МИНИСТЕРСТВА НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования «Самарский национальный исследовательский
университет имени академика С.П. Королева»
(Самарский университет)

Институт	информатики и кибернетики
Факультет	информатики
Кафедра	геоинформатики и информационной безопасности

КУРСОВОЙ ПРОЕКТ ПО ДИСЦИПЛИНЕ
"Системное программирование"

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
"Реализация отладчика для компьютерной программы"

Студент _____ В.А. Бобков
(подпись)

Студент _____ И.Д. Правдин
(подпись)

Студент _____ Е.В. Терновский
(подпись)

Студент _____ А.А. Хвацкова
(подпись)

Руководитель работы _____ А.В. Веричев
(подпись)

САМАРА 2022

МИНИСТЕРСТВА НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования «Самарский национальный исследовательский
университет имени академика С.П. Королева»
(Самарский университет)

Институт	информатики и кибернетики
Факультет	информатики
Кафедра	геоинформатики и информационной безопасности

ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ

Студентам Бобкову В.А. группы 6312, Правдину И.Д. группы 6312,
Терновскому Е.В. группы 6312, Хвацковой А.А. группы 6312
Тема проекта: «Реализация отладчика для компьютерной программы»

Планируемые результаты освоения образовательной программы (компетенции)	Планируемые результаты практики	Содержание задания
ОПК-3 - способность применять языки, системы и инструментальные средства программирования в профессиональной деятельности	Уметь использовать системные вызовы при написании программ. Владеть навыками создания программ для операционных систем, реализующих стандарт POSIX; механизм осуществления системных вызовов. Уметь использовать системные вызовы при написании программ. Владеть навыками создания программ для операционных систем, реализующих стандарт POSIX.	1. Изучение методов и подходов к задаче реализации отладчика. 2. Изучение связанных механизмов работы ОС. 3. Программная реализация отладчика. 4. Отладка и тестирование разработанной программы.

Дата выдачи задания 10 февраля 2022 г

Срок представления на кафедру пояснительной записки 9 июня 2022 г.

Руководитель курсового проекта _____ А.В. Веричев
(подпись)
ст. преподаватель каф. ГИиИБ

студент группы № 6312 _____ В.А. Бобков
(подпись)

студент группы № 6312 _____ И.Д. Правдин
(подпись)

студент группы № 6312 _____ Е.В. Терновский
(подпись)

студент группы № 6312 _____ А.А. Хвацкова
(подпись)

РЕФЕРАТ

Пояснительная записка к курсовому проекту: 56 с., 8 рисунков, 7 источников.

РЕАЛИЗАЦИЯ ОТЛАДЧИКА ДЛЯ КОМПЬЮТЕРНОЙ ПРОГРАММЫ

Цель работы: написание отладчика в операционной системе Linux с помощью языка программирования C++ с реализацией таких функций, как запуск, остановка и продолжение выполнения, установка точки останова, чтение и запись регистров и памяти, step методы и вывод исходного кода отлаживаемого приложения.

Оглавление

Введение

Отладчики — один из самых ценных инструментов в наборе любого разработчика. Эта компьютерная программа предназначена для поиска ошибок в других программах и ядрах операционных систем. Отладчик позволяет выполнять трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять точки останова. Отладчики бывают самыми разными, на данный момент существует множество отладчиков, в которых реализуется различный функционал в зависимости от задач, языков программирования и функций, стоящих перед разработчиком. В качестве примеров популярных на данный момент отладчиков можно привести DBX, Dtrace, GNU Debugger и так далее.

Отладчики могут иметь два типа интерфейсов: CLI и GIU.

1. CLI (command line interface) — интерфейс командной строки, при котором отладка выполняется с помощью терминала

2. GIU (Graphic User Interface) графический пользовательский интерфейс, элементы которого выполнены в виде графических изображений. То есть все основные объекты, присутствующие в этом интерфейсе — иконки, функциональные кнопки, объекты меню и т.д. — выполнены в виде изображений.

Стоит отметить, что на данный момент разработка такого инструмента как отладчик крайне актуальна, так как реализация любого кода или программы сильно усложняется без отладки, а нахождение различных типов ошибок в коде может занимать несколько часов или даже дней, если пренебрегать таким полезным и важным инструментом, как отладчик.

Выбранные технологии при разработке приложения

2.1 Язык программирования и операционная система

Для написания отладчика был выбран язык программирования C++. C++ компилируемый, структурированный, объектно-ориентированный, сильно упрощающий работу с большими программами. Компиляторы C++ есть на каждой операционной системе, большинство программ легко переносится с платформы на платформу. В рамках данного курсового проекта главным преимуществом C++ — нативный вызов системных функций типа ptrace.

В качестве платформы приложения был выбран Linux, главным преимуществом которого является системный вызов ptrace (Process Trace). Ptrace предоставляет механизм, с помощью которого родительский процесс может наблюдать и контролировать выполнение другого процесса. Он может проверять и изменять свой основной образ и регистры и используется в основном для реализации отладки точек останова и отслеживания системных вызовов. Так как ptrace — это системный вызов использующийся исключительно в Unix-подобных системах, Linux является оптимальной операционной системой для реализации отладчика.

2.2 Технология отладки

Также для реализации курсового проекта было необходимо выбрать технологию отладки. Для этого необходимо определить какой тип точек останова будет реализован в проекте.

Есть два основных вида точек останова: программные (soft breakponit) и аппаратные (hardware breakpoint). Они ведут себя очень похоже, но выполняются очень разными способами.

1. Программные точки останова

   Перед выполнением программа сначала загружается в память, что позволяет нам временно модифицировать участок памяти, связанный с программой, без влияния на процесс ее выполнения. Именно так и работают программные точки останова. Отладчик запоминает ассемблерную инструкцию, где должна быть вставлена точка останова, затем заменяет ее на ассемблерную инструкцию INT 3 (0xcc), которая заставляет процессор остановить выполнение программы. Как только точка останова достигнута, отладчик считывает текущий адрес памяти, достает ранее записанную инструкцию и показывает ее пользователю. Пользователю кажется, что программа остановился на этой инструкции, однако процессор не имеет ни малейшего представления о ее существовании.

