Базовые инструменты разработки

АКОС, МФТИ

12 сентября, 2024

Ваш семинарист:

Артем

Так и запишите.

klimov.aiu@phystech.edu

@prostokvasha

Ваши ассистенты:

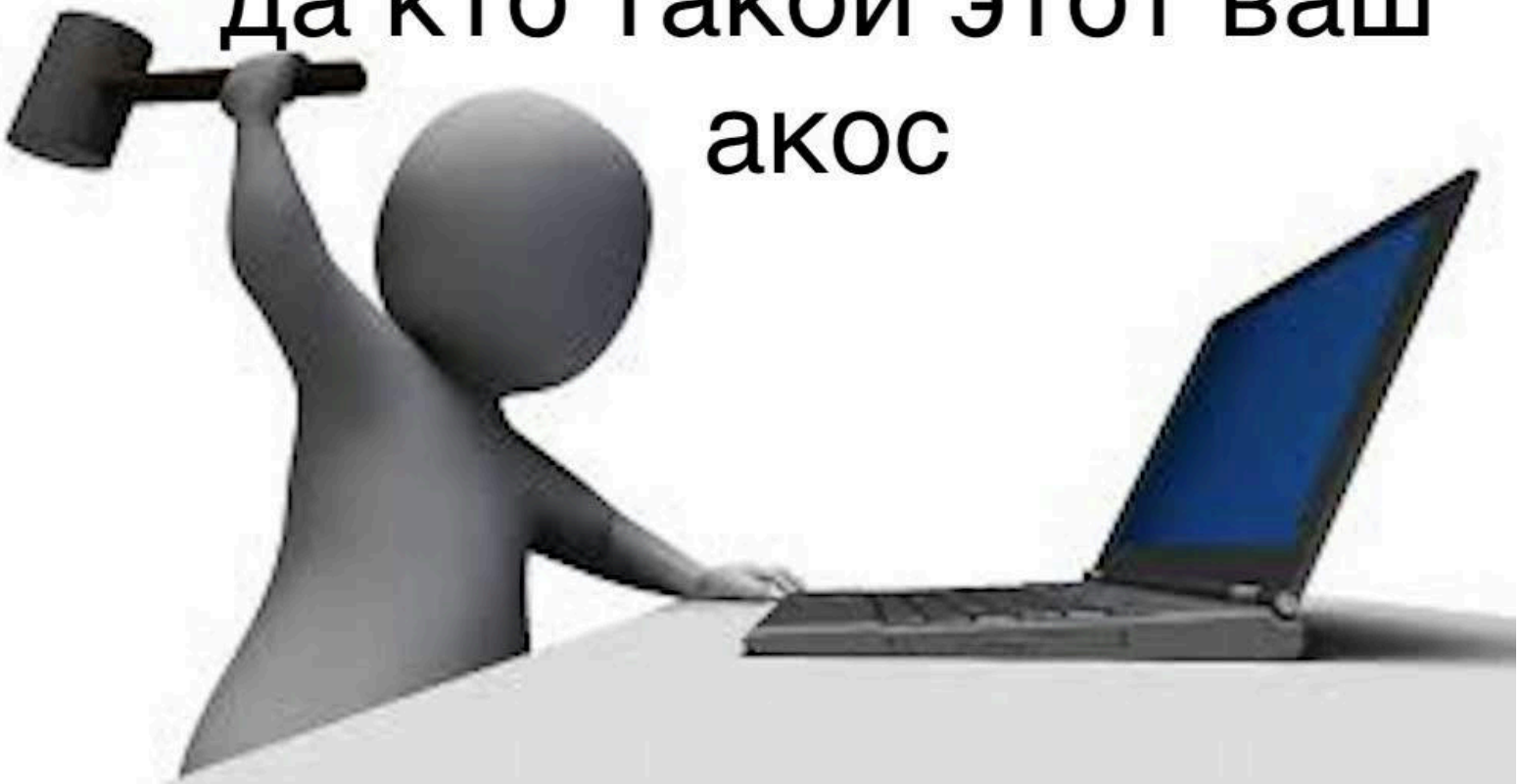
Морозов Артемий Андреевич

@tokreal

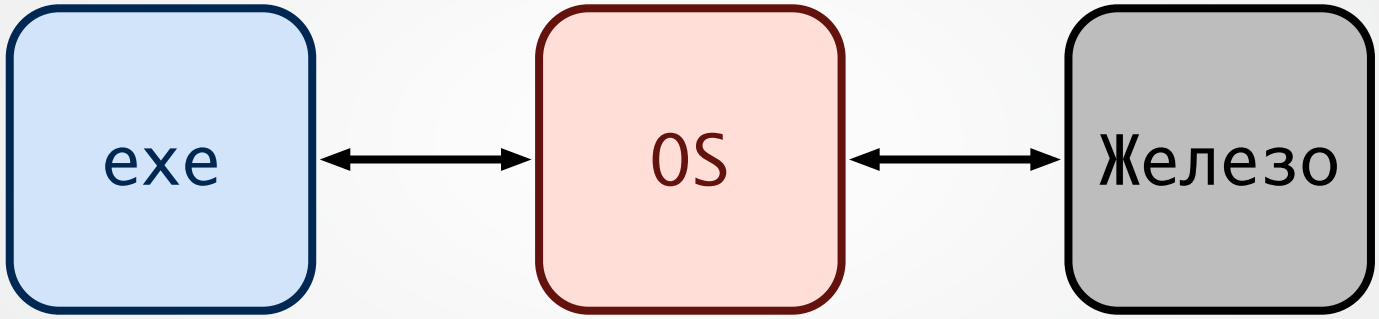
Бояров Алексей Алексеевич

@simpleus

да кто такой этот ваш акос



АКОС поможет вам глубже понимать вот это:



Мотивационный пример

Общие утилиты

- `$ man` : мануалы по чему угодно;
  - `$ man man` : мануалы по мануалам;
- `$ touch` : создать файл;
- `$ mkdir` : создать директорию;
- `$ pwd` : вывести текущую директорию.
- `$ cd` : сменить директорию;
- `$ ls` : вывести содержимое директории.

Работа с файлами

- `$ nano` , `$ micro` , `$ vim` , `$ emacs` : редакторы текста;
- `$ less` : быстрая навигация по файлу;
- `$ cat` : вывести содержимое файла;
- `$ grep` : найти какой-то текст в файле (директории);
- `$ find` : искать файлы по имени / дате создания / ...;
- `$ mv` : переместить / переименовать файл;
- `$ rm` : удалить файл / директорию.

