанию
  - Компилятор должен позаботиться о передаче аргументов в `main`
  - Строковые литералы должны лежать в особой памяти
- А кем/чем определяются остальные аспекты?

Обсуждение: что определяет реальность?

- Что из рассмотренного определяется стандартом?
  - `main` – точка входа
  - `main` возвращает `int`, `0` по умолчанию
  - Компилятор должен позаботиться о передаче аргументов в `main`
  - Строковые литералы должны лежать в особой памяти
- А кем/чем определяются остальные аспекты?
- Как определяют границу ответственности?

Что определяет стандарт языка? ISO/IEC 9899:1999

- **5. Environment**
  - An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this International Standard. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.
- C translation environment кажется ± все понятно
  - Source files, (or preprocessing files)
  - Preprocessing translation unit
  - Translation unit
  - Translation phases
  - Diagnostics

Execution environment

- **5.1.2 Execution environments**
  - Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects with static storage duration shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

- Как можем видеть, нам даже подсказывают, что среда исполнения делает *startup*, *инициализирует* статические переменные и забирает управление *в конце выполнения программы*

Freestanding execution environment

- **5.1.2.1 Freestanding environment**
  - In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined. The effect of program termination in a freestanding environment is implementation-defined.
- Freestanding environment это вещь в себе и ~~может работать довольно по-разному~~ зависит от реализации

Hosted execution environment

- **5.1.2.2 Hosted environment**
  - A hosted environment need not be provided, but shall conform to the following specifications if present.
- **5.1.2.2.1 Program startup**
  - The function called at program startup is named `main`. ... It shall be defined with a return type of `int` and with no parameters … or with two parameters (referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared) … or in some other implementation-defined manner.
  - If they are declared, the parameters to the main function shall obey the following constraints: …
- **5.1.2.2.2 Program execution**
  - In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7).
- **5.1.2.2.3 Program termination**
  - If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument; reaching the `}` that terminates the `main` function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

Среда исполнения

- Если есть что-то похожее на операционную систему, то она обеспечивает среду исполнения с немного описанным интерфейсом
- Если операционной системы нет, то средой исполнения может быть почти что угодно, удовлетворяющее небольшому числу требований
- Вообще если почитать стандарт дальше, то видно, что стандарт постоянно описывает какую-то абстрактную машину языка Си, на которой выполнится любая программа на этом языке

Среда исполнения

- Так что же такое среда исполнения в широком смысле?
- **Среда исполнения** — вычислительное окружение, необходимое для выполнения компьютерной программы и доступное во время выполнения компьютерной программы
- Примеры:
  - C/C++ runtime
  - JVM
  - Android runtime
  - V8, NodeJS, Bun (JS engines)
  - Python/Ruby/… interpreters
- Именно среда исполнения определяет *как* мы взаимодействуем с окружающим миром

Пытаемся общаться с миром

Как бы вы реализовали работу с переферией в процессоре?

Пытаемся выводить

Как бы вы это реализовали в процессоре?

- Специальная инструкция
- Memory-mapped IO
- Что-то среднее
  - Сегодня наиболее распространен подход, когда с помощью специальных инструкций передается управление среде исполнения, которая изолирует доступ MMIO

Пытаемся выводить

Как бы вы это реализовали в процессоре?

- Специальная инструкция
- Memory-mapped IO
- Что-то среднее
  - Сегодня наиболее распространен подход, когда с помощью специальных инструкций передается управление среде исполнения, которая изолирует доступ MMIO
  - В RISC-V такой инструкцией является `ecall` (environment call)

ECALL: Environment CALL

- Инструкция `ecall` имеет фиксированную кодировку, ни один из аргументов вызова не кодируется в инструкции
- Как тогда передаются параметры в `ecall`?

ECALL: Environment CALL

- Инструкция `ecall` имеет фиксированную кодировку, ни один из аргументов вызова не кодируется в инструкции
- Как тогда передаются параметры в `ecall`?
- Через регистры по ABI
- Пример linux syscalls для RISC-V:
  - Аргументы – `a0-a5`
  - Номер системного вызова – `a7`
    - `read` – 63
    - write – 64
    - `exit` – 93

ECALL: Environment CALL

- Как вы думаете почему номер системного вызова не кодируется прямо в инструкции?