2. Аппаратные точки останова

   Внутри большинства процессоров существуют специальные отладочные регистры, которые можно использовать для хранения адресов точек останова и специальных условий доступа, по которым срабатывают эти точки останова (например, на чтение, запись или выполнение). Точки останова, хранящиеся в таких регистрах, называются аппаратными (или процессорными) точками останова. Когда процессор доходит до адреса памяти, которые определен внутри отладочного регистра и выполняются условия доступа, программа останавливается.

В данном курсовом проекте была выбрана технология программных точек останова.

2.3 Выбор вспомогательных библиотек

2.3.1 Библиотека Linenoise

Библиотека linenoise предназначена для работы в терминале. Она позволяет реализовать автозамену, автодополнение и поддержку истории для ввода команд. Эта библиотека имеет лицензию BSD 2-Clause «Simplified» License.

2.3.2 Библиотека Libelfin

Библиотека Libelfin разработана пользователем aclements. Она предназначена для чтения бинарных файлов ELF и отладочной информации DWARF.

Стандартные средства разработки компилируют программу в файл ELF (Executable and Linkable Format) с возможностью включения отладочной информации. Исполняемый файл формата ELF состоит из таких частей:

- Заголовок (ELF Header)
  Содержит общую информацию о файле и его основные характеристики.

- Заголовок программы (Program Header Table)

  Это таблица соответствия секций файла сегментам памяти, указывает загрузчику, в какую область памяти писать каждую секцию.

- Секции

  Секции содержат всю информацию в файле (программа, данные, отладочная информация и т.д) У каждой секции есть тип, имя и другие параметры. В секции «.text» обычно хранится код, в «.symtab» — таблица символов программы (имена файлов, процедур и переменных), в «.strtab» — таблица строк, в секциях с префиксом «.debug» — отладочная информация и т.д. Кроме того, в файле должна обязательно быть пустая секция с индексом 0.

- Заголовок секций (Section Header Table)

Это таблица, содержащая массив заголовков секций.

DWARF — это стандартизованный формат отладочной информации. Отладочная информация позволяет: устанавливать точки останова (breakpoints) не на физический адрес, а на номер строки в файле исходного кода или на имя функции отображать и изменять значения глобальных и локальных переменных, а также параметров функции отображать стек вызовов (backtrace) исполнять программу пошагово не по одной инструкции ассемблера, а по строкам исходного кода.

Эта информация хранится в виде древовидной структуры. Каждый узел дерева имеет родителя, может иметь потомков и называется DIE (Debugging Information Entry). Каждый узел имеет свой тэг (тип) и список атрибутов (свойств), описывающих узел. Атрибуты могут содержать все, что угодно, например, данные или ссылки на другие узлы. Кроме того, существует информация, хранящаяся вне дерева. Узлы делятся на два основных типа: узлы, описывающие данные, и узлы, описывающие код.

2.4 Дополнительные средства, использованные при разработке:

1. Git и GitHub

   Неотъемлемой частью работы с над проектом является использование системы контроля версий, позволяющей разработчикам делегировать работу и параллельно реализовывать разные её части. В качестве такой системы был выбран Git, а веб-сервисом — GitHub, как один из самых популярных и актуальных на данный момент платформ.

2. Clion

   Для работы на C++ хорошо подходит Clion. CLion — это многофункциональная IDE, которая помогает разработчикам C и C++ сосредоточиться на важных элементах кода благодаря автоматическому выполнению стандартных заданий. CLion поддерживает опции автозавершения

кода, настраиваемые стили программирования, использование карт и различных ракурсов и т. д.

3. Make и Cmake

   Make — это система сборки. Она управляет компилятором и другими инструментами сборки для сборки кода.

   CMake — это генератор сборочных систем. Он может создавать Make файлы, он может создавать файлы сборки Ninja, он может создавать проекты KDEvelop или XCode, он может создавать решения Visual Studio. С той же начальной точки, тот же файл CMakeLists.txt. Поэтому, если есть независимый от платформы проект, CMake — это способ сделать его также независимым от системы.

4. GNU Compiler Collection

   GCC — это свободно доступный оптимизирующий компилятор для языков C, C++. Программа gcc, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, gcc запускает необходимые препроцессоры, компиляторы, линкеры.

Техническое задание на проект

3.1 Требования к проекту

3.1.1 Функционал программы

В рамках данной курсовой работы отладчик будет поддерживать следующие функции:

1. Запуск, остановка и продолжение выполнения
2. Установка точки останова на:
    - Адреса памяти
    - Строки исходного кода
    - Название функции
3. Чтение и запись регистров и памяти
4. Разновидность step методов:
    - Step in
    - Step out
    - Step over
5. Вывод исходного кода отлаживаемого приложения

3.1.2 Взаимодействие с пользователем

Пользователю предоставляется команда -h (-help). Эта команда передаётся как аргумент в командной строке при запуске приложения. Она выводит

краткую информацию по отладчику. Говорит, как корректно его запускать, а так же какие параметры запуска могут использоваться.

3.1.3 Сборка приложения

Правила сборки прописаны в файле CMakeLists.txt В правилах сборки прописаны пути до файлов с исходным кодом. Они используются для создания исполняемого файла. Также в этом файле прописаны правила для сборки и подключения к проекту вспомогательных библиотек.

