Дебаггеры

и немного о том, когда и как нужно заводить детей

Гавриленко Михаил, 1 курс ПИ

- инструмент отладки, который используется для тестирования и отладки других программ)

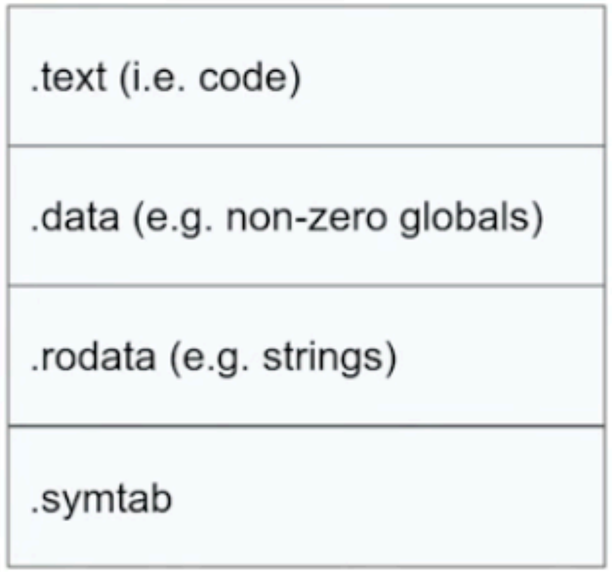
Он может:

- работать с уже запущенным процессом
- сам **запускать** процесс (отлаживать новый)
- проходиться **построчно** по коду
- ставить брейкпоинты
- ***“изучать”*** стек в определенный момент
- ***“изучать”*** переменные
- **изменять** код *“на лету”*
- вызывать функции в адресном пространстве

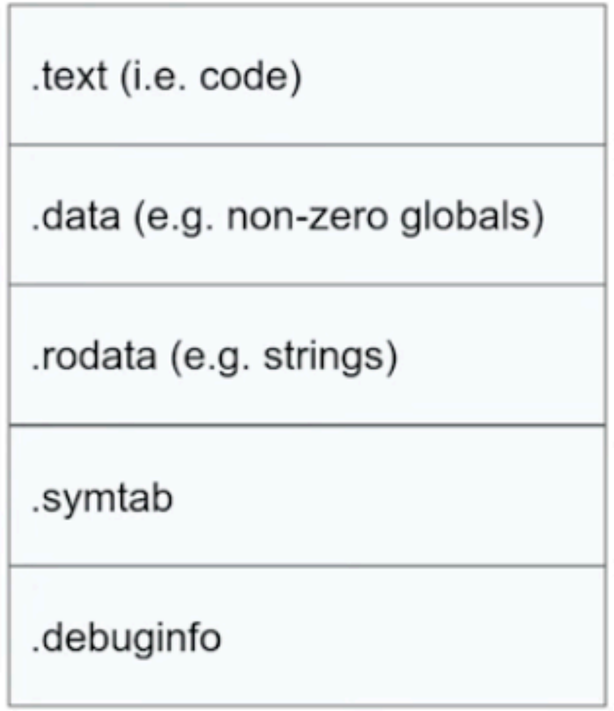
и это далеко не все…

Что значит -g?

```
gcc hello.c      => a.out
```




```
gcc -g hello.c   => a.out
```



Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

https://www.youtube.com/watch?v=0DDrseUomfU

ELF и DWARF

Эльфы внутри дворфов)))

ELF - “Executable Linkable Format”

DWARF - "Debugging With Arbitrary Record Formats”.

* - стандартизованный формат отладочной информации. (чаще всего используется в паре с elf))

Именно DWARF позволяет дебаггеру *понять* исходный код.

Как устроен DWARF?

Файлы **DWARF** разделены на несколько секций (DIE). Наиболее важные секции:

- Compile Unit (CU) - содержит информацию об одном блоке компиляции (о *файле с исх. кодом*)
- Debug Line(Line) - содержит информацию о номерах строк исходного кода, которые соответствуют каждой инструкции в машинном коде.
- Debug Frame (Frame) - содержит информацию о стеке фреймов программы, которые являются стеками вызовов, которые создаются при выполнении программой функций.
- Debug Symbol (Symtab)- содержит информацию о символах в программе, таких как имена функций, переменных и типов.

