**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Факультет физико-математических и естественных наук**

**Кафедра прикладной информатики и теории вероятностей**

# ОТЧЕТ
# ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9.

дисциплина: *Архитектура компьютеров*

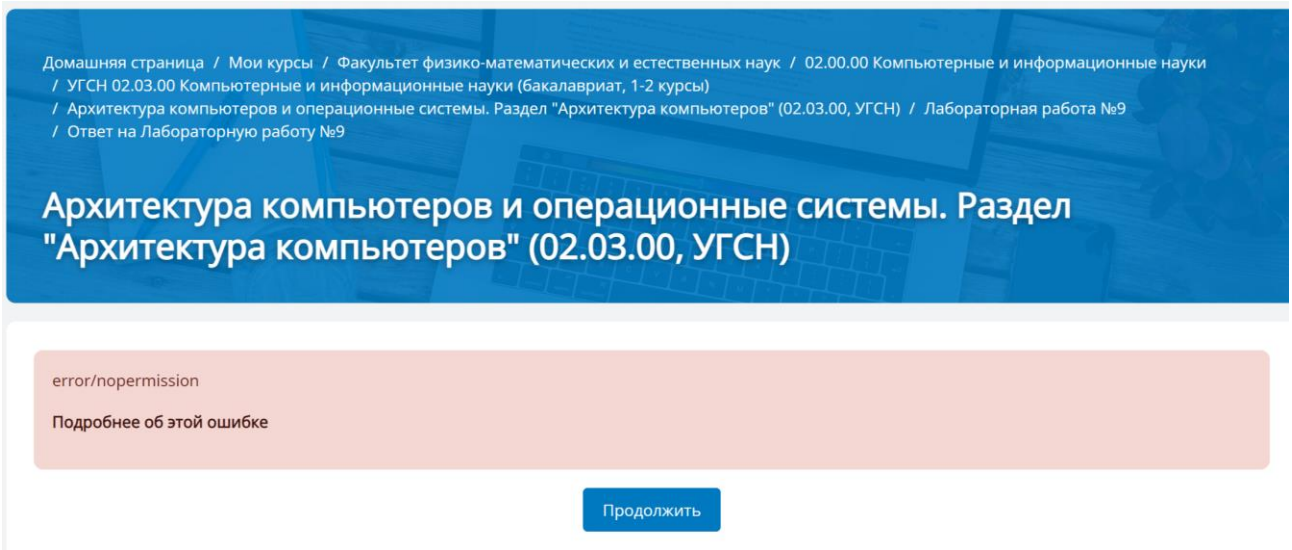
Студент: Подхалюзина Виолетта Михайловна

Группа: НКАбд-04-24

**МОСКВА**

2024 г.

# Оглавление




Прошу обратить ваше внимание. Сайт не давал мне возможности загрузить задание до окончания дедлайна. Я не знаю, с чем это связано, но надеюсь на ваше понимание.

# 1 Цель работы

Приобрести навыки написания программ с использованием подпрограмм. Познакомиться с методами отладки при помощи GDB и его основными возможностями.