3.1.4 Размещение исходных кодов

В корне проекта находится файл CMakeLists.txt, в котором прописаны правила сборки разработанного приложения Также там расположены три папки:

- /doc — в этой папке находится документация к проекту
- /lib — в этой папке находится исходный для двух вспомогательных библиотек Linenoise и Libelfin
- /src — в этой папке находится весь исходный код для разработанного отладчика

3.1.5 Требуемые версии средств сборки

- gcc — version 7.5.0
- cmake — version 3.10.2
- make — version 4.1
- стандарт C++ — version ++17

3.1.6 Поддерживаемые форматы отлаживаемых приложений

- gcc — version 7.5.0
- glibc — 2.27
- DWARF — 2.0.0

3.2 Метод реализации технологии отладки

В разработанном приложении используются программные точки останова, так как они проще и не ограничены по количеству. Программные точки останова устанавливаются путем изменения исполняемого кода. Для этого применяется ptrace, который может использоваться для чтения и записи в память. Вносимые изменения должны вызывать остановку процессора и сигнализировать программе, когда выполняется адрес точки останова. На x86-архитектуре это достигается путем перезаписи инструкции по этому адресу инструкцией int 3.

Int (interrupt) — инструкция на языке ассемблера для процессора архитектуры x86, генерирующая программное прерывание.

INT 3 — команда процессоров семейства x86, которая несёт функцию программной точки останова. Исполнение команды приводит к вызову обработчика прерывания номер 3, зарезервированного для отладочных целей. Команда INT 3 кодируется одним байтом с кодом 0xCC.

Когда процессор выполняет инструкцию int 3, управление передается обработчику прерывания точки останова, который сигнализирует процессу SIGTRAP в Linux.

3.2.1 Основные функции

1. **main**

```
int main(int argc, char *argv[]) {
```

```
    ArgParser args(argc, argv);
    if (!args.parse()) {
        return 1;
    }
    auto prog = args.getProgName();
    auto pid = fork();
    if (pid == 0) {
        personality(ADDR_NO_RANDOMIZE);
        execute_debugee(prog);
    } else if (pid >= 1) {
        std::cout << "Started debugging process "
        << pid << '\n';
        debugger dbg{prog, pid};
        dbg.run();
    }
}
```

С функции main начинается выполнение программы. В ней происходит саздание нового процесса с помощью системного вызова fork().это приводит к тому, что программа разделяется на два процесса.

2. **execute_debugee**

```
void execute_debugee(const std::string &prog_name) {
    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
        std::cerr << "Error in ptrace\n";
        return;
    }
    execl(prog_name.c_str(), prog_name.c_str(),
      nullptr);
}
```

В функции execute_debugee происходит вызов ptrace, который указывает, что текущий процесс должен позволить своему родителю отслеживать его.

3. **run**

```
void debugger::run() {
    wait_for_signal();
    initialise_load_address();

    char *line = nullptr;

    while (!end_of_program &&
      (line = linenoise("MEGAdbg> ")) != nullptr) {
            handle_command(line);
            linenoiseHistoryAdd(line);
            linenoiseFree(line);
    }
}
```

В функции происходит ожидание сигнала от родительского процесса и инициализация адреса отлаживаемой программы. Далее запускается цикл, в котором запрашиваются команды у пользователя. В функции происходит обработка пользовательских команд с помощью функции `handle_command`.

4. **continue_execution**

В функции происходит вызов ptrace, чтобы сообщить процессору о продолжении выполнения программы. Затем происходит ожидание сигнала от дочернего процесса.

5. **step_over**

Функция отвечает за шаг с обходом. Концептуально, необходимо поставить точку останова на следующую строку исходного кода. Но это может быть не та строка, если текущее выполнение программы происходит в цикле. По этой причине, необходимо установить точки останова на каждую строку текущей функции. После, продолжить выполнение до последней точки останова и удалить все поставленные точки.

6. **step_out**

   Данная функция предназначена для выполнения команды “finish”. В функции происходит установка точки останова на адрес возврата, переход на установленную точку и ее удаление.

7. **set_breakpoint_at_address**

   Функция устанавливает точку останова по заданному адресу. Для этого создается новая точка, включается и записывается в массив.

8. **set_breakpoint_at_source_line**

   Функция устанавливает точку останова по заданному номеру строки файле с исходным кодом. В функции происходит получение адреса строки и установка точки останова по этому адресу.

9. **set_breakpoint_at_function**

   Функция устанавливает точку останова по заданному названию функции. В функции происходит получение адреса функции и установка точки останова по этому адресу.

Пример работы программы

```
#include "stdio.h"

int a(){
        int a = 1;
        int b = 2;
        return a+b;
}

int b(){
        int a = 10;
        return a;
}

int main(){
        a();
        b();
        return 0;
}
```

Рисунок 4.1 – Исходный код отлаживаемого приложения

```
evgeniy@evgeniy:~/SysProg$ ./my_app a.out
Started debugging Process 1828
Unknown SIGTRAP code 0
MEGAdbg>
```

Рисунок 4.2 – Запуск отладчика

```
MEGAdbg> show
  #include "stdio.h"

  #include "stdio.h"

  int a(){
        int a = 1;
        int b = 2;
        return a+b;
  }

  int b(){
        int a = 10;
        return a;
  }

  int main(){
        a();
        b();
        return 0;
  }

MEGAdbg>
```

Рисунок 4.3 – Вывод исходного кода

```
MEGAdbg> break a
Set breakpoint at address 0x5555555545fe
MEGAdbg>
```

Рисунок 4.4 – Установка точки останова по названию функции

```
MEGAdbg> break a
Set breakpoint at address 0x5555555545fe
MEGAdbg> cont
Hit breakpoint at address 0x5555555545fe

  int a(){
          int a = 1;
          int b = 2;
          return a + b;
  }

MEGAdbg>
```

Рисунок 4.5 – Переход на точку останова

```
...
  int a(){
>         int a = 1;
          int b = 2;
          return a + b;
  }

MEGAdbg> step
  int a(){
          int a = 1;
>         int b = 2;
          return a + b;
  }
```

Рисунок 4.6 – Шаг с заходом

```
...
  int a(){
          int a = 1;
>         int b = 2;
          return a + b;
  }
MEGAdbg> finish
  int main(){
          a();
>         b();
          return 0;
  }

MEGAdbg>
```

Рисунок 4.7 – Шаг с выходом

```
...
  int main(){
          a();
>         b();
          return 0;
  }

MEGAdbg> next
  int main(){
          a();
          b();
>         return 0;
  }

MEGAdbg>
```

Рисунок 4.8 – Шаг без захода

Заключение

Таким образом, при написании программы были получены необходимые представления об устройстве отладчиков. Также получены навыки работы с системными вызовами ptrace, технологией программных точек останова, и различными вспомогательными средствами, такими как Clion, Cmake и GCC. Эти навыки помогут в дальнейшем при написании и отладке любого кода, а также для анализа работы различных, уже существующих, отладчиков, что позволит значительно улучшить качество будущей профессиональной деятельности.

