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	Знать основные понятия системного программирования (операционная система, файл, поток, процесс, сигнал и др.); основные системные вызовы, описанные в стандарте 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 решению задачи 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
ст. преподаватель каф. ГИиИБ _____ А.В. Веричев
(подпись)

Задание приняли к исполнению

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

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

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

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

РЕФЕРАТ

Пояснительная записка к курсовому проекту: 23с., 16 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.

1. Выбранные технологии при разработке приложения

1.1 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.

1.2 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.

1.3 Выбор вспомогательных библиотек

1.3.1 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.

1.3.2 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.

1.4 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 систе-

мы 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.

2. Техническое задание на проект

2.1 Требования к проекту

2.1.1 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

2.1.2 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.

2.1.3 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.

2.1.4 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

2.1.5 Требуемые версии средств сборки

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

2.1.6 Поддерживаемые форматы отлаживаемых приложений

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

2.2 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.

2.2.1 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) {
        //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();
    }
}
```

Рис. 2.1:

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. **fork**

   Когда выполнение программы происходит в дочернем процессе, fork возвращает 0, иначе - идентификатор дочернего процесса.

3. **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);
}
```

Рис. 2.2:

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

4. **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);
    }
}
```

Рис. 2.3:

В функции происходит ожидание сигнала от родительского процесса и инициализация адреса отлаживаемой программы. Далее запускается цикл, в котором запрашиваются команды у пользователя.

5. **handle_command**

В функции происходит обработка пользовательских команд.

Пользовательские команды	Вызов функций
cont	continue_execution
break	set_breakpoint_at_address
step	step_in
next	step_over
finish	step_out
show	print_source
register write	set_register_value
register read	get_register_value

6. **continue_execution**

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

Рис. 2.4:

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

7. **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.

8. **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.

9. **set_breakpoint_at_address**

```
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;
}
```

Рис. 2.5:

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

10. **set_breakpoint_at_source_line**

```
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;
                }
            }
        }
    }
}
```

Рис. 2.6:

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.

11. **set_breakpoint_at_function**

```
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);
            }
        }
    }
}
```

Рис. 2.7:

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

3. Пример работы программы

Исходный код отлаживаемого приложения:

```c
#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;
}
```

Рис. 3.1:

Старт:

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

Рис. 3.2:

Вывод исходного кода:

```
MEGAdbg> show
  #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;
  }
```

Рис. 3.3:

Установка точки останова по названию функции:

```
evgeniy@evgeniy:~/SysProg$ ./my_app a.out
Started debugging process 1828
Unknown SIGTRAP code 0
MEGAdbg> break a
Set breakpoint at address 0x5555555545fe
MEGAdbg>
```

Рис. 3.4:

Переход на точку останова:

```
evgeniy@evgeniy:~/SysProg$ ./my_app a.out
Started debugging process 2305
Unknown SIGTRAP code 0
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>
```

Рис. 3.5:

Step:

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

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


MEGAdbg>
```

Рис. 3.6:

Step out:

```
>       int b = 2;
        return a + b;
    }


MEGAdbg> finish
Set breakpoint at address 0x555555554634
Hit breakpoint at address 0x555555554634
    int main(){
        a();
>       b();
        return 0;
    }

MEGAdbg>
```

Рис. 3.7:

Step over:

```
Set breakpoint at address 0x555555554634
Hit breakpoint at address 0x555555554634
    int main(){
        a();
>       b();
        return 0;
    }

MEGAdbg> next
        a();
        b();
>       return 0;
    }

MEGAdbg>
```

Рис. 3.8:

Конец выполнения программы:

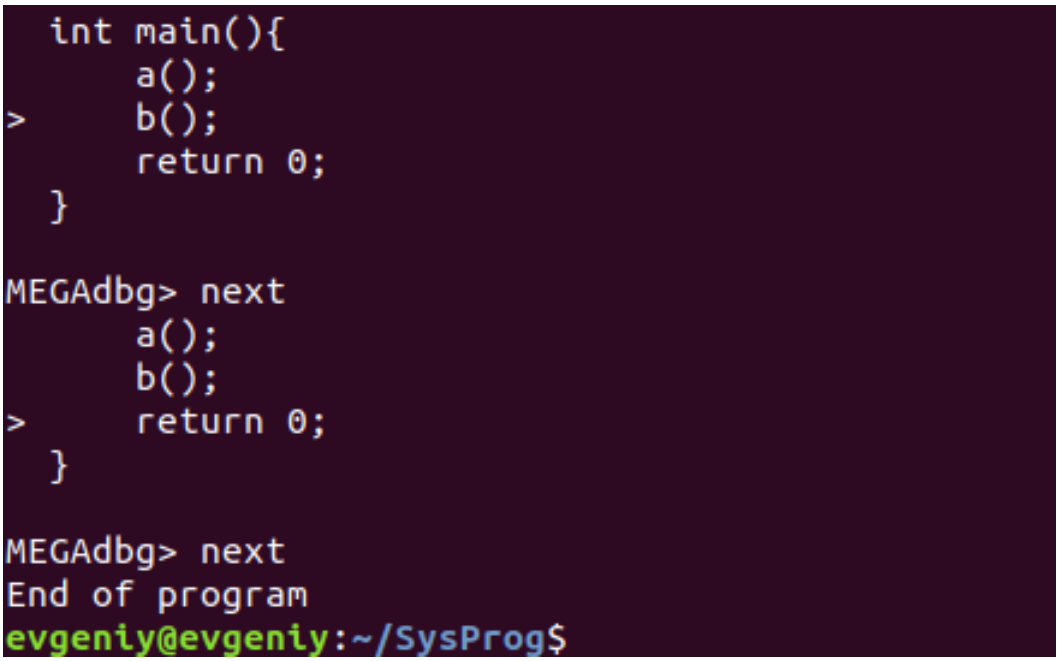

Рис. 3.9:

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