# 2 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.

# 3 Выполнение лабораторной работы

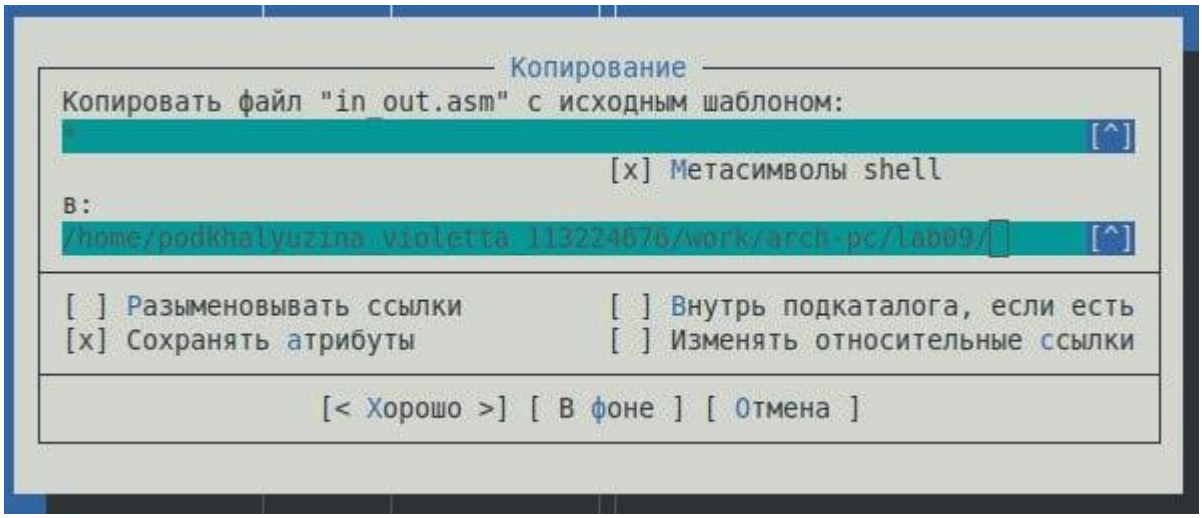
## 3.1 Начало работы

Я создаю каталог для программ лабораторной работы №9, перехожу в него и создаю файл lab9-1.asm

```
podkhalyuzina_violetta_113224676@violetta-Mint:~$ mkdir ~/work/arch-pc/lab09
podkhalyuzina_violetta_113224676@violetta-Mint:~$ cd ~/work/arch-pc/lab09
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ touch lab09-1.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ls
lab09-1.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

Далее я ввела в файл lab9-1.asm текст программы из листинга 9.1. Создала исполняемый файл и проверила его работу

```
 GNU nano 7.2
%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:
;-----------------------------------------
; Основная программа
;-----------------------------------------
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax,x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax,result
call sprint
mov eax,[res]
call iprintLF
call quit
;-----------------------------------------
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из подпрограммы
```




```
podkhalyuzina_violetta_1132246676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
podkhalyuzina_violetta_1132246676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
podkhalyuzina_violetta_1132246676@violetta-Mint:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 5
2x+7=17
podkhalyuzina_violetta_1132246676@violetta-Mint:~/work/arch-pc/lab09$
```

Далее я изменила текст программы, добавив подпрограмму _subcalcul в подпрограмму _calcul, для вычисления выражения □(□(□)), где □ вводится с клавиатуры, □(□) = 2□ + 7, □(□) = 3□ − 1. Т.е. □ передается в подпрограмму _calcul из нее в подпрограмму _subcalcul, где вычисляется выражение □(□), результат возвращается в _calcul и вычисляется выражение

```
 GNU nano 7.2
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2(3x-1)+7=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:
;-------------------------------------
; Основная программа
;-------------------------------------
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax,x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax,result
call sprint
mov eax,[res]
call iprintLF
call quit
;-------------------------------------
; Подпрограмма вычисления
; выражения
_calcul:
push eax
call _subcalcul
mov ebx,2
mul ebx
add eax,7
mov [res],eax
pop eax
ret ; выход из подпрограммы
_subcalcul:
mov ebx, 3
mul ebx
sub eax, 1
ret
```

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 5
2(3x-1)+7=35
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

Я создала файл lab09-2.asm с текстом программы из Листинга 9.2

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ touch lab09-2.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ls
in_out.asm  lab09-1  lab09-1.asm  lab09-1.o  lab09-2.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

```
 GNU nano 7.2
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
```

Для работы с GDB в исполняемый файл я добавила отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’, а после загрузила исполняемый файл в отладчик gdb.

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-2 lab09-2.o
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ 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)
(gdb) run
Starting program: /home/podkhalyuzina_violetta_113224676/work/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 5476) exited normally]
(gdb)
```

Далее провесила работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r). Для более подробного анализа программы установила брейкпоинт на метку _start, с которой начинается выполнение любой ассемблерной программы, и запустила её

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/podkhalyuzina_violetta_113224676/work/arch-pc/lab09/lab09-2

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

После я посмотрела дисассимилированный код программы с помощью команды disassemble начиная с метки _start

```
(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)
```

