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: Ниемек Яи Жак

Группа: НММБд-04-24

МОСКВА

2025_ г.

Цель работы

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

Задание

1. Создайте каталог для выполнения лабораторной работы № 9, перейдите в него и создайте файл lab09-1.asm 2. Внимательно изучите текст программы (Листинг 9.1). Введите в файл lab09- 1.asm текст программы из листинга 9.1. Создайте исполняемый файл и проверьте его работу 3. Создайте файл lab09-2.asm с текстом программы из Листинга 9.2 4. Проверьте работу программы, запустив ее в оболочке GDB

Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа: • обнаружение ошибки; • поиск её местонахождения; • определение причины ошибки; • исправление ошибки. Можно выделить следующие типы ошибок: • синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой

формы или структуры языка; • семантические ошибки— являются логическими и

приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата; • ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки,

связанные с переполнением или делением на ноль). Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить довольно трудно. Лучший

способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга. Третий этап — выяснение причины ошибки. После определения местонахождения ошибки

обычно
проще определить причину неправильной работы программы. Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново. Наиболее часто применяют следующие методы отладки: • создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных
значений на экран —так называемые диагностические сообщения); • использование специальных программ-отладчиков. Отладчики позволяют управлять ходом
выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам. Пошаговое выполнение — это выполнение программы с остановкой после каждой строчки, чтобы программист мог про□верить значения переменных и выполнить другие действия. Точки останова — это специально отмеченные места в программе, в которых программа отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова: • Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмечен□ной программистом); • Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его). Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы.

Разбор задания и план выполнения:

1. **Переделать программу из лабораторной работы №9**
    - Вынести вычисление функции f(x) в отдельную подпрограмму.
2. **Использовать отладчик GDB для исправления ошибки в ассемблерной программе**
    - Проверить, почему программа Листинг 9.3 дает неверный результат.
    - Найти и исправить ошибку.

Часть 1: Преобразование программы с подпрограммой

Допустим, в ЛР8 вам нужно было вычислять функцию:

f(x)=2x2+3y2+xy−4x−5yf(x) = 2x^2 + 3y^2 + xy - 4x - 5y

Теперь вынесем её в подпрограмму:

```
section .data
    msg db 'Результат: ', 0

section .bss
    result resd 1

section .text
    global _start

_start:
    ; Вводим x и y (можно заменить на ввод с клавиатуры)
    mov eax, 3
    mov ebx, 4

    call calculate_f    ; вызываем подпрограмму

    mov edi, eax
    mov eax, msg
    call sprint
    mov eax, edi
    call iprintLF
```

```
    call quit

calculate_f:
    ; Подпрограмма для f(x, y) = 2x^2 + 3y^2 + xy - 4x - 5y
    push ebx
    push eax

    mov ecx, eax
    imul ecx, eax          ; x^2
    imul ecx, 2            ; 2 * x^2

    mov edx, ebx
    imul edx, ebx          ; y^2
    imul edx, 3            ; 3 * y^2
    add ecx, edx           ; 2x^2 + 3y^2

    mov edx, eax
    imul edx, ebx          ; x * y
    add ecx, edx           ; + xy

    imul eax, 4            ; -4x
    sub ecx, eax           ; -4x

    imul ebx, 5            ; -5y
    sub ecx, ebx           ; -5y

    mov eax, ecx           ; Сохранение результата в EAX

    pop eax
    pop ebx
    ret
```

Часть 2: Исправление ошибки в Листинге 9.3

Анализ кода

Ошибка, скорее всего, связана с неправильным порядком операций. В строке:

mov ecx, 4

mul ecx

используется mul, но mul работает с eax, а не с ebx, как ожидалось.

Исправленный код

```
%include 'in_out.asm'

SECTION .data
div: DB 'Результат: ', 0

SECTION .text
GLOBAL _start

_start:
    mov eax, 3
    add eax, 2        ; (3+2)

    mov ecx, 4
    imul eax, ecx ; (3+2) * 4

    add eax, 5        ; (3+2) * 4 + 5

    mov edi, eax    ; Вывод результата
    mov eax, div
    call sprint
    mov eax, edi
    call iprintLF
    call quit
```

Исправления

1. Используем eax вместо ebx, так как mul работает только с eax.

2. Используем imul, чтобы явно умножить (3+2) * 4.
3. Теперь программа корректно вычисляет выражение (3+2) * 4 + 5.