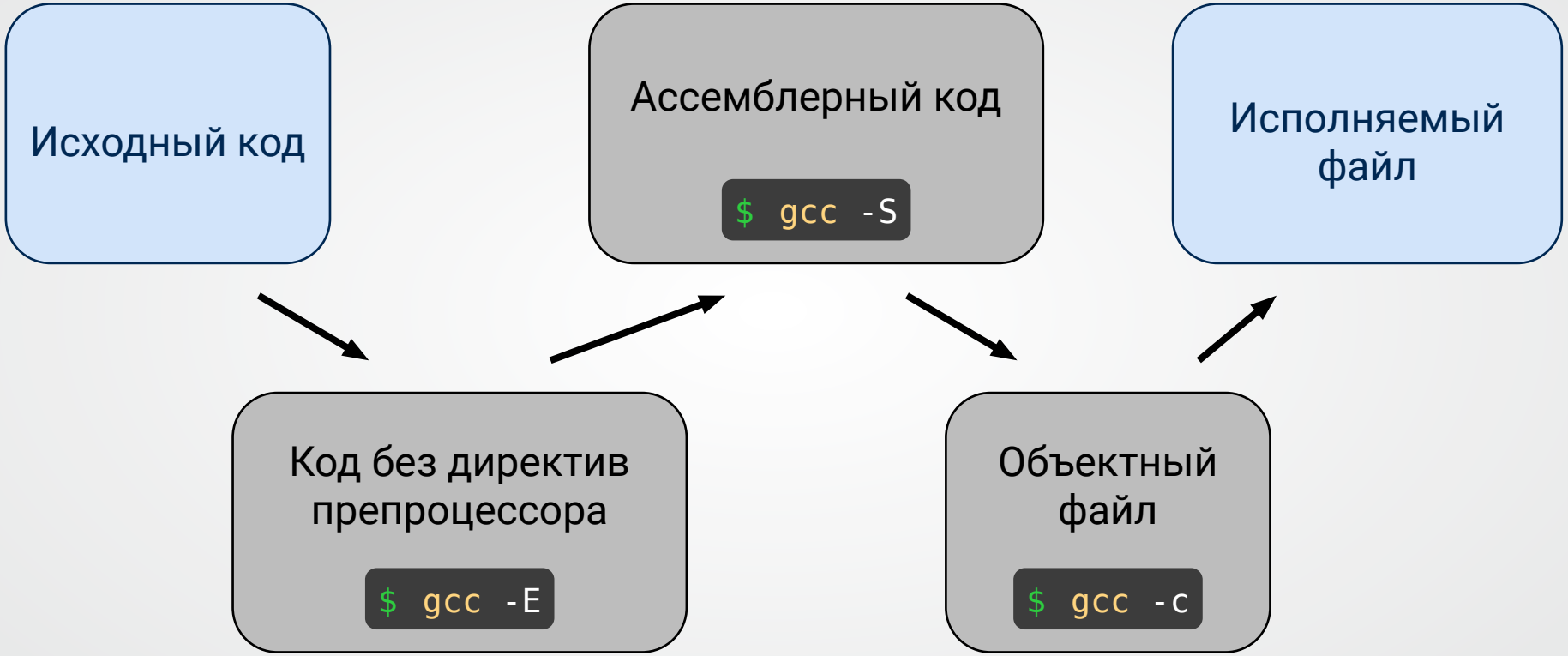
Что делает нас программистами

- `$ gcc` , `$ clang` : компиляторы;
- `$ gdb` , `$ lldb` : отладчики;
- `$ ld` : компоновщик;