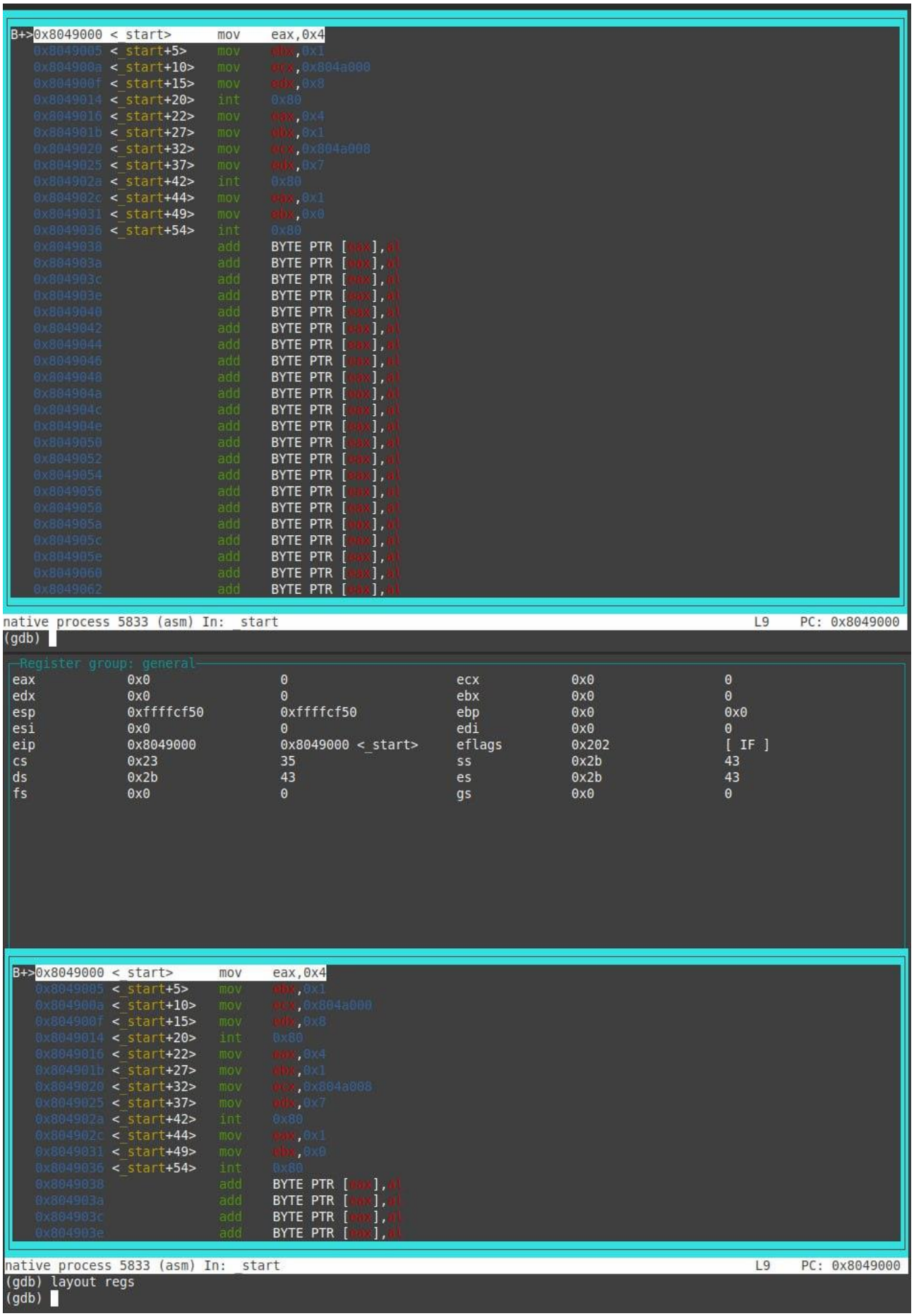
И переключилась на отображение команд с Intel’овским синтаксисом, введя команду set disassembly-flavor intel

```
(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)
```

Различия между синтаксисом в ATT - размер операндов указывается явно с помощью суффиксов, непосредственные операнды предваряются символом $, а у Intel размер операндов неявно определяется контекстом, как ax, eax, непосредственные операнды пишутся напрямую, в ATT имена регистров предваряются символом %, у Intel - имена регистров пишутся без префиксов.