*DIE - Debug Information Entry

Debug Info Format

$ gcc -g test.c

$ objdump -g a.out >> dwarf_dump.txt

```
Contents of the .debug_info section (loaded from a.out):

  Compilation Unit @ offset 0:
   Length:        0xaa (32-bit)
   Version:       4
   Abbrev Offset: 0
   Pointer Size:  8
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (indirect string, offset: 0): GNU C17 12.3.0 -mlittle-endian
ack-protector-strong -fstack-clash-protection
    <10>   DW_AT_language    : 12       (ANSI C99)
    <11>   DW_AT_name        : (indirect string, offset: 0x11c): test.c
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0xab): /Users/mihailgavrilenko/lim
    <19>   DW_AT_low_pc      : 0x754
    <21>   DW_AT_high_pc     : 0x20
    <29>   DW_AT_stmt_list   : 0
```

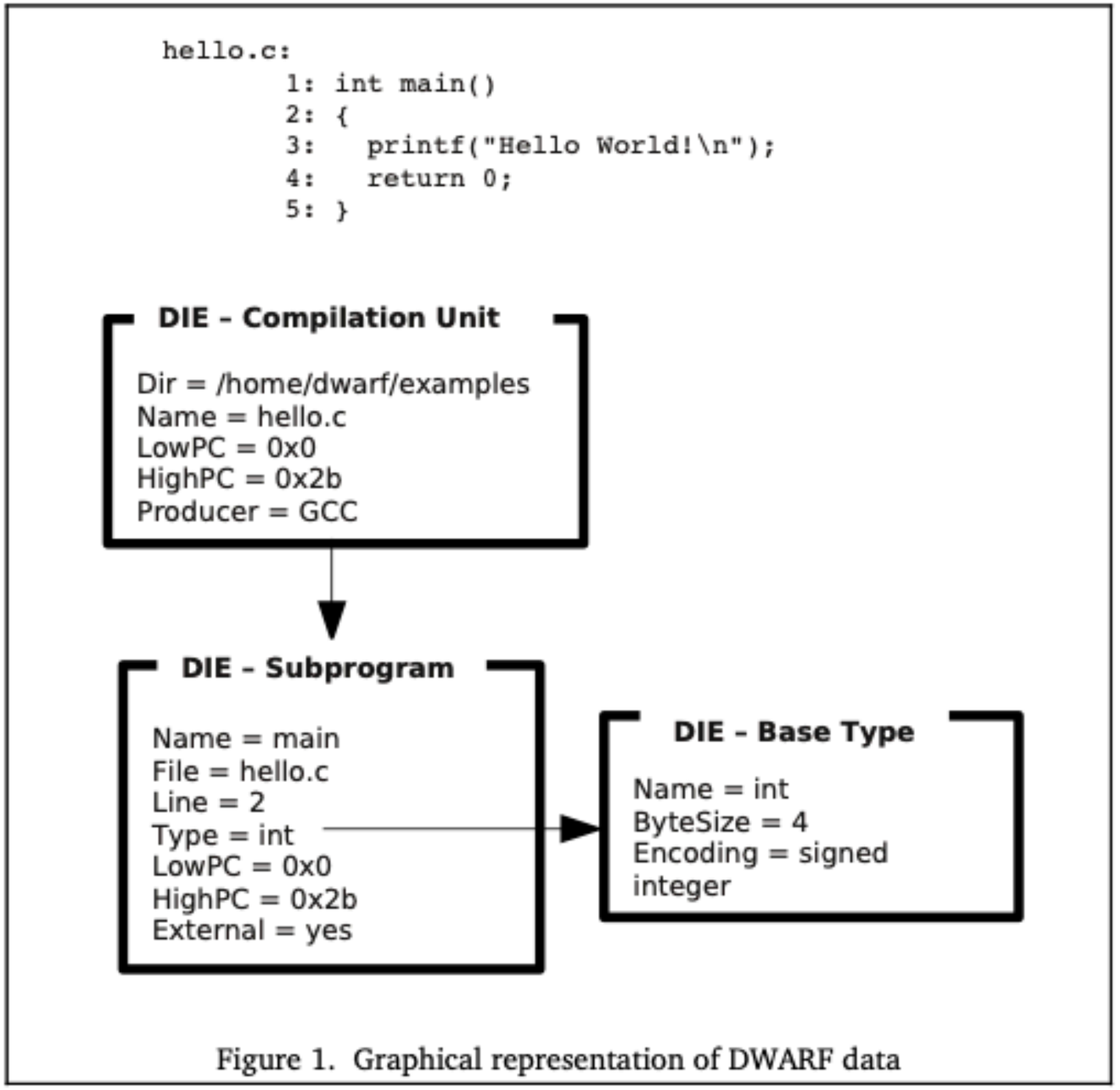


Figure 1. Graphical representation of DWARF data

Ptrace (process trace)

Отец strace