ECALL: Environment CALL

- Как вы думаете почему номер системного вызова не кодируется прямо в инструкции?
- Потому что декодирование immediate дороже, чем декодирование номера регистра

Передача управления в ОС

- При вызове `ecall` создается прерывание, передающее управление среде исполнения
- То как среда исполнения реализует дальнейший вызов системного вызова – деталь реализации
- Как правило это выглядит в формате знакомой нам таблицы вызовов

Номер системного вызова	Адрес функции, реализующей вызов
63	`0x00001000`
64	`0x00002000`
93	`0x00003000`
430	`0x00004000`
57	`0x00005000`

ECALL: Environment CALL

- Семантически ECALL очень похож на *особенный* вызов *особенной* функции
- После выполнения запрошенного сервисного вызова ECALL или заканчивается выполнение программы (аля `exit`), или управление возвращается на инструкцию следующую после ECALL
- Можно ли сделать такой вызов к среде исполнения, чтобы управление из нее предалось куда-то в третье место?

ECALL: Environment CALL

- Семантически ECALL очень похож на *особенный* вызов *особенной* функции
- После выполнения запрошенного сервисного вызова ECALL или заканчивается выполнение программы (аля `exit`), или управление возвращается на инструкцию следующую после ECALL
- Можно ли сделать такой вызов к среде исполнения, чтобы управление из нее предалось куда-то в третье место?
- Да, так работают отладчики с помощью инструкции EBREAK

EBREAK: Self-Hosted Debug

- Инструкция EBREAK вызывает прерывание, передающее управление отладчику
- В случае ОС и gdb – отладчик должен заранее зарегистрировать в системе, что он «мониторит» процесс и breakpoint будет обрабатывать он
- Такой вариант отладки, когда отладчик запущен на той же машине, что и отлаживаемая программа, называется **self-hosted**
- Каждый раз, когда вы ставите breakpoint, отладчик заменяет инструкцию, на которой необходимо остановиться, инструкцией EBREAK, а при остановке, заменяет ее исходной

Недостаток Self-Hosted Debug

- Представьте, что вам нужно отладить ошибку при загрузке ОС, когда вы еще не можете воспользоваться обычным отладчиком