После я включила режим псевдографики для более удобного анализа

программы




На предыдущих шагах была установлена точка останова по имени метки (_start). Проверила это с помощью команды info breakpoints

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

Использую адрес предпоследней инструкции и смотрю информацию о всех установленных точках останова

```
(gdb) b *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)
```

После я посмотрела содержимое регистров также можно с помощью команды info registers (или i r)

```
(gdb) info registers
eax            0x4                 4
ecx            0x0                 0
edx            0x0                 0
ebx            0x0                 0
esp            0xffffcf50          0xffffcf50
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0x8049005           0x8049005 <_start+5>
eflags         0x10202             [ IF RF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x0                 0
(gdb)
```

Для отображения содержимого памяти можно использовать команду x <адрес>, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: x/NFU <адрес>. С помощью команды x &<имя переменной> также можно посмотреть содержимое переменной. Я посмотрела значение переменной msg1 по имени

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

Далее я изменила первый символ переменной msg1

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

Я вывела в различных форматах значение регистра edx. С помощью команды set изменила значение регистра ebx

```
(gdb) p/t $eax
$5 = 100
(gdb) p/s $eax
$6 = 4
(gdb) p/x $eax
$7 = 0x4
(gdb)
```

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$8 = 50
(gdb)
```

```
(gdb) set $ebx=2
(gdb) p/s $ebx
$9 = 2
```

В первом случае выводится порядковый номер символа '2' в таблице ASCII, а во втором случае, так как 2 задана цифрой, а не символом, то и выводится цифра.

Далее я завершила выполнение программы с помощью команды continue (сокращенно c) и вывела из GDB с помощью команды quit (сокращенно q)

```
(gdb) c
Continuing.
hello, xorld!

Breakpoint 2, _start () at lab09-2.asm:20
(gdb)
```

```
(gdb) q
A debugging session is active.

        Inferior 1 [process 5833] will be killed.

Quit anyway? (y or n)
```

После я скопировала файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки  в файл с именем lab09-3.asm

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ cp ~/work/arch-pc/lab08/lab08-2.asm ~/work/arch-p
c/lab09/lab09-3.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ls
in_out.asm  lab09-1  lab09-1.asm  lab09-1.o  lab09-2  lab09-2.asm  lab09-2.lst  lab09-2.o  lab09-3.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

Далее создаю исполняемый файл

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-3 lab09-3.o
```

Для загрузки в gdb программы с аргументами необходимо использовать ключ --args. А также я загрузила исполняемый файл в отладчик, указав аргументы

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ gdb --args lab09-3 2 5 'test'
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 lab09-3...
(gdb)
```

Исследую расположение аргументов командной строки в стеке после запуска программы с помощью gdb. Для начала устанавливаю точку останова перед первой инструкцией в программе и запускаю ее

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/podkhalyuzina_violetta_113224676/work/arch-pc/lab09/lab09-3 2 5 test

Breakpoint 1, _start () at lab09-3.asm:5
5       pop ecx ; Извлекаем из стека в `ecx` количество
(gdb)
```

Адрес вершины стека хранится в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки

```
(gdb) x/x $esp
0xffffcf30:     0x00000004
(gdb)
```