- системный вызов, который предоставляет возможность **родительскому процессу наблюдать**, **контролировать** и **влиять** на выполнение **дочернего процесса**.

ptrace является частью ядра Linux, поэтому он имеет доступ ко всей информации уровня ядра о процессе.

*Отладчик является родителем процесса отладки (или он становится, процессы могут усыновить/удочерить) ребенка :-).

Не путать с strace

strace - system calls and signals trace.

- позволяет отслеживать эти взаимодействия, перехватывая и записывая системные вызовы, сделанные программой, а также любые сигналы, которые она получает.

Что может ptrace?

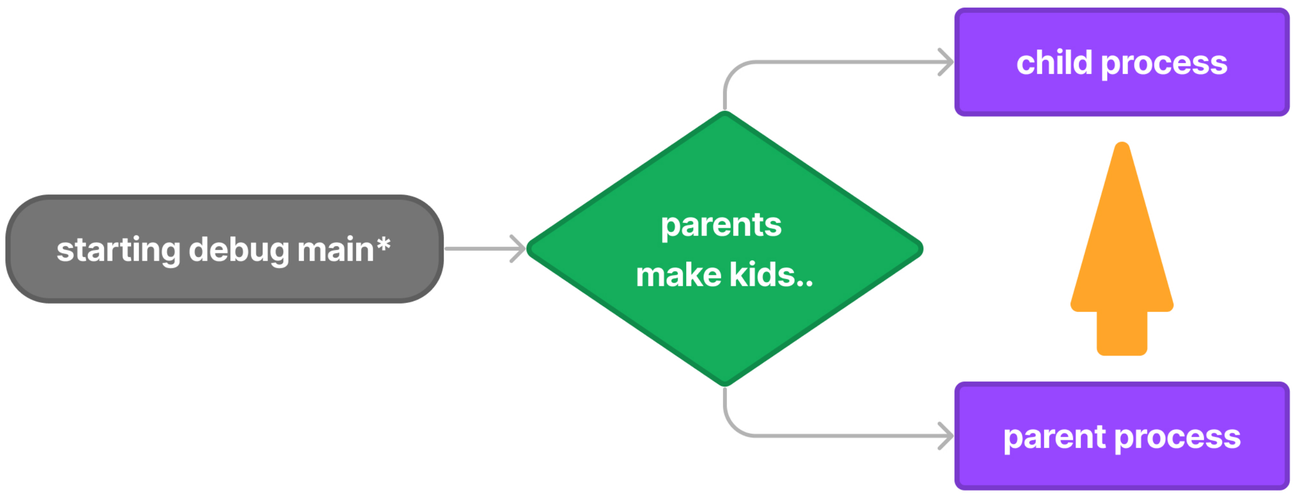
Интерфейс **ptrace** в linux - позволяет дебаггеру воспользоваться низкоуровневой информацией о нашем процессе. Отладчик может:

- читать и записывать память процесса:
- читать и записывать регистры процессора программы:
- быть уведомленным о системных вызовах и распознавать их
- Контролировать ход программы :
- получать и изменять сигналы :

man 2 ptrace

Как это выглядит?

