Отчёта по лабораторной работе №9

НПИ-03-23

Махмудов Суннатилло Баходир угли

Содержание

Список иллюстраций

Список таблиц

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. Работа с данными программы в GDB.
4. Обработка аргументов командной строки в GDB.
5. 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

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.

4 Выполнение лабораторной работы

Создадим каталог и файл для лабораторной работы(рис. 4.15).

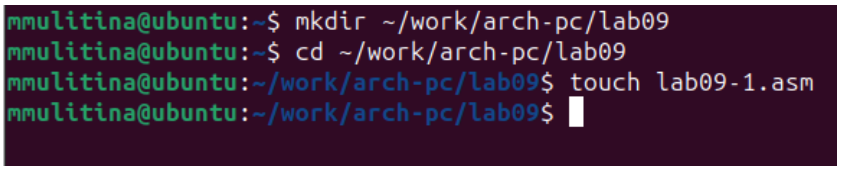


Рис. 4.1: Создание каталога

Введём текст программы и запустим её для проверки(рис. 4.15).

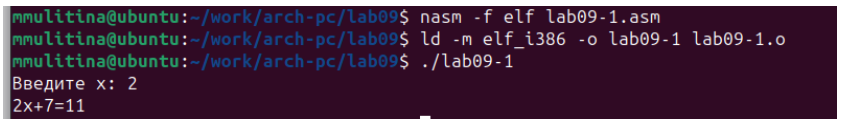


Рис. 4.2: Программа

Добавим подпрограмму _subcalcul, запустим программу для проверки(рис. 4.15).

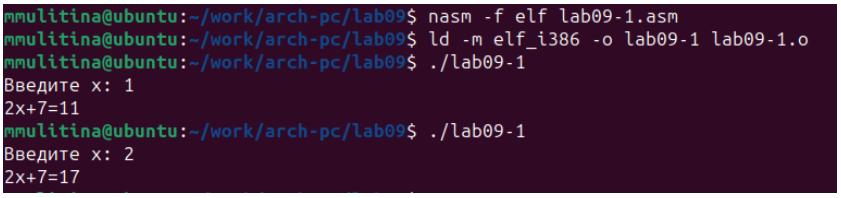


Рис. 4.3: Программа

Создадим файл lab09-2.asm, введём в него текст программы, получим исполняемый

файл и загрузим его в отладчик(рис. 4.15).

Рис. 4.4: Отладчик

Проверим работу программы, запустим ее в оболочке GDB(рис. 4.15).

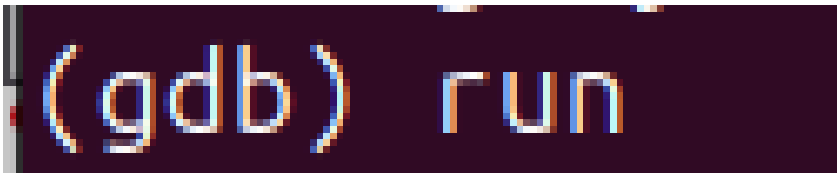


Рис. 4.5: Отладчик

(рис. 4.15).

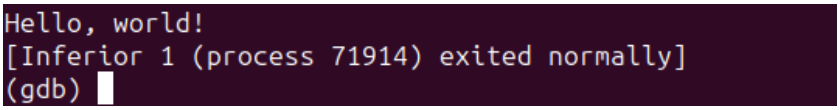


Рис. 4.6: Отладчик

Установим брейкпоинт и запустим программу(рис. 4.15).

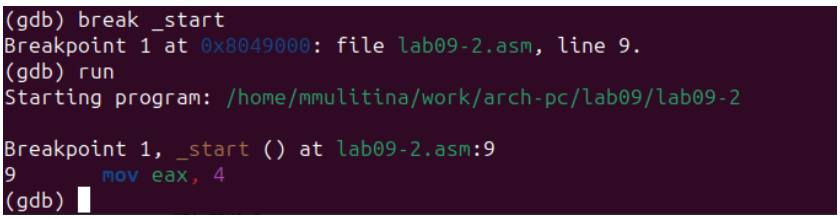


Рис. 4.7: Брейкпоинт

Посмотрим дисассимилированный код (рис. 4.15).

Рис. 4.8: Дисассимилированный код

Переключимся на отображение команд с Intel синтаксисом (рис. 4.15).

Рис. 4.9: Отображение

Включим режим псевдографики (рис. 4.15).