Источники

1. Ptrace: https://www.kernel.org/doc/html/latest/powerpc/ptrace.html
2. ELF: http://www.skyfree.org/linux/references/ELF_Format.pdf
3. DWARF: https://dwarfstd.org/doc/dwarf-2.0.0.pdf
4. Документация к C++: https://devdocs.io/cpp/
5. Документация к Linux: https://www.kernel.org/doc/html/latest/
6. Библиотека libelfin https://github.com/aclements/libelfin
7. Библиотека linenoise https://github.com/antirez/linenoise

Код программы

Код класса debugger

```
#pragma once

#include <utility>
#include <string>
#include <linux/types.h>
#include <unordered_map>

#include <fcntl.h>

#include "breakpoint.h"
#include "utility.h"
#include "libelfin/dwarf/dwarf++.hh"
#include "libelfin/elf/elf++.hh"


class debugger {
public:
    debugger(std::string prog_name, pid_t pid)
            : m_prog_name{std::move(prog_name)}, m_pid{pid
              } {
        auto fd = open(m_prog_name.c_str(), O_RDONLY);

        m_elf = elf::elf{elf::create_mmap_loader(fd)};
        m_dwarf = dwarf::dwarf{dwarf::elf::create_loader(
          m_elf)};
    }

    void run();
```

```
    void set_breakpoint_at_address(std::intptr_t addr, std
        ::string call = "break");

    void set_breakpoint_at_function(const std::string &
        name, std::string call = "break");

    void set_breakpoint_at_source_line(const std::string &
        file, unsigned line);

    void dump_registers();

    void print_source(const std::string &file_name,
        unsigned line, unsigned n_lines_context = 2, std::
        string = "step");

    void show();

    auto lookup_symbol(const std::string &name) -> std::
        vector<symbol>;

    void single_step_instruction();

    void single_step_instruction_with_breakpoint_check();

    void step_in();

    void step_over();

    void step_out();

    void remove_breakpoint(std::intptr_t addr);

private:
```

```
    bool end_of_program = false;

    void handle_command(const std::string &line);

    void continue_execution(std::string call = "break");

    uint64_t get_pc();

    uint64_t get_offset_pc();

    void set_pc(uint64_t pc);

    void step_over_breakpoint();

    void wait_for_signal(std::string call = "break");

    siginfo_t get_signal_info();

    void handle_sigtrap(siginfo_t info, std::string call =
        "break");

    void initialise_load_address();

    uint64_t offset_load_address(uint64_t addr);

    uint64_t offset_dwarf_address(uint64_t addr);

    dwarf::die get_function_from_pc(uint64_t pc);

    dwarf::line_table::iterator get_line_entry_from_pc(
      uint64_t pc);

    uint64_t read_memory(uint64_t address);
```

```
    void write_memory(uint64_t address, uint64_t value);

    std::string m_prog_name;
    pid_t m_pid;
    uint64_t m_load_address = 0;
    std::unordered_map<std::intptr_t, breakpoint>
      m_breakpoints;
    dwarf::dwarf m_dwarf;
    elf::elf m_elf;
};
```

```
#include <cstdint>
#include <sys/wait.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstring>

#include "linenoise/linenoise.h"
#include "utility.h"
#include "debugger.h"
#include "registers.h"


std::vector<symbol> debugger::lookup_symbol(const std::
  string &name) {
    std::vector<symbol> syms;

    for (auto &sec: m_elf.sections()) {
        if (sec.get_hdr().type != elf::sht::symtab && sec.
          get_hdr().type != elf::sht::dynsym)
            continue;
```

```
        for (auto sym: sec.as_symtab()) {
            if (sym.get_name() == name) {
                auto &d = sym.get_data();
                syms.push_back(symbol{to_symbol_type(d.
                    type()), sym.get_name(), d.value});
            }
        }
    }

    return syms;
}

void debugger::initialise_load_address() {
    if (m_elf.get_hdr().type == elf::et::dyn) {
        std::ifstream map("/proc/" + std::to_string(m_pid)
            + "/maps");

        std::string addr;
        std::getline(map, addr, '-');

        m_load_address = std::stol(addr, 0, 16);
    }
}

uint64_t debugger::offset_load_address(uint64_t addr) {
    return addr - m_load_address;
}

uint64_t debugger::offset_dwarf_address(uint64_t addr) {
    return addr + m_load_address;
}
```

```
void debugger::remove_breakpoint(std::intptr_t addr) {
    if (m_breakpoints.at(addr).is_enabled()) {
        m_breakpoints.at(addr).disable();
    }
    m_breakpoints.erase(addr);
}

void debugger::step_out() {
    auto frame_pointer = get_register_value(m_pid, reg::
      rbp);
    auto return_address = read_memory(frame_pointer + 8);

    bool should_remove_breakpoint = false;
    if (!m_breakpoints.count(return_address)) {
        set_breakpoint_at_address(return_address);
        should_remove_breakpoint = true;
    }

    continue_execution();

    if (should_remove_breakpoint) {
        remove_breakpoint(return_address);
    }
}

void debugger::step_in() {
    auto line = get_line_entry_from_pc(get_offset_pc())->
      line;

    while (get_line_entry_from_pc(get_offset_pc())->line
      == line) {
        single_step_instruction_with_breakpoint_check();
    }
```

```
    auto line_entry = get_line_entry_from_pc(get_offset_pc
        ());
    print_source(line_entry->file->path, line_entry->line)
        ;
}

void debugger::step_over() {
    auto func = get_function_from_pc(get_offset_pc());
    auto func_entry = at_low_pc(func);
    auto func_end = at_high_pc(func);

    auto line = get_line_entry_from_pc(func_entry);
    auto start_line = get_line_entry_from_pc(get_offset_pc
        ());

    std::vector<std::intptr_t> to_delete{};

    while (line->address < func_end) {
        auto load_address = offset_dwarf_address(line->
            address);
        if (line->address != start_line->address && !
            m_breakpoints.count(load_address)) {
            set_breakpoint_at_address(load_address, "show
                ");
            to_delete.push_back(load_address);
        }
        ++line;
    }

    auto frame_pointer = get_register_value(m_pid, reg::
        rbp);
    auto return_address = read_memory(frame_pointer + 8);
```

```
    if (!m_breakpoints.count(return_address)) {
        set_breakpoint_at_address(return_address, "show");
        to_delete.push_back(return_address);
    }

    continue_execution("show");

    for (auto addr: to_delete) {
        remove_breakpoint(addr);
    }
}

void debugger::single_step_instruction() {
    ptrace(PTRACE_SINGLESTEP, m_pid, nullptr, nullptr);
    wait_for_signal();
}

void debugger::
  single_step_instruction_with_breakpoint_check() {
    if (m_breakpoints.count(get_pc())) {
        step_over_breakpoint();
    } else {
        single_step_instruction();
    }
}

uint64_t debugger::read_memory(uint64_t address) {
    return ptrace(PTRACE_PEEKDATA, m_pid, address, nullptr
      );
}

void debugger::write_memory(uint64_t address, uint64_t
  value) {
```

```
    ptrace(PTRACE_POKEDATA, m_pid, address, value);
}

uint64_t debugger::get_pc() {
    return get_register_value(m_pid, reg::rip);
}

uint64_t debugger::get_offset_pc() {
    return offset_load_address(get_pc());
}

void debugger::set_pc(uint64_t pc) {
    set_register_value(m_pid, reg::rip, pc);
}

dwarf::die debugger::get_function_from_pc(uint64_t pc) {
    for (auto &cu: m_dwarf.compilation_units()) {
        if (die_pc_range(cu.root()).contains(pc)) {
            for (const auto &die: cu.root()) {
                if (die.tag == dwarf::DW_TAG::subprogram)
                  {
                    if (die_pc_range(die).contains(pc)) {
                        return die;
                    }
                }
            }
        }
    }

    throw std::out_of_range{"Cannot find function"};
}

dwarf::line_table::iterator debugger::
```

```
  get_line_entry_from_pc(uint64_t pc) {
    for (auto &cu: m_dwarf.compilation_units()) {
        if (die_pc_range(cu.root()).contains(pc)) {
            auto &lt = cu.get_line_table();
            auto it = lt.find_address(pc);
            if (it == lt.end()) {
                throw std::out_of_range{"Cannot find line
                    entry"};
            } else {
                return it;
            }
        }
    }

    throw std::out_of_range{"Cannot find line entry"};
}

void debugger::print_source(const std::string &file_name,
  unsigned line, unsigned n_lines_context, std::string
  call) {
    std::ifstream file{file_name};

    auto start_line = line <= n_lines_context ? 1 : line -
        n_lines_context;
    auto end_line = line + n_lines_context + (line <
      n_lines_context ? n_lines_context - line : 0) + 1;

    char c{};
    auto current_line = 1u;

    while (current_line != start_line && file.get(c)) {
        if (c == '\n') {
            ++current_line;
```

```
        }
    }

    if(call != "show") {