   Это исправляет ошибку, и теперь программа будет выдавать правильный результат **25**.

```
nyemeckyai@fedora:~/work/arch-pc/lab08$ cd
nyemeckyai@fedora:~$ cd work
nyemeckyai@fedora:~/work$ cd arch-pc
nyemeckyai@fedora:~/work/arch-pc$ mkdir lab09
nyemeckyai@fedora:~/work/arch-pc$ touch lab9-1.asm
nyemeckyai@fedora:~/work/arch-pc$ ls
lab03  lab04  lab05  lab06  lab07  lab08  lab09  lab9-1.asm
```

```
 GNU nano 8.1
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax,x
call atoi
call _calcul ;
mov eax,result
call sprint
mov eax,[res]
call iprintLF
call quit

_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret
```

```
nyemeckyai@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
nyemeckyai@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 23
2x+7=53
nyemeckyai@fedora:~/work/arch-pc/lab09$ nano lab9-1.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$
```

```
lab9-2.asm          [----]  0 L:[
SECTION .data
msg1: db "Hello, ",0x0
msg1Len: equ $ - msg1
msg2: db "world!",0xa
msg2Len: equ $ - msg2
SECTION .text
global _start
_start:
mov eax, 4
mov ebx, 1
mov ecx, msg1
mov edx, msg1Len
int 0x80
mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, msg2Len
int 0x80
mov eax, 1
mov ebx, 0
int 0x80
```

```
nyemeckyai@fedora:~/work/arch-pc/lab09$ ls
in_out.asm  lab9-1  lab9-1.asm  lab9-1.o  lab9-2.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$ mc

nyemeckyai@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-2.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
nyemeckyai@fedora:~/work/arch-pc/lab09$ ./lab9-2
Hello, world!
nyemeckyai@fedora:~/work/arch-pc/lab09$
```

```
nyemeckyai@fedora:~/work/arch-pc/lab09$ nasm -f elf -g -l  lab9-2.lst lab9-2.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
```

```
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   <PENDING>  start
2       breakpoint     keep y   0x08049000 lab9-2.asm:9
        breakpoint already hit 1 time
3       breakpoint     keep y   0x08049031 lab9-2.asm:20
(gdb)
```

```
nyemeckyai@fedora:~/work/arch-pc/lab09$ nasm -f elf -g -l  lab9-2.lst lab9-2.asm
nyemeckyai@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
```

```
nyemeckyai@fedora:~/work/arch-pc/lab09$ gdb lab9-2
GNU gdb (Fedora Linux) 15.1-1.fc41
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-redhat-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 lab9-2...
(gdb) run
Starting program: /home/nyemeckyai/work/arch-pc/lab09/lab9-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0xf7ffc000
Download failed: No route to host.  Continuing without separate debug info for system-supplied DSO at
 0xf7ffc000.
Hello, world!
[Inferior 1 (process 5610) exited normally]
(gdb)
(gdb) break _start
Breakpoint 2 at 0x8049000: file lab9-2.asm, line 9.
(gdb) run
Starting program: /home/nyemeckyai/work/arch-pc/lab09/lab9-2
Downloading separate debug info for system-supplied DSO at 0xf7ffc000
Download failed: No route to host.  Continuing without separate debug info for system-supplied DSO at
 0xf7ffc000.

Breakpoint 2, _start () at lab9-2.asm:9
9       mov eax, 4
(gdb)
```

```
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:     mov    $0x4,%eax
   0x08049005 <+5>:     mov    $0x1,%ebx
   0x0804900a <+10>:    mov    $0x804a000,%ecx
   0x0804900f <+15>:    mov    $0x8,%edx
   0x08049014 <+20>:    int    $0x80
   0x08049016 <+22>:    mov    $0x4,%eax
   0x0804901b <+27>:    mov    $0x1,%ebx
   0x08049020 <+32>:    mov    $0x804a008,%ecx
   0x08049025 <+37>:    mov    $0x7,%edx
   0x0804902a <+42>:    int    $0x80
   0x0804902c <+44>:    mov    $0x1,%eax
   0x08049031 <+49>:    mov    $0x0,%ebx
   0x08049036 <+54>:    int    $0x80
End of assembler dump.
(gdb)
```

```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:     mov    eax,0x4
   0x08049005 <+5>:     mov    ebx,0x1
   0x0804900a <+10>:    mov    ecx,0x804a000
   0x0804900f <+15>:    mov    edx,0x8
   0x08049014 <+20>:    int    0x80
   0x08049016 <+22>:    mov    eax,0x4
   0x0804901b <+27>:    mov    ebx,0x1
   0x08049020 <+32>:    mov    ecx,0x804a008
   0x08049025 <+37>:    mov    edx,0x7
   0x0804902a <+42>:    int    0x80
   0x0804902c <+44>:    mov    eax,0x1
   0x08049031 <+49>:    mov    ebx,0x0
   0x08049036 <+54>:    int    0x80
End of assembler dump.
(gdb)
```

