РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9

дисциплина: Архитектура компьютера

Студент: Грицко Сергей

Группа: НКАбд-02-25

МОСКВА

2025 г.

Оглавление

Цель работы

Приобретение навыков написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями.

Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. Этот процесс можно разделить на четыре ключевых этапа, которые выполняются последовательно:

1. **Обнаружение ошибки (Detection):** Фиксация того факта, что программа ведет себя не так, как ожидалось (например, падает или выдает неверный результат).
2. **Поиск её местонахождения (Localization):** Выделение конкретного фрагмента кода, который вызывает проблему. Для этого часто применяется метод «разделяй и властвуй», когда большие участки кода проверяются изолированно.
3. **Определение причины (Diagnosis):** Выяснение точной логической или структурной причины, по которой этот фрагмент кода работает некорректно.
4. **Исправление (Correction):** Внесение изменений в код для устранения выявленной причины.

Существуют различные типы ошибок, требующие разных подходов к исправлению:

1. **Синтаксические:** Это ошибки в правилах языка ассемблера (например, опечатка в имени инструкции или неправильный порядок операндов). Они обнаруживаются на этапе трансляции (компиляции) и не позволяют создать исполняемый файл. Компилятор (NASM) обычно точно указывает строку, где была допущена ошибка.
2. **Семантические (логические):** Самый сложный тип для отладки. Программа запускается и завершается без сбоев, но результат не соответствует заданию. Например, вместо сложения используется вычитание, или не учтено соглашение о регистрах. Такие ошибки требуют пошагового анализа с помощью отладчика.
3. **Ошибки выполнения (Runtime Errors):** Не обнаруживаются компилятором, но приводят к аварийному завершению программы (краху) во время ее работы. Типичными примерами являются переполнение регистров, попытка деления на ноль, или обращение к неверному адресу памяти.

Основными методами отладки являются вывод диагностических сообщений (промежуточных значений переменных или состояния регистров) и использование специальных программ-отладчиков, таких как GDB (GNU Debugger).