        std::cout << (current_line == line ? "> " : "  ");
    }
    else std::cout << "  ";

    while (current_line <= end_line && file.get(c)) {
        std::cout << c;
        if (c == '\n') {
            ++current_line;

            if(call != "show") {
                std::cout << (current_line == line ? "> "
                    : "  ");
            }
            else std::cout << "  ";
        }
    }

    std::cout << std::endl;
}

/*void debugger::show(){
    auto func = get_function_from_pc(get_offset_pc());
    auto func_entry = at_low_pc(func);
    auto func_end = at_high_pc(func);

    auto line_entry = get_line_entry_from_pc(func_entry);
    auto line_end = get_line_entry_from_pc(func_end-3);

    std::ifstream file{line_entry->file->path};
```

```
    auto tmp=line_end->end_sequence;
    auto tmp2=line_end->epilogue_begin;

    char c{};
    auto current_line = 1u;

    while (current_line != line_entry->line && file.get(c)
      ) {
        if (c == '\n') {
            ++current_line;
        }
    }

    std::cout << "   ";

    while (current_line <= line_end->line + 1 && file.get(
      c)) {
        std::cout << c;
        if (c == '\n') {
            ++current_line;
            std::cout << "   ";
        }
    }

    std::cout << std::endl;
}*/

siginfo_t debugger::get_signal_info() {
    siginfo_t info;
    ptrace(PTRACE_GETSIGINFO, m_pid, nullptr, &info);
    return info;
}
```

```
void debugger::step_over_breakpoint() {
    if (m_breakpoints.count(get_pc())) {
        auto &bp = m_breakpoints[get_pc()];
        if (bp.is_enabled()) {
            bp.disable();
            ptrace(PTRACE_SINGLESTEP, m_pid, nullptr,
                nullptr);
            wait_for_signal("show");
            bp.enable();
        }
    }
}

void debugger::wait_for_signal(std::string call) {
    int wait_status;
    auto options = 0;
    waitpid(m_pid, &wait_status, options);

    auto siginfo = get_signal_info();

    switch (siginfo.si_signo) {
        case SIGTRAP:
            handle_sigtrap(siginfo, call);
            break;
        case SIGSEGV:
            std::cout << "Yay, segfault. Reason: " <<
                siginfo.si_code << std::endl;
            break;
        default:
            end_of_program = true;
            std::cout << "Got signal " << strsignal(
                siginfo.si_signo) << std::endl;
    }
```

```
}

void debugger::handle_sigtrap(siginfo_t info, std::string
   call) {
    switch (info.si_code) {
        case SI_KERNEL:
        case TRAP_BRKPT: {
            set_pc(get_pc() - 1);
            if(call != "show" && call != "initial"){
                std::cout << "Hit breakpoint at address 0x
                   " << std::hex << get_pc() << std::endl;
            }
            auto offset_pc = offset_load_address(get_pc())
               ; //rember to offset the pc for querying
              DWARF
            try{
                auto line_entry = get_line_entry_from_pc(
                   offset_pc);
                if (call != "initial"){
                    print_source(line_entry->file->path,
                       line_entry->line);
                }
            }
            catch(std::out_of_range e){
                std::cout << "End of program" << std::endl
                   ;
                end_of_program = true;
                return;
            }
            return;
        }
        case TRAP_TRACE:
            return;
```

```
        default:
            std::cout << "Unknown SIGTRAP code " << info.
                si_code << std::endl;
            return;
    }
}

void debugger::continue_execution(std::string call) {
    step_over_breakpoint();
    ptrace(PTRACE_CONT, m_pid, nullptr, nullptr);
    wait_for_signal(call);
}

void debugger::dump_registers() {
    for (const auto &rd: g_register_descriptors) {
        std::cout << rd.name << " 0x"
                  << std::setfill('0') << std::setw(16) <<
                      std::hex << get_register_value(m_pid,
                      rd.r) << std::endl;
    }
}

void debugger::handle_command(const std::string &line) {
    auto args = split(line, ' ');
    auto command = args[0];

    if (is_prefix(command, "cont")) {
        continue_execution();
    } else if (is_prefix(command, "break")) {
        if (args[1][0] == '0' && args[1][1] == 'x') {
            std::string addr{args[1], 2};
            set_breakpoint_at_address(std::stol(addr, 0,
                16));
```

```
        } else if (args[1].find(':') != std::string::npos)
            {
            auto file_and_line = split(args[1], ':');
            set_breakpoint_at_source_line(file_and_line
                [0], std::stol(file_and_line[1]));
        } else {
            set_breakpoint_at_function(args[1]);
        }
    } else if (is_prefix(command, "step")) {
        step_in();
    } else if (is_prefix(command, "next")) {
        step_over();
    } else if (is_prefix(command, "finish")) {
        step_out();
    } else if (is_prefix(command, "register")) {
        if (is_prefix(args[1], "dump")) {
            dump_registers();
        } else if (is_prefix(args[1], "read")) {
            std::cout << get_register_value(m_pid,
                get_register_from_name(args[2])) << std::
                endl;
        } else if (is_prefix(args[1], "write")) {
            std::string val{args[3], 2}; //assume 0xVAL
            set_register_value(m_pid,
                get_register_from_name(args[2]), std::stol(
                val, 0, 16));
        }
    } else if (is_prefix(command, "memory")) {
        std::string addr{args[2], 2}; //assume 0xADDRESS

        if (is_prefix(args[1], "read")) {
            std::cout << std::hex << read_memory(std::stol
                (addr, 0, 16)) << std::endl;
```

```
        }
        if (is_prefix(args[1], "write")) {
            std::string val{args[3], 2}; //assume 0xVAL
            write_memory(std::stol(addr, 0, 16), std::stol
                (val, 0, 16));
        }
    } else if (is_prefix(command, "symbol")) {
        auto syms = lookup_symbol(args[1]);
        for (auto &&s: syms) {
            std::cout << s.name << ' ' << to_string(s.type
                ) << " 0x" << std::hex << s.addr << std::
                endl;
        }
    } else if (is_prefix(command, "show")) {
        if (m_breakpoints.size() == 0) {
            set_breakpoint_at_function("main", "show");
            continue_execution("initial");
        }
        auto func = get_function_from_pc(get_offset_pc());
        auto func_entry = at_low_pc(func);
        auto func_end = at_high_pc(func);

        auto line_entry = get_line_entry_from_pc(
            func_entry);
        auto line_end = get_line_entry_from_pc(func_end -
            3);
        print_source(line_entry->file->path, line_entry->
            line, line_end->line, "show");
        auto &bp = m_breakpoints[get_pc()];
        bp.disable();
    } else {
        std::cerr << "Unknown command\n";
    }
```

```
}

void debugger::set_breakpoint_at_function(const std::
    string &name, std::string call) {
    for (const auto &cu: m_dwarf.compilation_units()) {
        for (const auto &die: cu.root()) {
            if (die.has(dwarf::DW_AT::name) && at_name(die
                ) == name) {
                auto low_pc = at_low_pc(die);
                auto entry = get_line_entry_from_pc(low_pc
                    );
                ++entry; //skip prologue
                set_breakpoint_at_address(
                    offset_dwarf_address(entry->address),
                    call);
            }
        }
    }
}

void debugger::set_breakpoint_at_source_line(const std::
    string &file, unsigned line) {
    for (const auto &cu: m_dwarf.compilation_units()) {
        if (is_suffix(file, at_name(cu.root()))) {
            const auto &lt = cu.get_line_table();

            for (const auto &entry: lt) {
                if (entry.is_stmt && entry.line == line) {
                    set_breakpoint_at_address(
                        offset_dwarf_address(entry.address))
                        ;
                    return;
                }
```

```
            }
        }
    }
}

void debugger::set_breakpoint_at_address(std::intptr_t
  addr, std::string call) {
    if (call != "show"){
        std::cout << "Set breakpoint at address 0x" << std
           ::hex << addr << std::endl;
    }
    breakpoint bp{m_pid, addr};
    bp.enable();
    m_breakpoints[addr] = bp;
}

void debugger::run() {
    wait_for_signal();
    initialise_load_address();

    char *line = nullptr;

    while (!end_of_program && (line = linenoise("MEGAdbg>
       ")) != nullptr) {
        handle_command(line);
        linenoiseHistoryAdd(line);
        linenoiseFree(line);
    }
}
```