- `$ strace` : перехватчик системных вызовов.

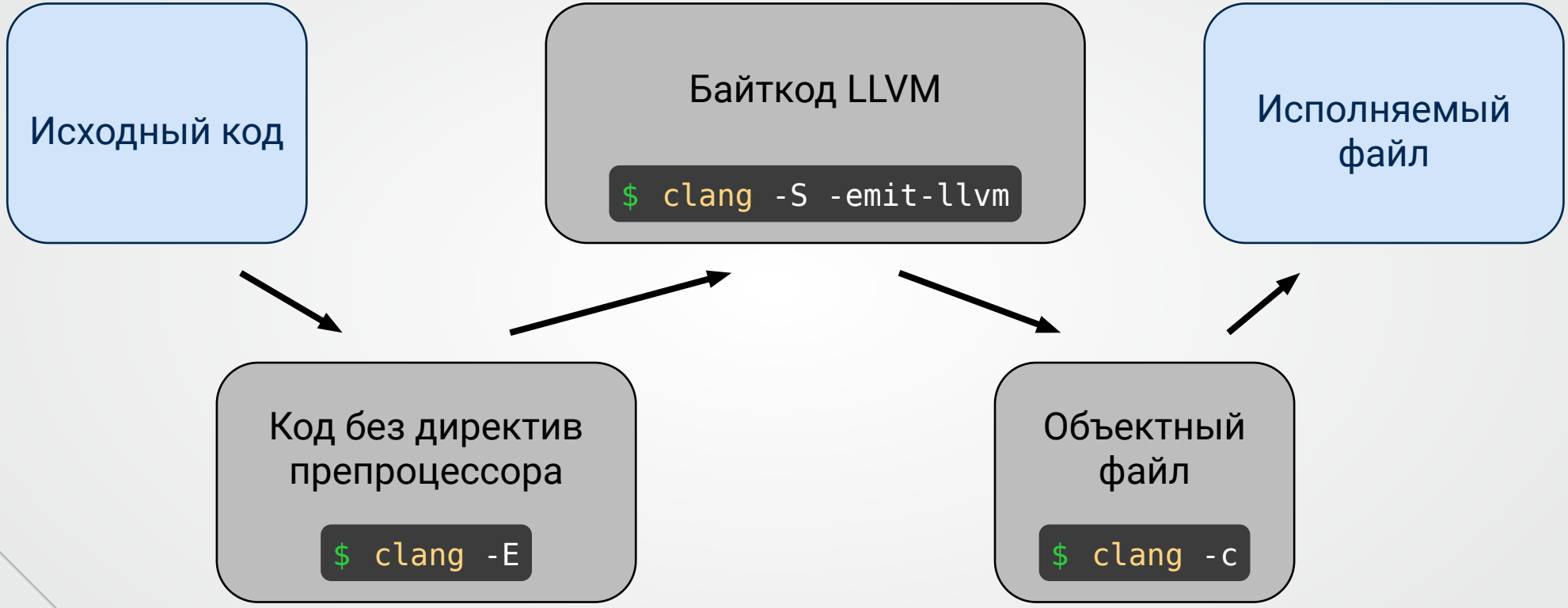
Пользуйтесь консолью!

Это кажется неудобным только первый год.