Diagram




```c
void run_debugger(pid_t child_pid)
{
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");

    /* Wait for child to stop on its first instruction */
    wait(&wait_status);

    while (WIFSTOPPED(wait_status)) {
        icounter++;
        /* Make the child execute another instruction */
        if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
            perror("ptrace");
            return;
        }

        /* Wait for child to stop on its next instruction */
        wait(&wait_status);
    }

    procmsg("the child executed %u instructions\n", icounter);
}
```

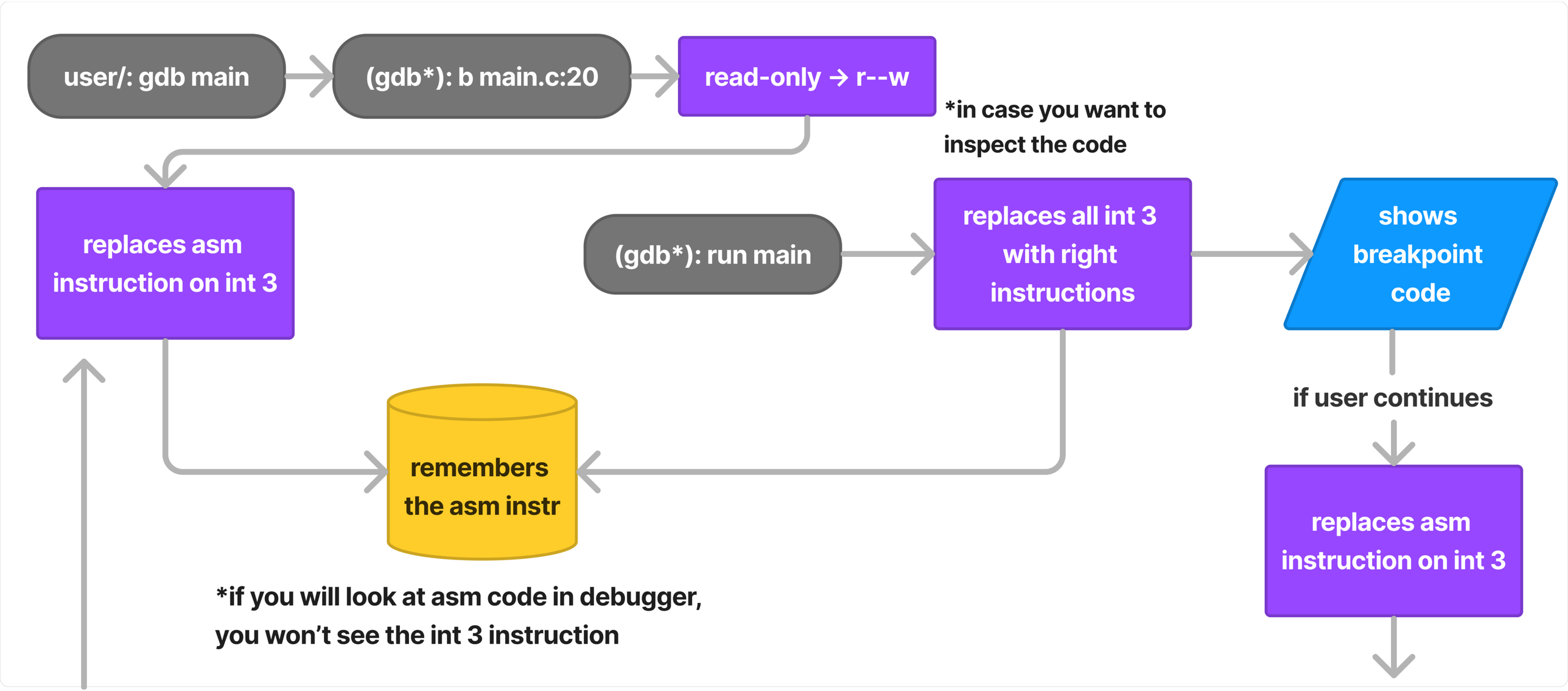
https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1

Breakpoints (software breakpoints)

(gdb) b main.c:X

Основа bp - **int3**.