Код класса breakpoint

```
#pragma once
```

```
#include <cstdint>
#include <sys/ptrace.h>
#include <aio.h>

class breakpoint {
public:
    breakpoint() = default;

    breakpoint(pid_t pid, std::intptr_t addr) : m_pid{pid
      }, m_addr{addr}, m_enabled{false}, m_saved_data{} {}

    void enable();

    void disable();

    bool is_enabled() const;

    std::intptr_t get_address() const;

private:
    pid_t m_pid;
    std::intptr_t m_addr;
    bool m_enabled;
    uint8_t m_saved_data; //data which used to be at the
      breakpoint address
};
```

```
#include "breakpoint.h"

void breakpoint::enable() {
    auto data = ptrace(PTRACE_PEEKDATA, m_pid, m_addr,
      nullptr);
    m_saved_data = static_cast<uint8_t>(data & 0xff); //
```

```
        save bottom byte
    uint64_t int3 = 0xcc;
    uint64_t data_with_int3 = ((data & ~0xff) | int3); //
        set bottom byte to 0xcc
    ptrace(PTRACE_POKEDATA, m_pid, m_addr, data_with_int3)
        ;

    m_enabled = true;
}

void breakpoint::disable() {
    auto data = ptrace(PTRACE_PEEKDATA, m_pid, m_addr,
        nullptr);
    auto restored_data = ((data & ~0xff) | m_saved_data);
    ptrace(PTRACE_POKEDATA, m_pid, m_addr, restored_data);

    m_enabled = false;
}

bool breakpoint::is_enabled() const { return m_enabled; }

std::intptr_t breakpoint::get_address() const { return
    m_addr; }
```

