10. Лабораторная работа №10. Понятие подпрограммы. Отладчик GDB.

10.1. Цель работы

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

10.2. Теоретическое введение

10.2.1. 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.

10.2.2. 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.

10.3. Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) [1] 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.

10.3.1. 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.

Далее приведён список некоторых команд GDB.

Команда `run` (сокращённо `r`) — запускает отлаживаемую программу в оболочке GDB.

Если точки останова не были установлены, то программа выполняется и выводятся сообщения:

```
(gdb) run
Starting program: test
Program exited normally.
(gdb)
```

Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др.

Команда `kill` (сокращённо `k`) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки:

```
Kill the program being debugged? (y or n) y
```

Если в ответ введено `y` (то есть «да»), отладка программы прекращается. Командой `run` её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.

Для выхода из отладчика используется команда `quit` (или сокращённо `q`):

```
(gdb) q
```

10.3.2. 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`.

Посмотреть дизассемблированный код программы можно с помощью команды `disassemble <метка/адрес>`:

```
(gdb) disassemble _start
```

Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду `set disassembly-flavor intel`

10.3.3. 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 *<адрес>
(gdb) b <метка>
```

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 <номер точки останова>
```

Обратно точка останова активируется командой `enable`:

```
enable breakpoint <номер точки останова>
```

Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`:

```
(gdb) delete breakpoint <номер точки останова>
```

Ввод этой команды без аргумента удалит все точки останова.

Информацию о командах этого раздела можно получить, введя

```
help breakpoints
```

10.3.4. Пошаговая отладка

Для продолжения остановленной программы используется команда `continue` (с) `(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 `nexti` (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
```

10.3.5. Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Посмотреть содержимое регистров можно с помощью команды `info registers` (или `i r`):

```
(gdb) info registers
```

Для отображения содержимого памяти можно использовать команду `x/NFU <адрес>`, выдаёт содержимое ячейки памяти по указанному адресу. `NFU` задает формат, в котором выводятся данные (см. Таблицу 10.1).

Таблица 10.1. Формат отображения данных команды `x`.

	Значение	Описание
`N`	Десятичное целое число	**Счётчик повторений**. Определяет, сколько ячеек памяти отобразить (считая в единицах), по умолчанию 1.
`F`	**Формат отображения**	
	`s`	строка, оканчивающаяся нулём
	`i`	машинная инструкция
	`x`	шестнадцатеричное число
	`a`	адрес
`U`	**Размер отображаемых ячеек памяти**	
	`b`	байт
	`h`	полуслово, 2 байта
	`w`	машинное слово, 4 байта (значение по умолчанию)
	`g`	длинное слово, 8 байт

Например, `x/4uh 0x63450` — это запрос на вывод четырёх полуслов (`h`) из памяти в формате беззнаковых десятичных целых (`u`), начиная с адреса `0x63450`.

Чтобы посмотреть значения регистров используется команда `print /F <val>`

(сокращенно p). Перед именем регистра обязательно ставится префикс $. Например, команда p/x $ecx выводит значение регистра в шестнадцатеричном формате.

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).

Справку о любой команде gdb можно получить, введя

```
(gdb) help [имя_команды]
```

10.3.6. 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.

10.3.6.1. 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 eip 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

eip. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией call.

Подпрограмма может вызываться как из внешнего файла, так и быть частью основной программы.

Основные моменты выполнения подпрограммы иллюстрируются на рис. 10.1.

Рис. 10.1. Основные моменты выполнения подпрограммы

Важно помнить, что если в подпрограмме занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата и это приведёт к ошибке выхода из подпрограммы. Кроме того, надо помнить, что подпрограмма без команды возврата не вернётся в точку вызова, а будет выполнять следующий за подпрограммой код, как будто он является её продолжением.

10.4. Порядок выполнения лабораторной работы

10.4.1. Реализация подпрограмм в NASM

1. Создайте каталог для выполнения лабораторной работы № 10, перейдите в него и создайте файл lab10-1.asm:

```
mkdir  ~/work/arch-pc/lab10
cd  ~/work/arch-pc/lab10
touch lab10-1.asm
```

2. 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 изучите текст программы (Листинг 10.1).

Листинг 10.1. Пример программы с использованием вызова подпрограммы

```
%include    'in_out.asm'

