

Отчёт по лабораторной работе №10

Уткина Алиина Дмитриевна

Содержание

1	Цель работы	4
2	Выполнение лабораторной работы	5
2.1	Реализация подпрограмм в NASM	5
2.2	Отладка программ с помощью GDB	7
2.2.1	Добавление точек останова	10
2.2.2	Работа с данными программы в GDB	11
2.2.3	Обработка аргументов командной строки в GDB	13
3	Выводы	15

Список иллюстраций

2.1	Пример программы с использованием вызова подпрограммы . .	6
2.2	Результат работы программы с вызовом подпрограммы	7
2.3	Добавление подпрограммы в подпрограмму	7
2.4	Результат работы измененной программы	7
2.5	Программа печати сообщения Hello world!	8
2.6	Трансляция программы для работы с GDB	8
2.7	Запуск программы в оболочке GDB	8
2.8	Запуск программы в оболочке GDB с подробным анализом	9
2.9	Просмотр дисассимилированного кода программы	9
2.10	Отображение команд с Intel'овским синтаксисом	9
2.11	Режим псевдографики	10
2.12	Проверка установленной метки	11
2.13	Установка точки останова	11
2.14	Просмотр значения регистров	12
2.15	Просмотр значения переменной	12
2.16	Изменение символа ппеременной msg1	12
2.17	Изменение символа переменной msg2	13
2.18	Просмотр значения регистра	13
2.19	Вывод значения регистра в различных форматах и их изменение	13
2.20	Работа с программой lab10-3	14
2.21	Просмотр позиций стека	14

1 Цель работы

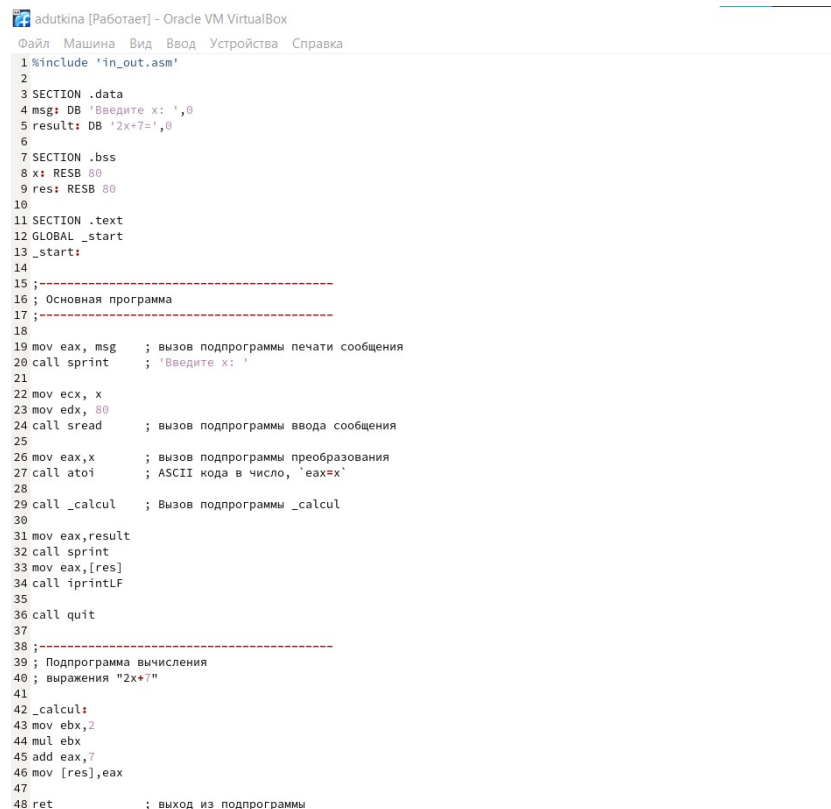
Целью данной работы является приобретение навыков написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями.

2 Выполнение лабораторной работы

2.1 Реализация подпрограмм в NASM

Создадим каталог для выполнения лабораторной работы No 10, перейдем в него и создадим файл lab10-1.asm.

