

Лабораторная работа №9

НКАБд-06-23

Улитина Мария Максимовна

Содержание

1 Цель работы.....	1
2 Задание	2
3 Теоретическое введение	2
3.1 Понятие об отладке	2
3.2 Методы отладки.....	3
3.3 Основные возможности отладчика GDB.....	3
3.4 Запуск отладчика GDB; выполнение программы; выход	4
3.5 Дизассемблирование программы	4
3.6 Точки останова	4
3.7 Пошаговая отладка	5
3.8 Работа с данными программы в GDB.....	5
3.9 Понятие подпрограммы.....	5
3.9.1 Инструкция call и инструкция ret.....	5
4 Выполнение лабораторной работы	6
4.1 Задания для самостоятельной работы	12
5 Выводы	14
Список литературы.....	14

1 Цель работы

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

2 Задание

1. Реализация подпрограмм в NASM.
2. Отладка программ с помощью GDB.
3. Добавление точек останова.
4. Работа с данными программы в GDB.
5. Обработка аргументов командной строки в GDB.
6. Преобразование программы из лабораторной работы №8, реализовав вычисление значения функции как подпрограмму.
7. Проверить неправильную работу программы, проанализировав изменения значения регистров. Определить ошибку и исправить ее.

3 Теоретическое введение

3.1 Понятие об отладке

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.

3.2 Методы отладки

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 проверить 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, отмеченной 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.

3.3 Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки. Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя. GDB может выполнять следующие действия:

- начать выполнение программы, задав всё, что может повлиять на её поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась;
- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

3.4 Запуск отладчика GDB; выполнение программы; выход

Синтаксис команды для запуска отладчика имеет следующий вид: `gdb [опции] [имя_файла / ID процесса]` После запуска `gdb` выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (`gdb`) для ввода команд.

3.5 Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

3.6 Точки останова

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»: `(gdb) break * (gdb) b` Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`): `(gdb) info breakpoints (gdb) i b` Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой `disable`: `disable breakpoint` Обратная точка останова активируется командой `enable`: `enable breakpoint` Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`: `(gdb) delete breakpoint` Ввод этой команды без аргумента удалит все точки останова. Информацию о командах этого раздела можно получить, введя `help breakpoints`

3.7 Пошаговая отладка

Для продолжения остановленной программы используется команда `continue` (`c`) `(gdb) c [аргумент]`. Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число n , которое указывает отладчику проигнорировать $n - 1$ точку останова (выполнение остановится на n -й точке). Команда `stepi` (кратко `si`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию: `(gdb) si [аргумент]` При указании в качестве аргумента целого числа n отладчик выполнит команду `step` n раз при условии, что не будет точек останова или выполнение

программы не прервётся по другим причинам. Команда `nexth` (или `ni`) аналогична `stepi`, но вызов процедуры (функции) трактуется отладчиком как одна инструкция: *(gdb) ni [аргумент]* Информацию о командах этого раздела можно получить, введя *(gdb) help running*

3.8 Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, та при необходимости позволяет вручную изменять значения регистров и переменных. Посмотреть содержимое регистров можно с помощью команды `info registers` (или `i r`): *(gdb) info registers*

3.9 Понятие подпрограммы

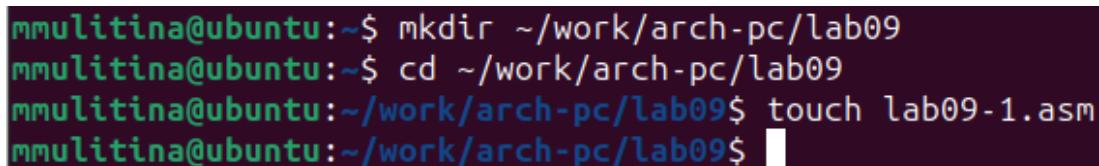
Подпрограмма — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом. Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

3.9.1 Инструкция `call` и инструкция `ret`

Для вызова подпрограммы из основной программы используется инструкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `esp` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы. Подпрограмма завершается инструкцией `ret`, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией `call`, и заносит его в `esp`. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией `call`.