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   <PENDING>  start
2       breakpoint     keep y   0x08049000 lab9-2.asm:9
        breakpoint already hit 1 time
(gdb)
```

1. Какие языковые средства используются в ассемблере для оформления и активизации подпрограмм?

В ассемблере для оформления подпрограмм (процедур) используются:

- **Метки (Labels)**: метки используются для обозначения начала подпрограммы.
- **Инструкция `call`**: используется для вызова подпрограммы.
- **Инструкция `ret`**: используется для возврата из подпрограммы.
- **Регистры**: регистры используются для передачи аргументов в подпрограмму и для возврата результата.
- **Инструкция `push` и `pop`**: используются для сохранения и восстановления значений регистров, которые нужно передать между подпрограммами.

2. Объясните механизм вызова подпрограмм.

Механизм вызова подпрограммы в ассемблере включает следующие шаги:

- **Подготовка параметров**: значения, которые передаются в подпрограмму, сохраняются в регистрах или на стеке.
- **Вызов подпрограммы**: при помощи инструкции `call` управление передается на метку, которая соответствует подпрограмме.
- **Сохранение состояния**: в подпрограмме обычно сохраняются регистры и другие важные данные на стеке с помощью инструкции `push`.
- **Выполнение подпрограммы**: выполняются инструкции, которые принадлежат подпрограмме.
- **Возврат из подпрограммы**: при помощи инструкции `ret` управление возвращается в место, где была вызвана подпрограмма, и значения регистров восстанавливаются из стека.

3. Как используется стек для обеспечения взаимодействия между вызывающей и вызываемой процедурами?

Стек используется для:

- **Сохранения состояния регистров**: перед тем, как вызвать подпрограмму, часто сохраняются значения регистров на стеке. После выполнения подпрограммы эти значения восстанавливаются.
- **Передачи параметров**: параметры могут быть переданы в подпрограмму через стек, особенно если количество параметров превышает количество доступных регистров.
- **Возврат из подпрограммы**: на стек также помещается адрес возврата (адрес инструкции, следующей после вызова подпрограммы), чтобы можно было вернуться в правильную точку выполнения.

4. Каково назначение операнда в команде `ret`?

Операнд в команде `ret` указывает, сколько байт следует освободить из стека при возврате из подпрограммы. Это полезно, если параметры были переданы через стек, и их нужно удалить после выполнения подпрограммы. Например:

- `ret 4` — освобождает 4 байта из стека (например, если 1 параметр был передан через стек).
- `ret` — стандартный возврат, не освобождая дополнительные байты.

5. Для чего нужен отладчик?

Отладчик используется для пошагового выполнения программы с целью обнаружения ошибок (багов) и понимания её поведения. Он позволяет:

- Остановить выполнение программы на определённых инструкциях (breakpoint).
- Просмотреть значения переменных и регистров.
- Пошагово выполнять код, проверяя, как изменяются данные.
- Анализировать память и стэк.

6. Объясните назначение отладочной информации и как нужно компилировать программу, чтобы в ней присутствовала отладочная информация.

Отладочная информация включает в себя данные, такие как имена переменных, функции, линии кода и другие элементы, которые помогают отладчику понять, что происходит в программе. Для включения отладочной информации при компиляции программы в GCC используется флаг `-g`. Пример:

```
gcc -g -o myprogram myprogram.c
```

Это создаст исполнимый файл с дополнительной информацией, которую можно использовать с отладчиком, например, GDB.

7. Расшифруйте и объясните следующие термины: breakpoint, watchpoint, checkpoint, catchpoint и call stack.

- **Breakpoint (точка останова)**: Это метка в программе, на которой её выполнение останавливается, чтобы анализировать состояние программы.
- **Watchpoint (точка отслеживания)**: Это точка останова, которая активируется, когда изменяется значение переменной или регистра.
- **Checkpoint (контрольная точка)**: Место в программе, где сохранено состояние, чтобы в случае ошибки можно было вернуться в это состояние.
- **Catchpoint (точка перехвата)**: Устанавливается для перехвата определённого события в программе (например, исключения или сигналов).
- **Call stack (стек вызовов)**: Список функций, которые были вызваны, но ещё не завершены. Это помогает отладчику проследить последовательность вызовов функций и выяснить, где возникла ошибка.

8. Назовите основные команды отладчика GDB и как они могут быть использованы для отладки программ.

- **`break <location>`**: Устанавливает точку останова на указанной строке или в функции.
- **`run`**: Запускает программу с аргументами.
- **`step`**: Пошагово выполняет программу, заходя в функции.
- **`next`**: Пошагово выполняет программу, не заходя в функции.
- **`continue`**: Возобновляет выполнение программы до следующей точки останова.
- **`print <variable>`**: Печатает значение переменной.
- **`bt` (backtrace)**: Показывает стек вызовов.
- **`quit`**: Завершается работа отладчика.

Эти команды помогают отладчику отслеживать выполнение программы, анализировать ошибки и проверять правильность работы кода.