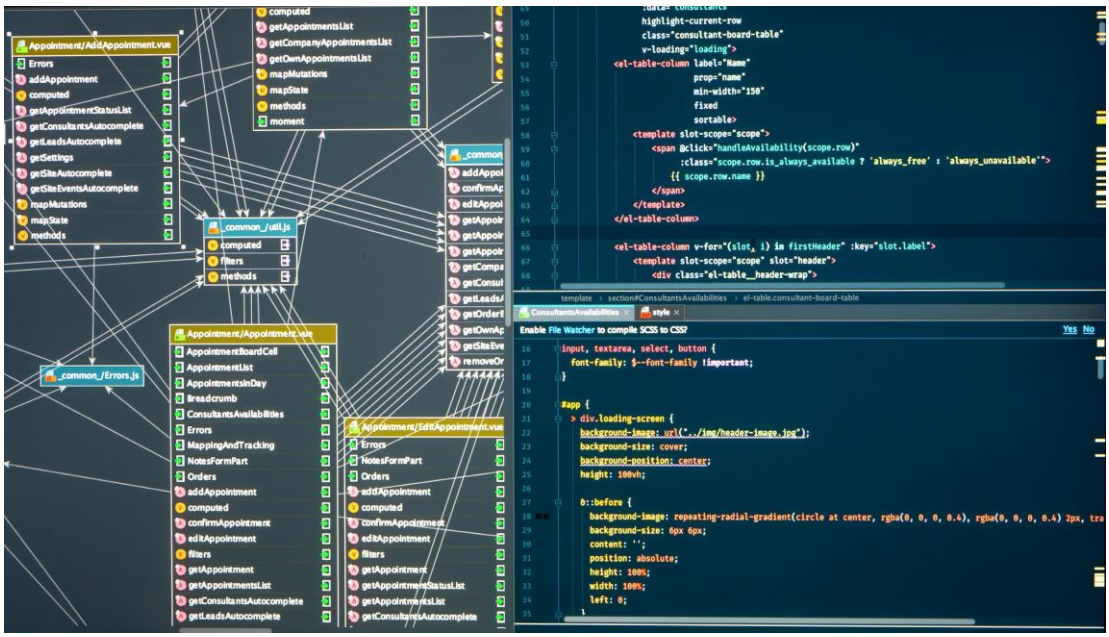

Рисунок 1. GNU Debugger

Отладчики позволяют выполнять программу по шагам (построчно), просматривать значения регистров и переменных, а также устанавливать точки останова (breakpoints) — места, где выполнение программы принудительно прерывается для ручного анализа текущего состояния.

Понятие подпрограммы Подпрограмма (или процедура) — это именованный, самодостаточный блок кода, предназначенный для выполнения определенной задачи. Использование подпрограмм позволяет избежать дублирования кода, делает программу более модульной и удобной для чтения и отладки. Для работы с подпрограммами в ассемблере используются две ключевые инструкции, которые тесно связаны со стеком:

1. call — вызов подпрограммы. При выполнении этой инструкции процессор **сохраняет адрес следующей инструкции** (адрес возврата) в стеке и передает управление по адресу метки подпрограммы.
2. ret — возврат из подпрограммы. Эта инструкция **извлекает сохраненный адрес** из вершины стека и передает управление по этому адресу, тем самым возвращаясь в основную программу (или вызывающую подпрограмму) ровно туда, откуда был произведен вызов.

Это обеспечивает автоматический и корректный возврат управления, что критически важно для реализации как простых, так и вложенных вызовов.

Выполнение лабораторной работы

3.1. Реализация подпрограмм в NASM

Мною была создана программа lab09-1.asm для вычисления сложного выражения $f(g(x))$, где $f(x) = 2x+7$, а $g(x) = 3x-1$.

Для реализации этой задачи я применил принцип модульности, используя вложенные подпрограммы. Это позволяет разделить логику вычислений на независимые блоки. Основная программа вызывает подпрограмму _calcul, которая отвечает за внешнюю функцию $f(x)$. В свою очередь, _calcul вызывает вложенную подпрограмму _subcalcul для предварительного вычисления значения внутренней функции $g(x)$. Передача результатов вычислений между подпрограммами осуществляется через регистр EAX, что является стандартным соглашением для возврата значений.

Текст программы lab9-1.asm: (Смотри файл lab9-1.asm, приложенный к отчету)

```
sgritsko@sgritsko:~/works/arch-pc/lab9$ nasm -f elf lab09-1.asm
sgritsko@sgritsko:~/works/arch-pc/lab9$ ld -m elf_i386 -o lab09-1 lab09-1.o
sgritsko@sgritsko:~/works/arch-pc/lab9$ ./lab09-1
Введите x: 2
Результат f(g(x)): 17
```

Рисунок 2. ***Запуск и проверка программы lab09-1***

Я скомпилировал файл и проверил его корректность на тестовом значении $x = 2$. Проверка правильности выполнения проводилась пошагово:

1. Сначала вычисляется внутренняя функция: $g(2) = 3 \cdot 2 - 1 = 5$.
2. Затем результат передается во внешнюю функцию: $f(5) = 2 \cdot 5 + 7 = 17$.
   Программа выдала ожидаемый результат 17, что подтверждает корректность работы вложенных вызовов и инструкции ret.