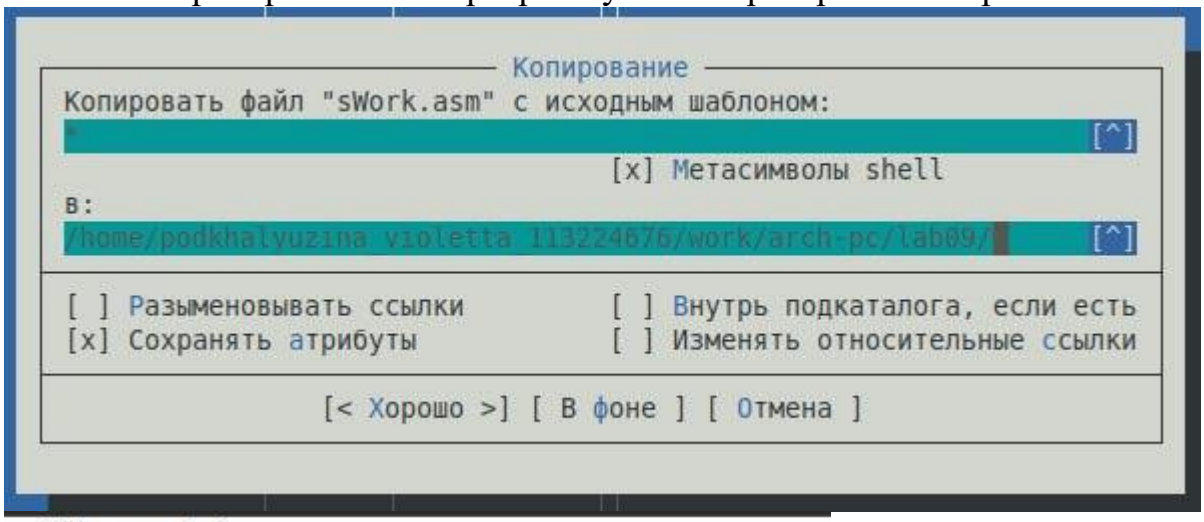
Далее я смотрю остальные позиции стека – по адесу [esp+4] располагается адрес в памяти, где находится имя программы, по адесу [esp+8] храниться адрес первого аргумента, по аресу [esp+12] – второго и так далее

```
(gdb) x/s *(void**)($esp + 4)
0xffffd0fe:     "/home/podkhalyuzina_violetta_113224676/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd140:     "2"
(gdb) x/s *(void**)($esp + 12)
0xffffd142:     "5"
(gdb) x/s *(void**)($esp + 16)
0xffffd144:     "test"
```

Изменение аргумента команды просмотра регистра esp на +4, число обусловлено разрядностью системы, а указатель void занимает как раз 4 байта, ошибка при аргументе +24 означает, что аргументы на вход программы закончились.

### 3.2 Самостоятельная работа

Сначала я преобразовываю программу из лабораторной номер 8.




```
 GNU nano 7.2
%include 'in_out.asm'
SECTION .data
msg db "Результат: ",0
form db "Формула 3x - 1",0
SECTION .text
global _start
_start:
pop ecx ; Извлекаем из стека в `ecx` количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в `edx` имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем `ecx` на 1 (количество
; аргументов без названия программы)
mov esi, 0
mov eax, form
call sprintLF
mov ebx, 3
next:
cmp ecx,0h
jz _end
pop eax
call atoi
call _func
add esi,eax
loop next
_end:
mov eax, msg ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр `eax`
call iprintLF ; печать результата
call quit ; завершение программы
_func:
mul ebx
sub eax, 1
ret
```

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf sWork.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o sWork sWork.o
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ./sWork 1 2 3
Формула 3x - 1
Результат: 15
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

В листинге 9.3 приведена программа вычисления выражения (3 + 2) ∗ 4 + 5. При запуске данная программа дает неверный результат и я это проверила.

```
 GNU nano 7.2
%include 'in_out.asm'
SECTION .data
div: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:
; ---- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add ebx,eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; ---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit
```

После этого с помощью отладчика GDB, анализируя изменения значений регистров, я определила ошибку и исправила ее.

```
0x80490e8 <_start>      mov    ebx,0x3
0x80490ed <_start+5>    mov    eax,0x2
0x80490f2 <_start+10>   add    ebx,eax
0x80490f4 <_start+12>   mov    ecx,0x4
0x80490f9 <_start+17>   mul    ecx
0x80490fb <_start+19>   add    ebx,0x5
0x80490fe <_start+22>   mov    edi,ebx
0x8049100 <_start+24>   mov    eax,0x804a000
0x8049105 <_start+29>   call   0x804900f <sprint>
0x804910a <_start+34>   mov    eax,edi
0x804910c <_start+36>   call   0x8049086 <iprintLF>
0x8049111 <_start+41>   call   0x80490db <quit>
```

```
 GNU nano 7.2
%include 'in_out.asm'
SECTION .data
div: DB 'Результат: ', 0
SECTION .text
GLOBAL _start
_start:
mov ebx, 3
mov eax, 2
add ebx, eax
mov eax, ebx
mov ecx, 4
mul ecx
add eax, 5
mov edi, eax
mov eax, div
call sprint
mov eax, edi
call iprintLF
call quit
```

```
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ nasm -f elf sWork2.asm
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ld -m elf_i386 -o sWork2 sWork2.o
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$ ./sWork2
Результат: 25
podkhalyuzina_violetta_113224676@violetta-Mint:~/work/arch-pc/lab09$
```

Вывод: я преобрела навыки написания программ с использованием подпрограмм, а также познакомилась с методами отладки при помощи GDB и его основными возможностями.