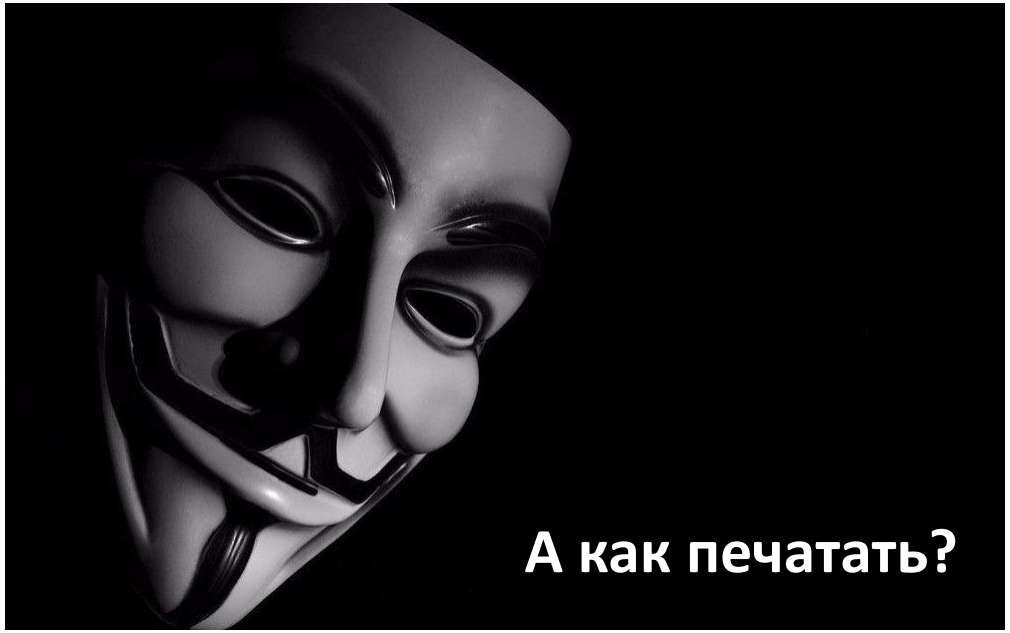
EBREAK: External Debug

- Представьте, что вам нужно отладить ошибку при загрузке ОС, когда вы еще не можете воспользоваться обычным отладчиком
- В таком случае приходят на помощь внешние отладчики
- Внешние отладчики подключаются к специальным *физическим* портам процессора, реализующим интерфейс для протокола отладки
- При загрузке с внешним отладчиком процессор работает в специальном режиме отладки – еще одна среда исполнения
- Инструкция EBREAK в таком режиме останавливает ядро и посылает сигнал внешнему отладчику, который имеет возможность прочитать регистры, память и тд., чтобы принять решение о возобновлении исполнения

Так а как печатать?

`printf` – основа основ

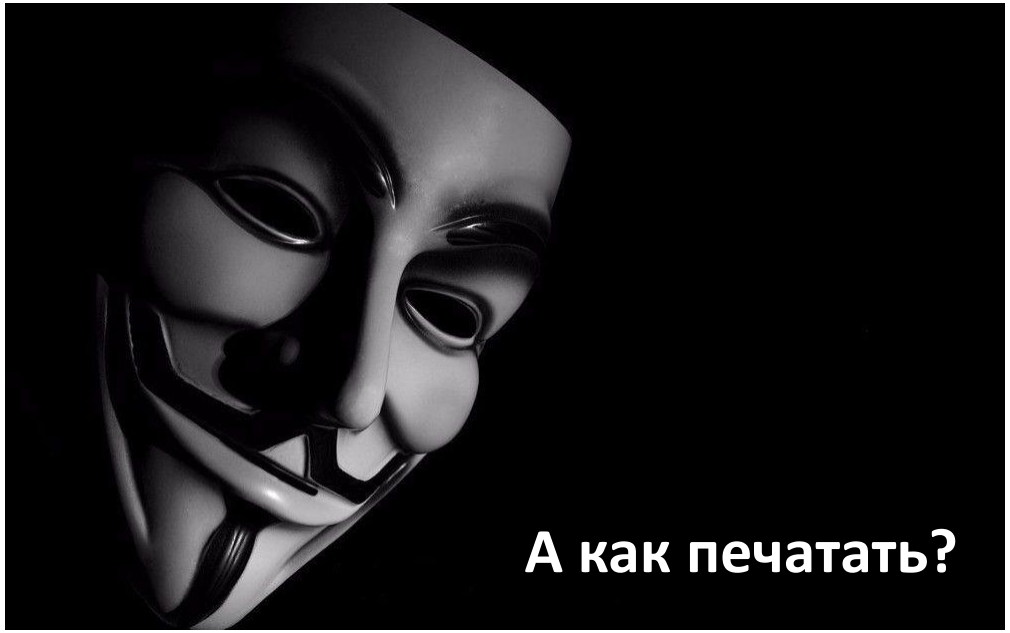
- Что происходит внутри `printf` в среде исполнения с UNIX ОС?



Так а как печатать?