Код класса ArgParser

```
#include <iostream>

using namespace std;

class ArgParser {

    const char *opts = "hp:";
    string _progName; //name_prog
```

```
    int _argc;
    char **_argv;

    void help();

    bool fileExist();

public:
    ArgParser(int argc, char *argv[]) : _argc(argc), _argv
      (argv) {}

    bool parse();

    string getProgName();
};
```

```
#include "parser.h"
#include <string.h>
#include <unistd.h>
#include <fstream>

using namespace std;




string ArgParser::getProgName() {
    return _progName;
}

bool ArgParser::fileExist() {
    fstream fileStream;
    fileStream.open(_progName);
    if (fileStream.fail()) {
```

```
        cout << "File does not exist" << endl;
        return false;
    }
    return true;
}

bool ArgParser::parse() {
    int opt = 0;
    string ProgName = string(_argv[1]);
    if (ProgName.find("-") != 0) {
        _progName = ProgName;
    }
    while ((opt = getopt(_argc, _argv, opts)) != -1) {
        switch (opt) {
            case 'h':
                help();
                return false;
            case 'p':
                if (!_progName.empty()) {
                    cout << "Incorrect args" << endl;
                    return false;
                }
                _progName = string(optarg);
                break;
            default:
                help();
                return false;
        }
    }
    if (_progName.empty() || !fileExist()) {
        return false;
    }
    return true;
```

```
}

void ArgParser::help() {
    cout << "This is the debugger.  Usage:" << endl <<
      endl <<
           "    ./my_app [options] [executable-file]" <<
             endl << endl <<
           "Selection of debuggee:" << endl << endl <<
           "   -h              Print this message and then
              exit." << endl <<
           "   -p              Option requires an argument
             "<< endl;
}
```