4 Выполнение лабораторной работы

Создадим каталог и файл для лабораторной работы (рис. 1).



```
mmulitina@ubuntu:~$ mkdir ~/work/arch-pc/lab09
mmulitina@ubuntu:~$ cd ~/work/arch-pc/lab09
mmulitina@ubuntu:~/work/arch-pc/lab09$ touch lab09-1.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$
```

Figure 1: Создание каталога

Введём текст программы и запустим её для проверки (рис. 2).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 2
2x+7=11
```

Figure 2: Программа

Добавим подпрограмму `_subcalcul`, запустим программу для проверки (рис. 3).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 1
2x+7=11
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 2
2x+7=17
```

Figure 3: Программа

Создадим файл `lab09-2.asm`, введём в него текст программы, получим исполняемый файл и загрузим его в отладчик (рис. 4).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ touch lab09-2.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-2 lab09-2.o
mmulitina@ubuntu:~/work/arch-pc/lab09$ gdb lab09-2
```

Figure 4: Отладчик

Проверим работу программы, запустим ее в оболочке GDB (рис. 5).

```
(gdb) run
```

Figure 5: Отладчик

(рис. 6).

```
Hello, world!
[Inferior 1 (process 71914) exited normally]
(gdb)
```

Figure 6: Отладчик

Установим брейкпоинт и запустим программу (рис. 7).

```

(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/mmilitina/work/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:9
9      mov eax, 4
(gdb) █

```

Figure 7: Брейкпоинт

Посмотрим дисассимилированный код (рис. 8).

```

(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) █

```

Figure 8: Дисассимилированный код

Переключимся на отображение команд с Intel синтаксисом (рис. 9).

```

(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.

```

Figure 9: Отображение

Включим режим псевдографики (рис. 10).

```

[ Register Values Unavailable ]

0x80491a4      add     BYTE PTR [eax],al
0x80491a6      add     BYTE PTR [eax],al
0x80491a8      add     BYTE PTR [eax],al
0x80491aa      add     BYTE PTR [eax],al
0x80491ac      add     BYTE PTR [eax],al
0x80491ae      add     BYTE PTR [eax],al

native process 71924 In: _start          L9      PC: 0x8049000
(gdb) layout regs
(gdb)

```

Figure 10: Псевдографика

Проверим точки останова с помощью команды info breakpoints (рис. 11).


```
(gdb) layout regs
(gdb) info breakpoints
Num      Type          Disp Enb Address      What
1        breakpoint    keep y  0x08049000 lab09-2.asm:9
breakpoint already hit 1 time
```

Figure 11: Точки останова

Установим ещё одну точку останова и снова посмотрим информацию о точках останова (рис. 12).

```
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031: file lab09-2.asm, line 20.
(gdb) i b
Num      Type          Disp Enb Address      What
1        breakpoint    keep y  0x08049000 lab09-2.asm:9
breakpoint already hit 1 time
2        breakpoint    keep y  0x08049031 lab09-2.asm:20
(gdb)
```

Figure 12: Точки останова

Выполним 5 инструкций с помощью команды `stepi` и проследим изменения регистров (рис. 13).

```
Register group: general
eax      0x8          8
ecx      0x804a000    134520832
edx      0x8          8
ebx      0x1          1
esp      0xffffd030   0xffffd030
ebp      0x0          0x0

0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
> 0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a008

native process 72040 In: _start L14 PC: 0x8049016
breakpoint already hit 1 time
2 breakpoint keep y 0x08049031 lab09-2.asm:20
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb)
```

Figure 13: `stepi`

Изменились значения регистров `eax`, `ecx`, `ebx`, `edx`.

Посмотрим значения регистров с помощью `info registers` (рис. 14).

```

eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd030 0xffffd030
ebp      0x0      0x0
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--

```

Figure 14: Значения регистров

Посмотрим значение переменной msg1 по имени (рис. 15).

```

(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "

```

Figure 15: Значение переменной

Изменим первый символ переменной (рис. 16).

```

(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "

```

Figure 16: Изменение символа

Изменим любой символ второй переменной msg2 (рис. 17).

```

(gdb) set {char}&msg2 = 'a'
(gdb) x/1sb &msg2
0x804a008 <msg2>: "aorld!\n\034"

```

Figure 17: Изменение символа

Посмотрим значений регистра edx (рис. 18).