Инструкция INT3 представляет собой прерывание, которое используется в качестве программной точки остановки. Без наличия отладчика, после получения инструкции INT3, создается исключение EXCEPTION_BREAKPOINT (0x80000003) и вызывается обработчик исключений. Если отладчик присутствует, управление не будет передано обработчику исключений.


user/: gdb main
(gdb*): b main.c:20
read-only → r--w
*in case you want to inspect the code
replaces asm instruction on int 3
(gdb*): run main
replaces all int 3 with right instructions
shows breakpoint code
if user continues
remembers the asm instr
replaces asm instruction on int 3
*if you will look at asm code in debugger, you won't see the int 3 instruction

Магия SIGTRAP

```
dotraplinkage void notrace
do_int3(struct pt_regs *regs, long error_code) {
  //...
  debug_stack_usage_inc();
  cond_local_irq_enable(regs);
  do_trap(X86_TRAP_BP, SIGTRAP, "int3", regs, error_code, NULL);
  cond_local_irq_disable(regs);
  debug_stack_usage_dec();
  //...
}
```

Когда наша программа запущена (run/conitnue), дебаггер находится в ожидании сигнала и ничего не делает.

Посылается сигнал SIGTRAP (Trace/breakpoint trap), после чего дебаггер понимает, что мы остановились перед bp

https://www.youtube.com/watch?v=0DDrseUomfU

Source-level breakpoints

(gdb) break main

```
<1><8f>: Abbrev Number: 9 (DW_TAG_subprogram)
   <90>   DW_AT_external    : 1
   <90>   DW_AT_name        : (indirect string, offset: 0x117): main
   <94>   DW_AT_decl_file   : 1
   <95>   DW_AT_decl_line   : 3
   <96>   DW_AT_decl_column : 5
   <97>   DW_AT_type        : <0x34>
   <9b>   DW_AT_low_pc      : 0x754
   <a3>   DW_AT_high_pc     : 0x20
   <ab>   DW_AT_frame_base  : 1 byte block: 9c        (DW_OP_call_frame_cfa)
   <ad>   DW_AT_GNU_all_tail_call_sites: 1
```

Ставим bp

Inline gdb expressions

(gdb) print my_int_a

```
#include <stdio.h>

int main() {
    int my_int_a = 10;
    int my_int_b = 5;
    printf("a equals %d, b equals %d", my_int_a, my_int_b);
    return 0;
}
```

```
 <2><ae>: Abbrev Number: 2 (DW_TAG_variable)
    <af>   DW_AT_name        : (indirect string, offset: 0): my_int_a
    <b3>   DW_AT_decl_file   : 1
    <b3>   DW_AT_decl_line   : 4
    <b4>   DW_AT_decl_column : 6
    <b4>   DW_AT_type        : <0x35>
    <b8>   DW_AT_location    : 2 byte block: 91 78      (DW_OP_fbreg: -8)
```

offset from stack frame

Stepping

- Instruction step

ptrace(PTRACE_SINGLESTEP, debuggee_pid, nullptr, nullptr);

- Code step

** зачастую все это выполняется благодаря маппингу исходного кода и инструкций, который содержится в dwarf’e.*

**На самом деле line-by-line отладка по коду - лишь улучшенная версия line-by-line отладки по инструкциям*

Спасибо за внимание!

Отлаживайте код по росту оптимизации.

Список источников:

- Linux Debuginfo Formats: DWARF, ELF, dwo, dwp - What are They All? - Greg Law - ACCU 2023
- CppCon 2018: Simon Brand “How C++ Debuggers Work” (must watch)
- “How Does a C Debugger Work?” (Mostly about ptrace)
- Writing a Linux Debugger Part 4: Elves and dwarves
- Introduction to the DWARF Debugging Format
- “Про брэйкпоинты”
- DWARF Specification
- GCC Online Docs

Для особо интересующихся:

- How debuggers work: Part 1
- The Architecture of Open Source Applications (Volume 2) GDB