В качестве примера рассмотрим программу вычисления арифметического выражения $f(x) = 2x + 7$ с помощью подпрограммы `_calcul`. В данном примере `x` вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Внимательно изучим текст программы листинга 10.1 и введем его в созданный файл (рис. 2.1)



```
1 %include 'in_out.asm'
2
3 SECTION .data
4 msg: DB 'Введите x: ',0
5 result: DB '2x+7=',0
6
7 SECTION .bss
8 x: RESB 80
9 res: RESB 80
10
11 SECTION .text
12 GLOBAL _start
13 _start:
14
15 ;-----
16 ; Основная программа
17 ;-----
18
19 mov eax, msg      ; вызов подпрограммы печати сообщения
20 call sprint       ; 'Введите x: '
21
22 mov ecx, x
23 mov edx, 80
24 call sread       ; вызов подпрограммы ввода сообщения
25
26 mov eax, x        ; вызов подпрограммы преобразования
27 call atoi        ; ASCII кода в число, 'eax=x'
28
29 call _calcul      ; Вызов подпрограммы _calcul
30
31 mov eax, result
32 call sprint
33 mov eax, [res]
34 call iprintLF
35
36 call quit
37
38 ;-----
39 ; Подпрограмма вычисления
40 ; выражения "2x+7"
41
42 _calcul:
43 mov ebx, 2
44 mul ebx
45 add eax, 7
46 mov [res], eax
47
48 ret              ; выход из подпрограммы
```

Рис. 2.1: Пример программы с использованием вызова подпрограммы

Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi). После следующей инструкции call _calcul, которая передает управление подпрограмме _calcul, будут выполнены инструкции подпрограммы, написанные до ret. Инструкция ret является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией call, которая вызвала данную подпрограмму. Последние строки программы реализуют вывод сообщения (call sprint), результата вычисления (call iprintLF) и завершение программы (call quit).

Создадим исполняемый файл и проверим его работу (рис. 2.2)

```
[adutkina@fedora lab10]$ ./lab10-1
Введите x: 3
2x+7=13
[adutkina@fedora lab10]$
```

Рис. 2.2: Результат работы программы с вызовом подпрограммы

Изменим текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения $f(g(x))$, где x вводится с клавиатуры, $f(x) = 2x + 7$, $g(x) = 3x - 1$. Т.е. x передается в подпрограмму `_calcul`, из нее подпрограмму `_subcalcul`, где вычисляется выражение $g(x)$, результат возвращается в `_calcul` и вычисляется выражение $f(g(x))$. Результат возвращается в основную программу для вывода результата на экран (рис. 2.3), (рис. 2.4).

```
-----
; Подпрограмма вычисления
; выражения "2x+7"

_calcul:
call _subcalcul ; выход подпрограммы _subcalcul
mov ebx, 2
mul ebx
add eax, 7
mov [res], eax

ret          ; выход из подпрограммы

-----
; Подпрограмма вычисления
; выражения "3x-1"

_subcalcul:
mov ebx, 3
mul ebx
sub eax, 1
mov [res], eax

ret          ; выход из подпрограммы
```

Рис. 2.3: Добавление подпрограммы в подпрограмму

```
[adutkina@fedora lab10]$ ./lab10-1
Введите x: 3
f(g(x))=23
[adutkina@fedora lab10]$ ./lab10-1
Введите x: 5
f(g(x))=35
[adutkina@fedora lab10]$
```

Рис. 2.4: Результат работы измененной программы

2.2 Отладка программ с помощью GDB

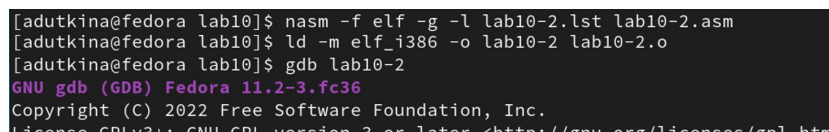
Создадим файл `lab10-2.asm` с текстом программы из Листинга 10.2 (рис. 2.5).