Как работает GCC



Как работает Clang



Директивы препроцессора

```
1  #define MACRO 42                  // Определение макросов
2  #include "my-header.h"            // Включение других файлов с кодом
3
4  #ifdef WIN32                      // Проверка платформы
5    #error Please, install Linux // Ошибки
6  #endif
7
8  #ifndef DEBUG                      // Условная компиляция
9  void debug_function() { /* noop */ }
10 #else
11 void debug_function() { printf("debug_function called!\n"); }
12 #endif
```

Макросы также можно определять флагами компилятора:

```
$ gcc -D<MACRO_NAME>[=VALUE] ...
```

Хитрости с препроцессором

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

// Макрос для вывода ошибки с названием файла и номером строки:
#define BAIL(message) { \
  printf("%s:%d: Fatal error: %s\n", __FILE__, __LINE__, message); \
  exit(1); }

// Пример использования:
int* allocate_int_array(int n) {
  int* result = (int*)calloc(n, sizeof(int));
  if(!result) BAIL("Cannot allocate memory");
  return result;
}
```

```
main.c:12: Fatal error: Cannot allocate memory
```

Исполняемые файлы

Файл считается **исполняемым**, если он имеет права на исполнение. Linux **не смотрит на расширение**.

Чтобы понять, как запускать файл, Linux смотрит на его начало:

- `0x7f 0x45 0x4c 0x46` : магический заголовок ELF-файла;
- `#!/usr/bin/python3` : shebang, указывает на интерпретатор;
- Ни то, ни другое - файл запускается как шелл-скрипт.

✨ Магические заголовки ✨

Формат	Заголовок
.png	`89 50 4E 47 0D 0A 1A 0A`
.gif	`GIF87a`, `GIF89a`
.jpg, .jpeg	`FF D8 FF`
.rar	`52 61 72 21 1A 07`
.pdf	`25 50 44 46 2D`

🔗 wikipedia.org/wiki/List_of_file_signatures

Автоматизация сборки: `$ make`

- Отслеживает даты изменений файлов;
- Пересобирает то, что устарело, пользуясь явным деревом зависимостей.

Плюсы:

- Пересобирает только то, что нужно;
- Гибкий, не привязан к компилятору, работает с произвольными командами.

Минусы:

- Нужно заморачиваться с зависимостями заголовков;
- …И с кроссплатформенностью;
- Если заморочиться со всем, мейкфайл будет огромным.

Автоматизация сборки: `$ cmake`

- Скриптоподобный язык для генерации схем сборки.

Плюсы:

- Знает особенности разных платформ;
- Удобнее организована работа с библиотеками (vcpkg, pkg-config, …);
- Сам разбирается с зависимостями между заголовками.

Минусы:

- Нужно учить целый отдельный язык;
- Менее гибкий.

Рекомендации по написанию кода

- Code Style может быть любым, главное - **консистентным**;
- `$ clang-format` поможет следить за этим;
- За чем он **не** сможет следить:
  - За названиями переменных;
  - За освобождением памяти;
  - За обработкой ошибок;
  - За структурой кода;
  - За наличием комментариев в нетривиальных местах.
- Умные IDE и `$ clang-tidy` могут помочь и с этим, но лучше следить самим;
- Постарайтесь не пользоваться нейросетями.

Инструменты дебага

 `printf("debug 374\n")` : конечно, способ;

Но куда проще использовать:

 `$ strace` : перехватчик системных вызовов;

 `$ gdb` или `$ lldb` : отладчики для пошагового выполнения программы.

Как пользоваться gdb

- Запуск отладочной консоли: `$ gdb ./a.out`
  - `run` : запустить программу;
  - `break <where>` : поставить точку останова;
  - `next` : выполнить следующую строку;
  - `step` : войти в процедуру;
  - `print <expression>` : вывести значение выражения;
  - `quit` : выйти из gdb.
- Команды можно сокращать: (`r`, `b`, `n`, `p`, `q`);
- 🔗 **GDB cheat-sheet**.

⚠️ Нужно сгенерировать отладочную информацию: `$ gcc -g main.c`

Как пользоваться strace

- Запуск программы с помощью strace: `$ strace ./a.out`
- Основные команды:
  - `$ strace -e trace=open,exec,... ./a.out` : фильтрация системных вызовов;
  - `$ strace -e trace=%file ./a.out` : отслеживать только работу с файлами;
  - `$ strace -p <pid>` : подключиться к уже запущенному процессу;
  - `$ strace -o output.txt ./a.out` : сохранить вывод в файл;
  - `$ strace -c ./a.out` : собрать статистику по системным вызовам;
- 🔗 **Strace docs**

Утечка памяти...

...это потеря указателя на выделенную память:

```
char* buffer = calloc(1024, 1);
read(input, buffer, 1024);
write(output, buffer, 1024);
// free(buffer);
buffer = NULL;
```

Менее очевидный пример:

```
buffer = realloc(buffer, 2048);
```

Утечки файловых дескрипторов

```
int file = open("input.txt", O_RDONLY);
read(file, buffer, 1024);
// close(file)
file = open("output.txt", O_WRONLY);
write(file, buffer, 1024);
```

Открытые дескрипторы занимают память в ядре, так что это тоже утечка памяти.

Чем чреваты утечки?

- **В утилите, которая работает недолго** - ничем, ресурсы освободятся при завершении программы;
- **В долгоживущей программе (веб-сервер, игра, …)** - рано или поздно ресурсы закончатся;
- **В ядре** - вы повторите судьбу CrowdStrike;
- **В домашке по АКОСу** - бан.

Поиск утечек памяти: `$ valgrind`

- Эмулирует процессор и следит за аллокациями;
- Использование: `$ valgrind your-awesome-program`.

Плюсы:

- Находит много других проблем (например, чтение неинициализированной памяти).

Минусы:

- Очень медленный (замедляет в 10-100 раз);
- Многопоточные программы становятся однопоточными.

Поиск утечек памяти: LeakSanitizer

- Санитайзер компилятора `$ clang`
- Встраивает код для отслеживания аллокаций прямо в исполняемый файл;
- Использование: `$ clang -fsanitize=leak`.

Плюсы:

- Почти не замедляет программу;
- Поддерживает многопоточность.

Минусы:

- Доступен только в `$ clang`
- Использует `ptrace()` $\Rightarrow$ не работает под дебаггером, под `$ strace` , и в некоторых контейнерах.

`-fsanitize=`

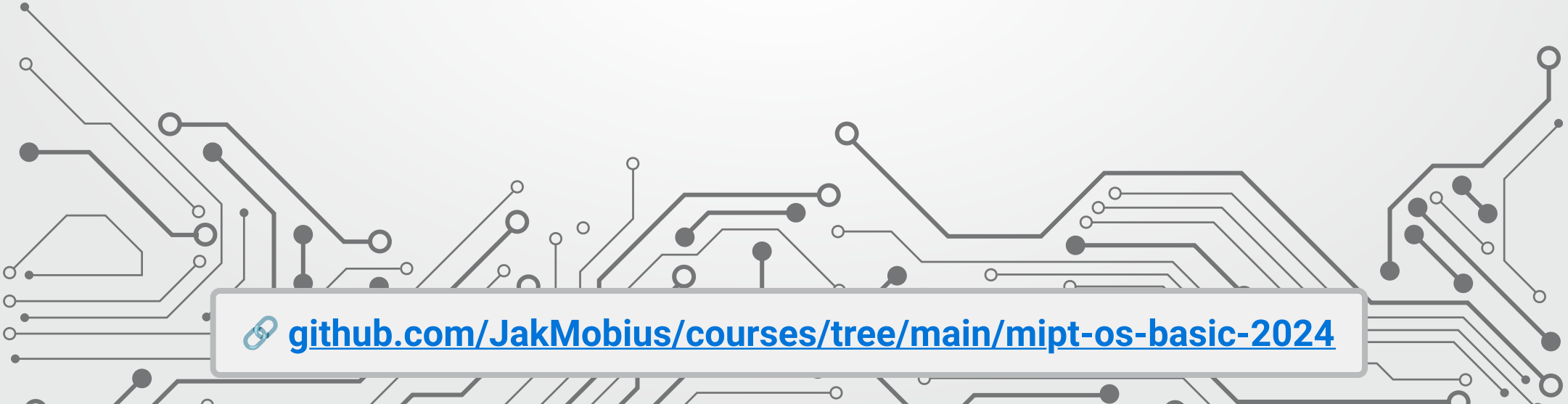
- `address` : поиск ошибок использования памяти (переполнения, use-after-free, …);
- `thread` : поиск гонок;
- `undefined` : поиск неопределённого поведения;
- `memory` : поиск использования неинициализированной памяти;
- `leak` : поиск утечек памяти.

⚠️ : **MemorySanitizer и LeakSanitizer доступны только в `$ clang`.**

⚠️ : **Не все санитайзеры совместимы друг с другом**

Интерактив

Спасибо за внимание!



🔗 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024