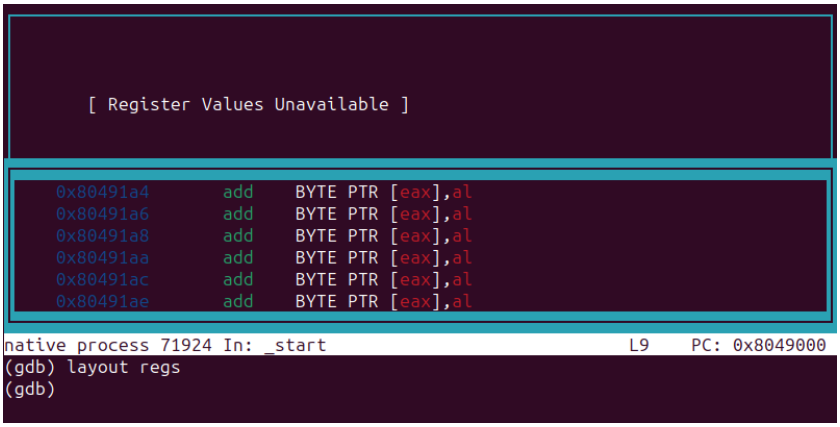


Рис. 4.10: Псевдографика

Проверим точки останова с помощью команды info breakpoints (рис. 4.15).

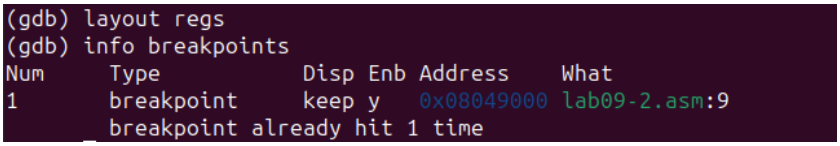


Рис. 4.11: Точки останова

Установим ещё одну точку останова и снова посмотрим информацию о точках останова (рис. 4.15).

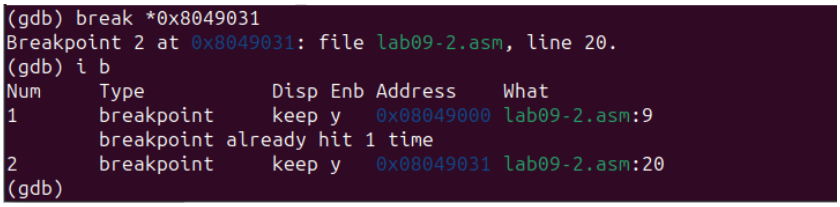


Рис. 4.12: Точки останова

Выполним 5 инструкций с помощью команды stepi и проследим изменения регистров(рис. 4.15).

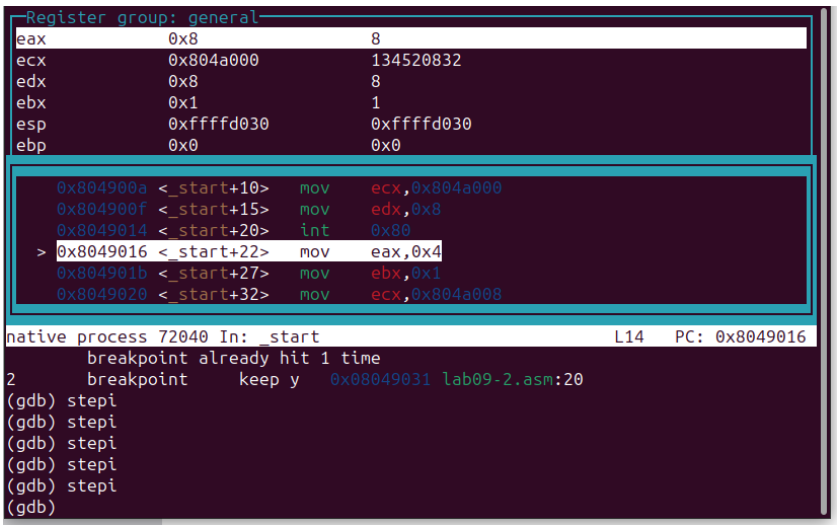


Рис. 4.13: stepi

Изменились значения регистров eax, ecx, ebx, edx.

Посмотрим значения регистров с помощью info registers(рис. 4.15).