The screenshot shows a text editor window titled 'adutkina [Работает] - Oracle VM VirtualBox'. The menu bar includes 'Файл', 'Машина', 'Вид', 'Ввод', 'Устройства', and 'Справка'. The code is an x86 assembly program for NASM, starting with a comment in Russian: 'Программы вывода сообщения Hello world!'. It defines two data sections: 'msg1' containing 'Hello, ' and 'msg2' containing 'world!'. The main logic is in the '.text' section, starting at '_start', which moves the address of 'msg1' into 'ecx' and its length into 'edx', then prints it. It then repeats the same steps for 'msg2'. The program ends with a 'ret' instruction.

```
1 ;-----
2 ; Программы вывода сообщения Hello world!
3 ;-----
4
5 SECTION .data
6 msg1: db "Hello, ",0x0
7 msg1len: equ $ - msg1
8
9 msg2: db "world!",0xa
10 msg2len: equ $ - msg2
11
12 SECTION .text
13 global _start
14
15 _start:
16 mov eax, 4
17 mov ebx, 1
18 mov ecx, msg1
19 mov edx, msg1len
20 int 0x80
21
22 mov eax, 4
23 mov ebx, 1
24 mov ecx, msg2
25 mov edx, msg2len
26 int 0x80
27
28 mov eax, 1
29 mov ebx, 0
30 int 0x80
31
ret
```

Рис. 2.5: Программа печати сообщения Hello world!

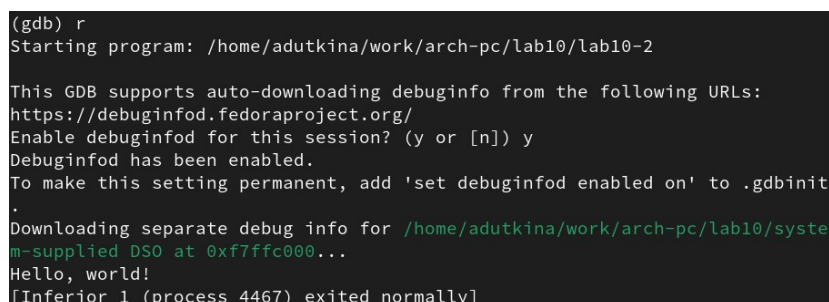
Получим исполняемый файл. Для работы с GDB в исполняемый файл добавим отладочную информацию, для этого трансляцию программ проводим с ключом '-g', запустим исполняемый файл в отладчик GDB(рис. 2.6)

The screenshot shows a terminal window with the following commands and output:

```
[adutkina@fedora lab10]$ nasm -f elf -g -l lab10-2.lst lab10-2.asm
[adutkina@fedora lab10]$ ld -m elf_i386 -o lab10-2 lab10-2.o
[adutkina@fedora lab10]$ gdb lab10-2
GNU gdb (GDB) Fedora 11.2-3.fc36
Copyright (C) 2022 Free Software Foundation, Inc.
License: GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

Рис. 2.6: Трансляция программы для работы с GDB

Проверим работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r) (рис. 2.7).

The screenshot shows a GDB terminal session. The user enters '(gdb) r', and the program starts. It prints 'Hello, world!' and then exits normally. The output includes information about debuginfo and the process ID.

```
(gdb) r
Starting program: /home/adutkina/work/arch-pc/lab10/lab10-2

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 /home/adutkina/work/arch-pc/lab10/system-supplied DSO at 0xf7ffc000...
Hello, world!
[Inferior 1 (process 4467) exited normally]
```

Рис. 2.7: Запуск программы в оболочке GDB

Для более подробного анализа программы установим брейкпоинт на метку

_start, с которой начинается выполнение любой ассемблерной программы, и запустим её. (рис. 2.8)

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 16.
(gdb) r
Starting program: /home/adutkina/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:16
16      mov eax, 4
```

Рис. 2.8: Запуск программы в оболочке GDB с подробным анализом

Посмотрим дисассимилированный код программы с помощью команды disassemble начиная с метки _start (рис. 2.9)

```
(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)
```