`printf` – основа основ

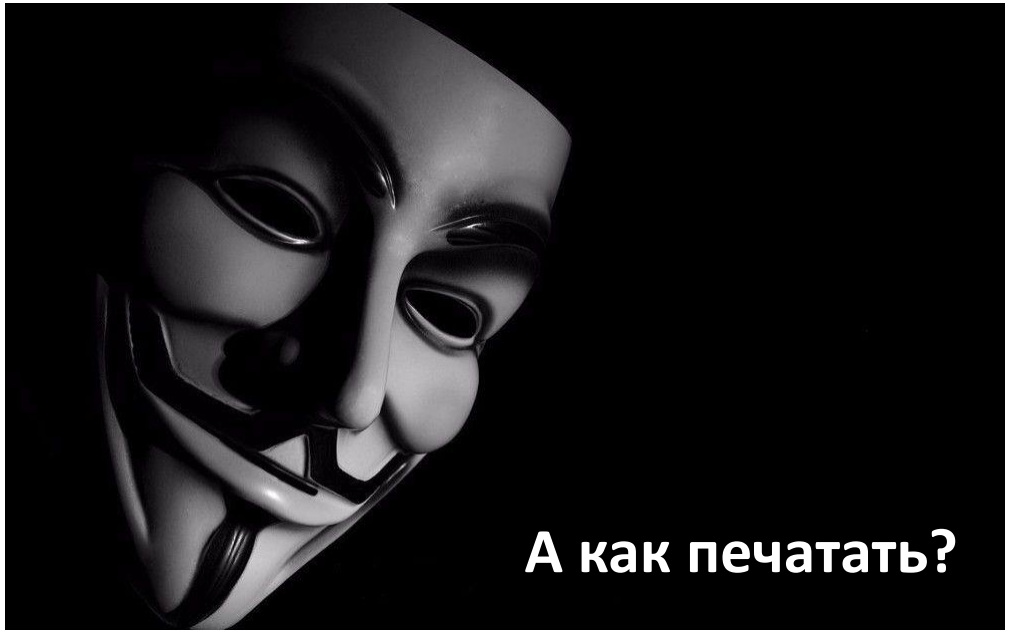
- Что происходит внутри `printf` в среде исполнения с UNIX ОС?
- В UNIX после форматирования там вызывается системный вызов `write`
- А как реализовано тело функции этого системного вызова?



Так а как печатать?

`printf` – основа основ

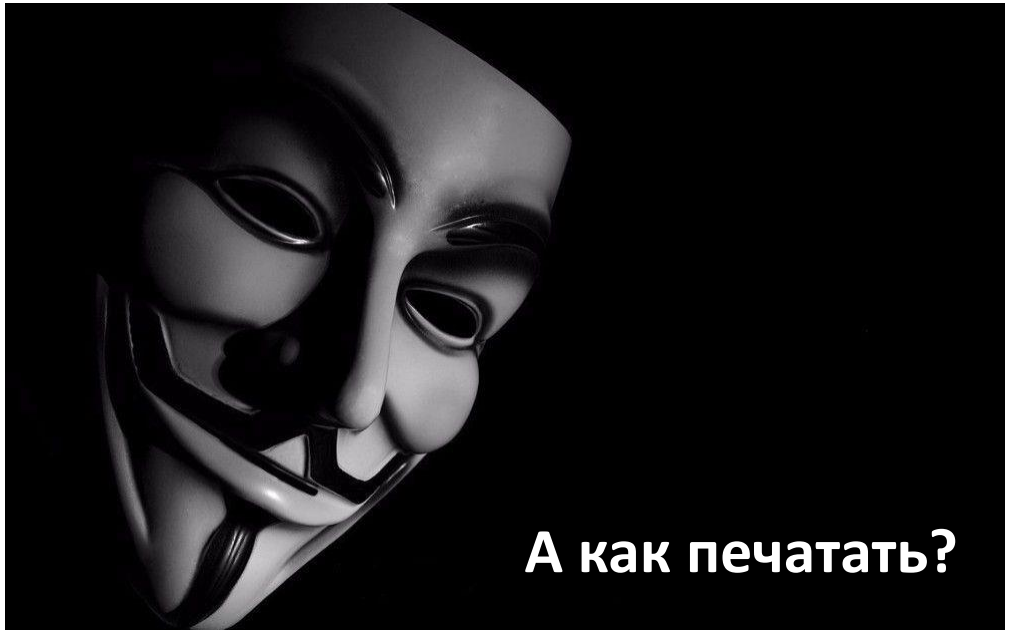
- Что происходит внутри `printf` в среде исполнения с UNIX ОС?
- В UNIX после форматирования там вызывается системный вызов `write`
- Оно состоит из ассемблерной вставки, подготавливающей регистры и вызывающей ECALL



Так а как печатать?