```

(gdb) p/x $edx
$1 = 0x8
(gdb) p/t $edx
$2 = 1000
(gdb) p/c $edx
$3 = 8 '\b'

```

Figure 18: Значения регистра

С помощью set изменим значение регистра ebx (рис. 19).

```
(gdb) set $ebx = '2'
```

Figure 19: Изменение значения регистра

и проверим его значение (рис. 20).

```
(gdb) p/s $ebx  
$4 = 50
```

Figure 20: Значение

Снова изменим значение ebx (рис. 21).

```
(gdb) set $ebx=2  
(gdb) p/s $ebx  
$5 = 2
```

Figure 21: Значение

В первом случае мы ввели символьное значение, во втором цифру.

Скопируем файл из прошлой лабораторной работы (рис. 22).

```
mmulitina@ubuntu:~$ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
```

Figure 22: Копирование файла

Создадим исполняемый файл (рис. 23).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm  
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-3 lab09-3.o
```

Figure 23: Исполняемый файл

Загрузим исполняемый файл в отладчик, указав аргументы (рис. 24).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ gdb --args lab09-3 arg1 arg 2 'arg3'
```

Figure 24: Отладчик

Установим точку останова перед первой инструкцией в программе и запустим её (рис. 25).

```
(gdb) b _start  
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.  
(gdb) run
```

Figure 25: Точка останова

Посмотрим значение регистра esp, где хранится адрес вершины стека (рис. 26).

```
(gdb) x/x $esp
0xffffd010: 0x00000005
```

Figure 26: Регистр esp

Посмотрим остальные позиции стека по адресу (рис. 27).

```
(gdb) x/s *(void**)(esp + 4)
0xffffd1f4: "/home/mmilitina/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd21f: "arg1"
(gdb) x/s *(void**)(esp + 12)
0xffffd224: "arg"
(gdb) x/s *(void**)(esp + 16)
0xffffd228: "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd22a: "arg3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
```

Figure 27: Позиции стека

Шаг изменения равен 4, т.к. у нас 4 аргумента.

4.1 Задания для самостоятельной работы

1. Преобразуем программу из лабораторной работы №8, реализовав вычисление значения функции как подпрограмму (рис. 28).

```
_calcul:
mov edx,10
mul edx
sub eax,5
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент `esi=esi+eax`
ret
```

Figure 28: Подпрограмма

Запустим программу для проверки (рис. 29).

```
mmilitina@ubuntu:~/work/arch-pc/lab09$ ./prog1 1 2 3 4
Результат: 80
```

Figure 29: Проверка

2. Создадим файл для программы, введём в него текст программы, запустим его в отладчике GDB (рис. 30).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ touch lab09-4.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-4.lst lab09-4.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-4 lab09-4.o
mmulitina@ubuntu:~/work/arch-pc/lab09$ gdb lab09-4
```

Figure 30: Файл

При умножении с помощью `mul`, мы умножаем `eax` на `ecx` и записываем в `eax`. Получаем $24=9$ вместо $(3+2)4$ (рис. 31).

<code>eax</code>	<code>0x8</code>	8
<code>ecx</code>	<code>0x4</code>	4
<code>edx</code>	<code>0x0</code>	0
<code>ebx</code>	<code>0x5</code>	5

Figure 31: Проверка

Потом скалдываем с регистром `ebx` 5 и получаем 10. Проверим это, запустив программу (рис. 32).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-4
Результат: 10
```

Figure 32: Запуск программы

Исправим программу (рис. 33).

```
mov ebx,3
mov eax,2
add eax,ebx
mov ecx,4
mul ecx
add eax,5
mov edi,eax
```

Figure 33: Исправление программы

Запустим её для проверки (рис. 34).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab09-4.asm
mmulitina@ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-4 lab09-4.o
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-4
Результат: 25
```

Figure 34: Запуск

5 Выводы

В процессе выполнения работы я приобрела навыки написания программ с использованием подпрограмм и познакомилась с методами отладки при помощи GDB и его основными возможностями.

Список литературы

Лабораторная работа №9.