3.2. Отладка программ с помощью GDB

Я создал файл lab09-2.asm, выводящий сообщение "Hello World", и скомпилировал его с ключом -g. Использование этого ключа критически важно, так как он добавляет в исполняемый файл отладочную информацию, позволяющую отладчику сопоставлять машинный код с исходным текстом программы. Затем я загрузил исполняемый файл в отладчик gdb.

```
sgritsko@sgritsko:~/works/arch-pc/lab9$ nasm -f elf -g lab09-2.asm -o lab09-2.o
sgritsko@sgritsko:~/works/arch-pc/lab9$ ld -m elf_i386 -o lab09-2 lab09-2.o
sgritsko@sgritsko:~/works/arch-pc/lab9$ gdb lab09-2
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-2...
(gdb)
```

Рисунок 3. ***Запуск GDB***

В ходе работы с отладчиком я выполнил ряд действий для анализа состояния процессора:

1. Запустил программу командой run.
2. Установил точку останова на начальную метку _start с помощью команды break _start, чтобы прервать выполнение перед первой инструкцией.
3. Включил режим дизассемблирования (disassemble) и переключился на синтаксис Intel командой set disassembly-flavor intel. Это сделало код более читаемым, так как порядок операндов стал привычным (приемник, источник), как в исходном коде NASM, в отличие от стандартного для GDB синтаксиса AT&T.
4. Активировал псевдографический интерфейс командой layout asm и layout regs. Это разделило окно терминала на области, отображающие текущий ассемблерный код, состояние регистров и командную строку.

Используя команду si (step instruction) для пошагового выполнения, я наблюдал, как меняются значения в регистрах eax, ebx, ecx и edx после каждой инструкции mov и int. Я также следил за регистром eip (Instruction Pointer), который указывал на адрес следующей выполняемой команды.