`printf` – основа основ

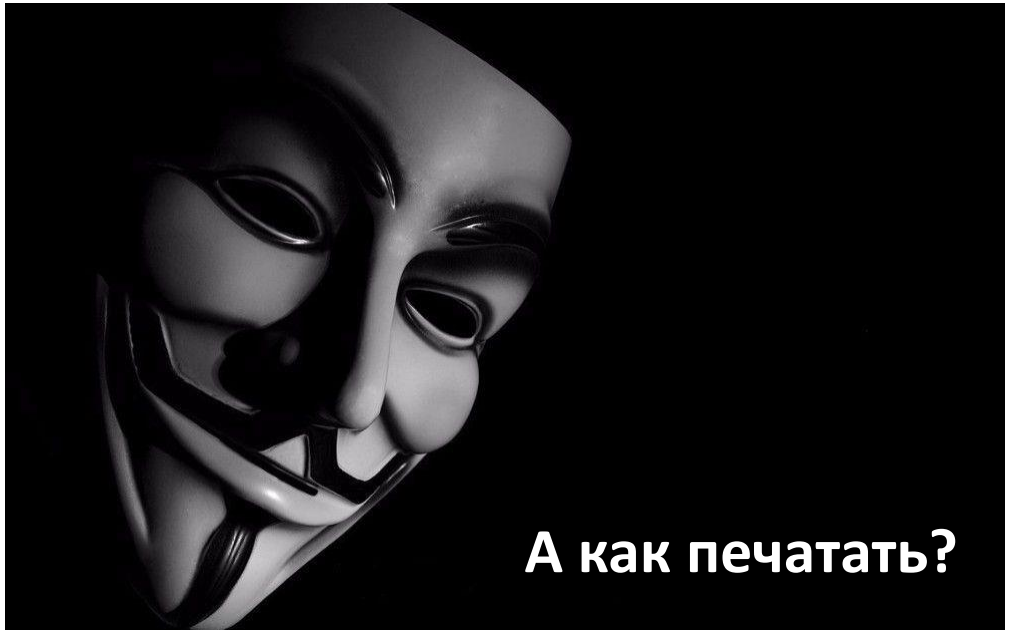
- А как печатать когда ОС нет?



Так а как печатать?

`printf` – основа основ

- А как печатать когда ОС нет?
- На микроконтроллерах у нас нет ни ОС, ни терминала
- Но у нас есть прямой доступ к периферии
- Поэтому `printf` в микроконтроллерах как правило просто печатается в UART



Лирическое отступление

- Как из `write` буковки попадают в терминал?

Лирическое отступление

- Как из `write` буковки попадают в терминал?
- Что такое терминал и почему он так называется?

Терминал DEC VT05

Лирическое отступление

- Исторически терминал это физическое устройство
- Может иметь дисплей, клавиатуру, трекбол и тд.
- Подключался к компьютеру по RS-232 интерфейсу

Терминал DEC VT05

Лирическое отступление

- На раннем этапе развития компьютеров это был основной способ взаимодействия с компьютером
- Такой интерфейс поддержали все основные поставщики ПО
- А потом …

Лирическое отступление

- На раннем этапе развития компьютеров это был основной способ взаимодействия с компьютером
- Такой интерфейс поддержали все основные поставщики ПО
- А потом … просто заменили эмуляторами
- Поэтому когда вы вызываете `write(1, …)`, linux отправляет вашу строку в эмулятор терминала, который вынужден поддерживать legacy абракадабру

Задание: пишем симулятор

Реализуйте симулятор rv32i процессора без привилегированных инструкций

`ecall` – semihosting

`ebreak` – остановка

fence.i – nop

Входной формат: образ памяти и адрес _start

To be continued …

На следующем занятии

- C runtime в деталях
- Привилегированные инструкции

Список литературы

- The RISC-V Instruction Set Manual Volume I Unprivileged Architecture Version 20240411
- Programming languages — C (1999). ISO/IEC 9899:1999