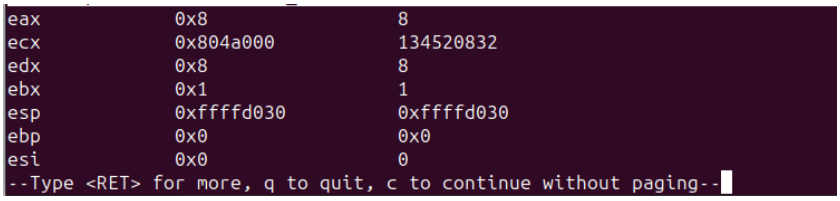
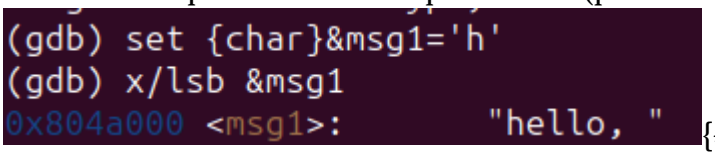


Рис. 4.14: Значения регистров

Посмотрим значение переменной msg1 по имени(рис. 4.15).

Рис. 4.15: Значение переменной

Изменим первый символ переменной (рис. 4.15).



{#fig:001 width=70%

Изменим любой символ второй переменной msg2 (рис. 4.15).

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

{#fig:001 width=70%

Посмотрим значений регистра edx (рис. 4.15).

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

{#fig:001 width=70%

С помощью set изменим значение регистра ebx (рис. 4.15).

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

{#fig:001 width=70%

Проверим его значение(рис. 4.15).

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

{#fig:001 width=70%

Снова изменим значение ebx(рис. 4.15).

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

{#fig:001 width=70%

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

Скопируем файл из прошлой лабораторной работы(рис. 4.15).

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

{#fig:001 width=70%

Создадим исполняемый файл (рис. 4.15).

```
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
```

{#fig:001 width=70%

Загрузим исполняемый файл в отладчик, указав аргументы(рис. 4.15).

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

{#fig:001 width=70%

Установим точку останова перед первой инструкцией в программе и запустим

её(рис. 4.15).

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

{#fig:001 width=70%

Посмотрим значение регистра esp, где хранится адрес вершины стека(рис. 4.15).

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

{#fig:001 width=70%

Посмотрим остальные позиции стека по адресу (рис. 4.15).

```
(gdb) x/s *(void**)($esp + 4)
0xffffd1f4:     "/home/mmulitina/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)($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>
```

{#fig:001 width=70%

Шаг изменения равен 4, т.к. у нас 4 аргумента.(рис. 4.15).

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

{#fig:001 width=70%

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

5 4.1 Задания для самостоятельной работы

1. Преобразуем программу из лабораторной работы №8, реализовав вычисление значения функции как подпрограмму (рис. 4.15).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./prog1 1 2 3 4
Результат: 80
```
{#fig:001 width=70%

Запустим программу для проверки(рис. 4.15).

```
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
```
{#fig:001 width=70%

2. Создадим файл для программы, введём в него текст программы, запустим его в отладчике GDB (рис. 4.15).

```
eax            0x8                8
ecx            0x4                4
edx            0x0                0
ebx            0x5                5
```
{#fig:001 width=70%

При умножении с помощью mul, мы умножаем eax на ecx b записываем в eax. Получаем 24=9 вместо (3+2)4(рис. 4.15).

```
mmulitina@ubuntu:~/work/arch-pc/lab09$ ./lab09-4
Результат: 10
```
{#fig:001 width=70%

Потом скалдываем с регистром ebx 5 и получаем 10. Проверим это, запустив программу(рис. 4.15).

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

{#fig:001 width=70%

Исправим программу(рис. 4.15).

```
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$
```

{#fig:001 width=70%

Запустим её для проверки (рис. 4.15).

```
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
```

{#fig:001 width=70%

6 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

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 с. — (In a Nutshell). — ISBN 0596009658. — URL: http://www.amazon.com/Learning- bash-Shell-Programming-Nutshell/dp/0596009658.
6. Robbins A. Bash Pocket Reference. — O'Reilly Media, 2016. — 156 с. — ISBN 978-1491941591.
7. The NASM documentation. — 2021. — URL: https://www.nasm.us/docs.php.
8. Zarrelli G. Mastering Bash. — Packt Publishing, 2017. — 502 с. — 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,
11.
12. Новожилов О. П. Архитектура ЭВМ и систем. — М. : Юрайт, 2016.
13. Расширенный ассемблер: NASM. — 2021. — URL: https://www.opennet.ru/docs/RUS/nasm/.
14. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е изд. — БХВ- Петербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.
15. 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.
16. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).
17. Таненбаум Э., Бос Х. Современные операционные системы. — 4-е изд. — СПб. : Питер,
18. — 1120 с. — (Классика Computer Science).