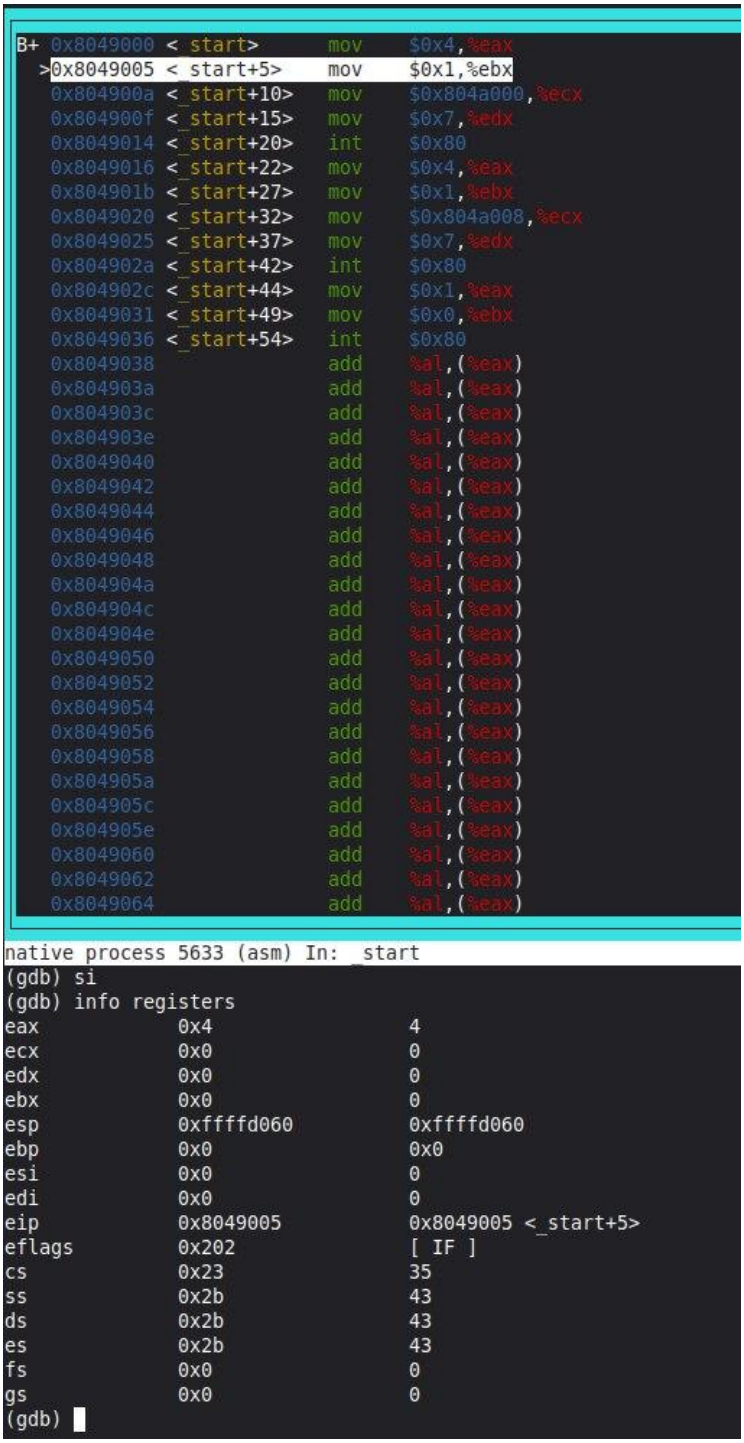

Рисунок 4. ***Работа в GDB (просмотр регистров)***

3.3. Аргументы командной строки в GDB

Для изучения передачи параметров я скопировал программу из предыдущей лабораторной работы в файл lab09-3.asm и запустил её под отладчиком с указанием аргументов командной строки. Особое внимание было уделено анализу стека.

Исследовав вершину стека с помощью команды x/x $esp (просмотр памяти по адресу регистра Stack Pointer), я убедился в следующем:

1. Вершина стека (адрес в $esp) хранит количество аргументов (argc).
2. Следующая ячейка памяти по адресу $esp + 4 содержит адрес имени программы.
3. Далее, по адресам $esp + 8, $esp + 12 и так далее, хранятся указатели на строки самих аргументов.

Шаг смещения равен 4 байтам, поскольку мы работаем в 32-битной архитектуре, где каждый адрес памяти занимает ровно 4 байта (32 бита). Это наблюдение подтверждает теоретические знания о том, как операционная система передает данные в программу при запуске.

Выполнение заданий для самостоятельной работы

Задание 1. Преобразование программы (Вариант 14)

Согласно варианту 14, мне необходимо вычислить функцию $f(x) = 2x + 10$ (на основе заданий из ЛР №8). Я выделил вычисление функции в отдельную подпрограмму _function_variant.

Листинг программы (основные части):

```
sgritsko@sgritsko:~/works/arch-pc/lab9$ ./lab09-ind1
Введите x (Вариант 14: 2x + 10): 2
Ответ: 14
```

Рисунок 5. ***Проверка работы программы***

Задание 2. Альтернативная реализация (Вариант 14)

Второе задание заключалось в альтернативной реализации функции Варианта 14 ($f(x) = 2x +

10$). Эта реализация была выполнена без использования подпрограммы (в отличие от Задания 1) для демонстрации прямого вычисления.

Листинг программы (основные части):

```
sgritsko@sgritsko:~/works/arch-pc/lab9$ nasm -f elf lab09-ind2.asm
sgritsko@sgritsko:~/works/arch-pc/lab9$ ld -m elf_i386 -o lab09-ind2 lab09-ind2.o
sgritsko@sgritsko:~/works/arch-pc/lab9$ ./lab09-ind2
Введите x (для 2x + 10, Самостоятельная 2): 2
Результат: 14
```

Рисунок 6. ***Запуск альтернативной реализации***

Выводы

В ходе лабораторной работы я:

1. Научился оформлять повторяющиеся участки кода в виде подпрограмм, используя инструкции call и ret.
2. Понял принцип работы стека при вызове процедур (сохранение адреса возврата).
3. Освоил работу с отладчиком GDB: запуск, пошаговое выполнение (si), установка точек останова (break), просмотр регистров (info registers) и памяти (x).
4. Научился использовать отладчик для поиска логических ошибок в ассемблерном коде.

Список литературы

1. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/.
2. NASM Assembly Language Tutorials. — 2021. — URL: https://asmtutor.com/.
3. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
4. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.
5. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. М. : МАКС Пресс, 2011. 6. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. 12