Рис. 2.9: Просмотр дисассимилированного кода программы

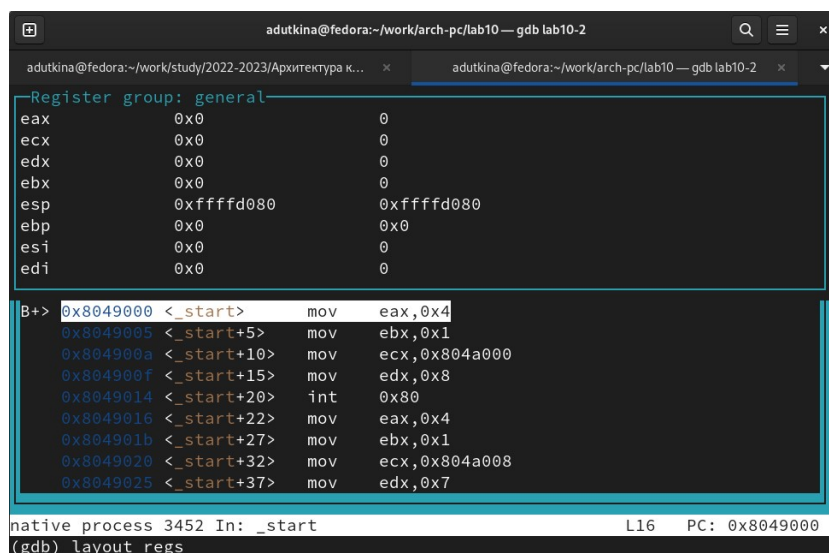
Переключимся на отображение команд с Intel'овским синтаксисом, введя команду set disassembly-flavor intel (рис. 2.10).

```
(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)
```

Рис. 2.10: Отображение команд с Intel'овским синтаксисом

Различия отображения синтаксиса машинных команд в режимах АТТ и Intel в том, что во втором варианте опускается ‘%’ перед именами регистров и инструкции с несколькими операндами перечисляются в разном порядке: Intel в прямом, то есть как записано в программе, а АТТ в обратном.

Включим режим псевдографики для более удобного анализа программы (рис. 2.11)



```
adutkina@fedora:~/work/arch-pc/lab10 — gdb lab10-2
adutkina@fedora:~/work/study/2022-2023/Архитектура к... x adutkina@fedora:~/work/arch-pc/lab10 — gdb lab10-2
Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd080 0xffffd080
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
B> 0x8049000 <_start> mov    eax,0x4
0x8049005 <_start+5> mov    ebx,0x1
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
0x8049025 <_start+37> mov    edx,0x7
native process 3452 In: _start L16 PC: 0x8049000
(gdb) layout regs
```

Рис. 2.11: Режим псевдографики

В этом режиме есть три окна: - В верхней части видны названия регистров и их текущие значения; - В средней части виден результат дисассимилирования программы; - Нижняя часть доступна для ввода команд.

2.2.1 Добавление точек останова

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»: на предыдущих шагах была установлена точка останова по имени метки (`_start`). Проверим

это с помощью команды `info breakpoints` (кратко `i b`) (рис. 2.12)

```
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x08049000 lab10-2.asm:16
breakpoint already hit 1 time
(gdb)
```

Рис. 2.12: Проверка установленной метки

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Установим точку останова для предпоследней инструкции (`mov ebx,0x0`), определив ее адрес (рис. 2.13)

```
0x804901b <_start+27> mov     ebx,0x1
0x8049020 <_start+32> mov     ecx,0x804a008
0x8049025 <_start+37> mov     edx,0x7
0x804902a <_start+42> int     0x80
0x804902c <_start+44> mov     eax,0x1
b+ 0x8049031 <_start+49> mov     ebx,0x0
0x8049036 <_start+54> int     0x80
0x8049038      add     BYTE PTR [eax],al
0x804903a      add     BYTE PTR [eax],al

native process 3452 In: _start L16 PC: 0x8049000
(gdb) break *0x8049031
Note: breakpoint 2 also set at pc 0x8049031.
Breakpoint 3 at 0x8049031: file lab10-2.asm, line 29.
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x08049000 lab10-2.asm:16
breakpoint already hit 1 time
2        breakpoint     keep y   0x08049031 lab10-2.asm:29
```

Рис. 2.13: Установка точки останова

2.2.2 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Выполним 5 инструкций с помощью команды `stepi` (или `si`). На этих шагах изменяются значения регистров `eax`, `ebx`, `ecx`, `edx` и еще раз `eax`.