## 4 Контрольные вопросы для самопроверки

1. В ассемблере для оформления и активизации подпрограмм используются инструкции call и ret. Call вызывает подпрограмму, сохраняя адрес возврата в стеке, а ret завершает выполнение подпрограммы, извлекая этот адрес из стека и возвращая управление основной программе.

2. Механизм вызова подпрограмм состоит в том, что инструкция call сохраняет адрес следующей команды (адрес возврата) в стек и передает управление указанной подпрограмме. После завершения выполнения подпрограммы инструкция ret извлекает этот адрес из стека и возвращает управление основной программе, начиная выполнение с команды, следующей за вызовом call.

3. Стек используется для сохранения адреса возврата при вызове подпрограммы. Если подпрограмма сохраняет дополнительные данные в стек, важно восстановить исходное состояние стека перед завершением выполнения, иначе поврежденный адрес возврата приведет к ошибке.

4. Операнд в команде ret определяет количество байт, которые нужно удалить из стека после извлечения адреса возврата. Это используется для очистки параметров вызова, переданных через стек.

5. Отладчик предназначен для поиска и исправления ошибок в программе. Он позволяет выполнять программу по шагам, устанавливать точки останова, отслеживать значения переменных и регистров, а также изменять данные программы во время её выполнения.

6. Отладочная информация связывает исполняемый код с исходным текстом программы, что позволяет анализировать её на уровне строк исходного кода. Для её включения программу нужно компилировать с использованием ключа -g.

7. Breakpoint — это точка останова, при достижении которой выполнение программы приостанавливается. Watchpoint — это точка просмотра, выполнение программы останавливается при изменении или чтении указанной переменной. Checkpoint — это сохраненное состояние программы, которое можно восстановить. Catchpoint — это точка перехвата, срабатывающая при определенных событиях, например, исключениях. Call stack — это стек вызовов, отображающий текущую цепочку активных функций или процедур.

8. Основные команды отладчика gdb включают run (r) для запуска программы, break (b) для установки точки останова, continue (c) для

продолжения выполнения после останова, step (s) и next (n) для пошагового выполнения программы, info (i) для получения информации о состоянии программы, print (p) для вывода значений переменных, quit (q) для выхода из отладчика.

# 5 Список литературы


1. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/.

2. GNU Bash Manual. — 2016. — URL: https://www.gnu.org/software/bash/manual/.

3. Midnight Commander Development Center. — 2021. — URL: https://midnight-commander. org/.

4. NASM Assembly Language Tutorials. — 2021. — URL: https://asmtutor.com/.

5. Newham C. Learning the bash Shell: Unix Shell Programming. — O'Reilly Media, 2005. — 354 c. — (In a Nutshell). — ISBN 0596009658. — URL: http://www.amazon.com/Learningbash-Shell-Programming-Nutshell/dp/0596009658.

6. Robbins A. Bash Pocket Reference. — O'Reilly Media, 2016. — 156 c. — ISBN 978-1491941591.

7. The NASM documentation. — 2021. — URL: https://www.nasm.us/docs.php.

8. Zarrelli G. Mastering Bash. — Packt Publishing, 2017. — 502 c. — ISBN 9781784396879.

9. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.

10. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.

11. Новожилов О. П. Архитектура ЭВМ и систем. — М. : Юрайт, 2016.

12. Расширенный ассемблер: NASM. — 2021. — URL: https://www.opennet.ru/docs/RUS/nasm/.

13. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е изд. — БХВПетербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.

14. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011. — URL:

http://www.stolyarov.info/books/asm_unix.

15. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).

16. Таненбаум Э., Бос Х. Современные операционные системы. — 4-е изд. — СПб. : Питер, 2015. — 1120 с. — (Классика Computer Science).