Код utility

```
#pragma once

#include <cstdint>
#include <sys/personality.h>
#include <unistd.h>
#include <vector>

#include "registers.h"
#include "libelfin/elf/data.hh"

enum class symbol_type {
    notype,             // No type (e.g., absolute symbol)
    object,             // Data object
    func,               // Function entry point
    section,            // Symbol is associated with a
      section
    file,               // Source file associated with the
};
```

```
std::string to_string(symbol_type st);

symbol_type to_symbol_type(elf::stt sym);

struct symbol {
    symbol_type type;
    std::string name;
    std::uintptr_t addr;
};

uint64_t get_register_value(pid_t pid, reg r);

void set_register_value(pid_t pid, reg r, uint64_t value);

uint64_t get_register_value_from_dwarf_register(pid_t pid,
    unsigned regnum);

std::string get_register_name(reg r);

reg get_register_from_name(const std::string &name);

std::vector<std::string> split(const std::string &s, char
   delimiter);

bool is_prefix(const std::string &s, const std::string &of
   );

bool is_suffix(const std::string &s, const std::string &of
   );

void print_source(const std::string &file_name, unsigned
   line, unsigned n_lines_context = 2);
```

```
#include <cstdint>
#include <vector>
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>

#include "utility.h"
#include "registers.h"

uint64_t get_register_value(pid_t pid, reg r) {
    user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, nullptr, &regs);
    auto it = std::find_if(begin(g_register_descriptors),
      end(g_register_descriptors),
                           [r](auto &&rd) { return rd.r ==
                               r; });

    return *(reinterpret_cast<uint64_t *>(&regs) + (it -
      begin(g_register_descriptors)));
}

void set_register_value(pid_t pid, reg r, uint64_t value)
  {
    user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, nullptr, &regs);
    auto it = std::find_if(begin(g_register_descriptors),
      end(g_register_descriptors),
                           [r](auto &&rd) { return rd.r ==
                               r; });
```

```
    *(reinterpret_cast<uint64_t *>(&regs) + (it - begin(
      g_register_descriptors))) = value;
    ptrace(PTRACE_SETREGS, pid, nullptr, &regs);
}

uint64_t get_register_value_from_dwarf_register(pid_t pid,
    unsigned regnum) {
    auto it = std::find_if(begin(g_register_descriptors),
      end(g_register_descriptors),
                           [regnum](auto &&rd) { return rd
                             .dwarf_r == regnum; });
    if (it == end(g_register_descriptors)) {
        throw std::out_of_range{"Unknown dwarf register"};
    }

    return ::get_register_value(pid, it->r);
}

std::string get_register_name(reg r) {
    auto it = std::find_if(begin(g_register_descriptors),
      end(g_register_descriptors),
                           [r](auto &&rd) { return rd.r ==
                             r; });
    return it->name;
}

reg get_register_from_name(const std::string &name) {
    auto it = std::find_if(begin(g_register_descriptors),
      end(g_register_descriptors),
                           [name](auto &&rd) { return rd.
                             name == name; });
    return it->r;
}
```

```
std::vector<std::string> split(const std::string &s, char
   delimiter) {
    std::vector<std::string> out{};
    std::stringstream ss{s};
    std::string item;

    while (std::getline(ss, item, delimiter)) {
        out.push_back(item);
    }

    return out;
}

bool is_prefix(const std::string &s, const std::string &of
   ) {
    if (s.size() > of.size()) return false;
    return std::equal(s.begin(), s.end(), of.begin());
}

bool is_suffix(const std::string &s, const std::string &of
   ) {
    if (s.size() > of.size()) return false;
    auto diff = of.size() - s.size();
    return std::equal(s.begin(), s.end(), of.begin() +
       diff);
}

std::string to_string(symbol_type st) {
    switch (st) {
        case symbol_type::notype:
            return "notype";
        case symbol_type::object:
```

```
            return "object";
        case symbol_type::func:
            return "func";
        case symbol_type::section:
            return "section";
        case symbol_type::file:
            return "file";
    }
}

symbol_type to_symbol_type(elf::stt sym) {
    switch (sym) {
        case elf::stt::notype:
            return symbol_type::notype;
        case elf::stt::object:
            return symbol_type::object;
        case elf::stt::func:
            return symbol_type::func;
        case elf::stt::section:
            return symbol_type::section;
        case elf::stt::file:
            return symbol_type::file;
        default:
            return symbol_type::notype;
    }
}

void print_source(const std::string &file_name, unsigned
  line, unsigned n_lines_context) {
    std::ifstream file{file_name};

    auto start_line = line <= n_lines_context ? 1 : line -
        n_lines_context;
```

```
    auto end_line = line + n_lines_context + (line <
      n_lines_context ? n_lines_context - line : 0) + 1;

    char c{};
    auto current_line = 1u;

    while (current_line != start_line && file.get(c)) {
        if (c == '\n') {
            ++current_line;
        }
    }

    std::cout << (current_line == line ? "> " : "  ");

    while (current_line <= end_line && file.get(c)) {
        std::cout << c;
        if (c == '\n') {
            ++current_line;

            std::cout << (current_line == line ? "> " : "
                ");
        }
    }

    std::cout << std::endl;
}
```

Вспомогательный код

```
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/personality.h>
#include <unistd.h>
#include <iostream>
```

```
#include "parser.h"
#include "debugger.h"

void execute_debugee(const std::string &prog_name) {
    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
        std::cerr << "Error in ptrace\n";
        return;
    }
    execl(prog_name.c_str(), prog_name.c_str(), nullptr);
}

int main(int argc, char *argv[]) {
    ArgParser args(argc, argv);
    if (!args.parse()) {
        return 1;
    }

    auto prog = args.getProgName();

    auto pid = fork();
    if (pid == 0) {
        //child
        personality(ADDR_NO_RANDOMIZE);
        execute_debugee(prog);
    } else if (pid >= 1) {
        //parent
        std::cout << "Started debugging process " << pid
          << '\n';
        debugger dbg{prog, pid};
        dbg.run();
    }
}
```

```
#pragma once

#include <iostream>
#include <sys/user.h>
#include <algorithm>
#include <string>
#include <sys/ptrace.h>
#include <array>

enum class reg {
    rax, rbx, rcx, rdx,
    rdi, rsi, rbp, rsp,
    r8, r9, r10, r11,
    r12, r13, r14, r15,
    rip, rflags, cs,
    orig_rax, fs_base,
    gs_base,
    fs, gs, ss, ds, es
};

static constexpr std::size_t n_registers = 27;

struct reg_descriptor {
    reg r;
    int dwarf_r;
    std::string name;
};

static const std::array<reg_descriptor, n_registers>
  g_register_descriptors{{
    {reg::r15, 15, "r15"},
```

```
    {reg::r14, 14, "r14"},
    {reg::r13, 13, "r13"},
    {reg::r12, 12, "r12"},
    {reg::rbp, 6, "rbp"},
    {reg::rbx, 3, "rbx"},
    {reg::r11, 11, "r11"},
    {reg::r10, 10, "r10"},
    {reg::r9, 9, "r9"},
    {reg::r8, 8, "r8"},
    {reg::rax, 0, "rax"},
    {reg::rcx, 2, "rcx"},
    {reg::rdx, 1, "rdx"},
    {reg::rsi, 4, "rsi"},
    {reg::rdi, 5, "rdi"},
    {reg::orig_rax, -1, "orig_rax"},
    {reg::rip, -1, "rip"},
    {reg::cs, 51, "cs"},
    {reg::rflags, 49, "eflags"},
    {reg::rsp, 7, "rsp"},
    {reg::ss, 52, "ss"},
    {reg::fs_base, 58, "fs_base"},
    {reg::gs_base, 59, "gs_base"},
    {reg::ds, 53, "ds"},
    {reg::es, 50, "es"},
    {reg::fs, 54, "fs"},
    {reg::gs, 55, "gs"},
}};
```