Посмотреть содержимое регистров также можно с помощью команды `info registers` (или `i r`) (рис. 2.14)

```
native process 3560 In: _start L22 PC: 0x8049016
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd080 0xffffd080
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>
--Type <RET> for more, q to quit, c to continue without paging--
```

Рис. 2.14: Просмотр значения регистров

Для отображения содержимого памяти можно использовать команду `x`, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/NFU`. С помощью команды `x &` также можно посмотреть содержимое переменной.

Посмотрим значение переменной `msg1` по имени и значение переменной `msg2` по адресу (рис. 2.15).

```
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n\034"
(gdb)
```

Рис. 2.15: Просмотр значения переменной

Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си). Изменим первый символ переменной `msg1` (рис. 2.16)

```
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n\034"
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "
(gdb)
```

Рис. 2.16: Изменение символа в переменной `msg1`

Заменим второй символ в переменной `msg2` на заглавную букву (рис. 2.17).

```
(gdb) set {char}0x804a009='0'
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "wOrld!\n\034"
(gdb)
```

Рис. 2.17: Изменение символа переменной msg2

Чтобы посмотреть значения регистров используется команда print /F (перед именем регистра обязательно ставится префикс \$) (рис. 2.18)

```
(gdb) p/s $eax
$1 = 8
(gdb) p/t $eax
$2 = 1000
(gdb)
```

Рис. 2.18: Просмотр значения регистра

Выведем в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx. С помощью команды set изменим значение регистра ebx (рис. 2.19)

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$3 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$4 = 2
(gdb)
```

Рис. 2.19: Вывод значения регистра в различных форматах и их изменение

Завершим выполнение программы с помощью команды continue (сокращенно c) и выйдем из GDB с помощью команды quit (сокращенно q).

2.2.3 Обработка аргументов командной строки в GDB

Скопируем файл lab9-2.asm, созданный при выполнении лабораторной работы №9, с программой выводящей на экран аргументы командной строки в файл с именем lab10-3.asm и создадим исполняемый файл.

Для загрузки в gdb программы с аргументами необходимо использовать ключ -args. Загрузим исполняемый файл в отладчик, указав аргументы 1, 2, 3.

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Исследуем расположение

аргументов командной строки в стеке после запуска программы с помощью gdb.

Для начала установим точку останова перед первой инструкцией в программе и запустим ее (рис. 2.20)

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab10-3.asm, line 11.
(gdb) run
Starting program: /home/adutkina/work/arch-pc/lab10/lab10-3 1 2 3

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 /home/adutkina/work/arch-pc/lab10/system-supplied DSO at 0xf7ffc000...

Breakpoint 1, _start () at lab10-3.asm:11
11      pop ecx          ; Извлекаем из стека в `ecx` количество
(gdb)
```

Рис. 2.20: Работа с программой lab10-3

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы). Как видно, число аргументов равно 4 - расположение программы и три аргумента.

Посмотрим остальные позиции стека – по адресу [esp+4] располагается адрес в памяти где находится имя программы, по адресу [esp+8] храниться адрес первого аргумента, по адресу [esp+12] – второго и т.д. (рис. 2.21)

```
(gdb) x/x $esp
0xffffd060: 0x00000004
(gdb) x/s *(void**)($esp+4)
0xffffd21a: "/home/adutkina/work/arch-pc/lab10/lab10-3"
(gdb) x/s *(void**)($esp+8)
0xffffd244: "1"
(gdb) x/s *(void**)($esp+12)
0xffffd246: "2"
(gdb) x/s *(void**)($esp+16)
0xffffd248: "3"
(gdb) x/s *(void**)($esp+20)
0x0: <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 2.21: Просмотр позиций стека

3 Выводы

В ходе лабораторной работы были приобретены навыки написания программ с использованием подпрограмм и изучены методы отладки при помощи GDB и его основные возможности.