  SECTION .data
        msg:      DB 'Введите x: ',0
        result:   DB '2x+7=',0

  SECTION  .bss
        x:      RESB 80
        rezs:   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 [rez],eax

           ret         ; выход из 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`).

```
   mov  eax, msg     ; вызов подпрограммы печати сообщения
```

```
call sprint        ; 'Введите x: '

mov  ecx, x
mov  edx, 80
call sread         ; вызов подпрограммы ввода сообщения

mov  eax,x         ; вызов подпрограммы преобразования
call atoi          ; ASCII кода в число, `eax=x`
```

После следующей инструкции `call _calcul`, которая передает управление подпрограмме `_calcul`, будут выполнены инструкции подпрограммы:

```
mov ebx,2
mul ebx
add eax,7
mov [rez],eax

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 реализую вывод сообщения (`call sprint`), результата вычисления (`call iprintLF`) и завершение программы (`call quit`).

Введите в файл `lab10-1.asm` текст программы из листинга 10.1. Создайте исполняемый файл и проверьте его работу.

Измените 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 в 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.

10.4.2. Отладка программам с помощью GDB

Создайте файл `lab10-2.asm` с текстом программы из Листинга 10.2. (Программа печати сообщения `Hello world!`):

Листинг 10.2. Программа вывода сообщения `Hello world!`

```
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
```

Получите исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом '-g'.

```
nasm -f elf -g -l lab10-2.lst lab10-2.asm
ld -m elf_i386 -o lab10-2 lab10-2.o
```

Загрузите исполняемый файл в отладчик gdb:

```
user@dk4n31:~$ gdb lab10-2
```

Проверьте работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r):

```
(gdb) run
Starting program: ~/work/arch-pc/lab10/lab10-2
Hello, world!
[Inferior 1 (process 10220) exited normally]
(gdb)
```

Для более подробного анализа программы установите брейкпоинт на метку _start, с которой начинается выполнение любой ассемблерной программы, и запустите её.

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 12.
(gdb) run
Starting program: ~/work/arch-pc/lab10/lab10-2
Breakpoint 1, _start () at lab10-2.asm:12
12              mov eax, 4
```

Посмотрите дисассимилированный код программы с помощью команды disassemble начиная с метки _start

```
(gdb) disassemble _start
```

Переключитесь на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel`

```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
```

Перечислите различия отображения синтаксиса машинных команд в режимах ATT и Intel.

Включите режим псевдографики для более удобного анализа программы (рис. 10.2):

```
(gdb) layout asm
(gdb) layout regs
```

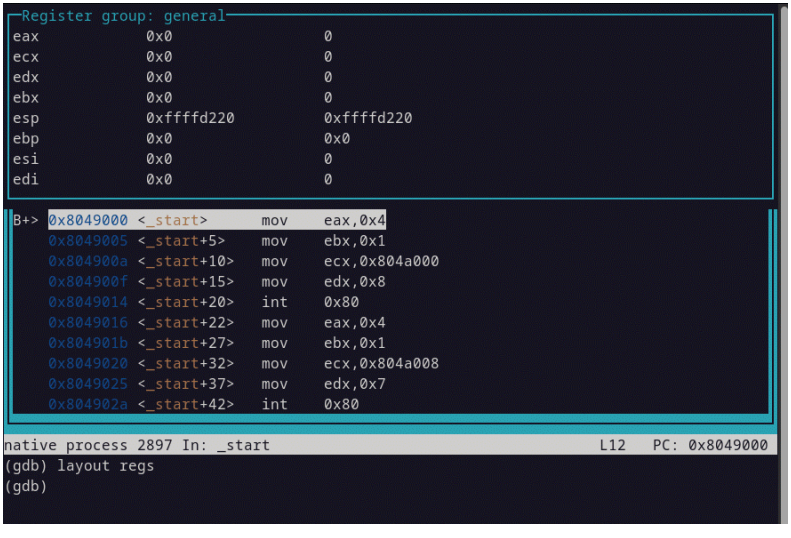

Рис. 10.2. Режим псевдографики gdb

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.

10.4.2.1. 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`). Проверьте это с помощью команды `info breakpoints` (кратко `i b`):

```
(gdb) info breakpoints
```

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции (см. рис. 10.3). Определите адрес предпоследней инструкции (`mov ebx,0x0`) и установите точку останова.

```
(gdb) break *<адрес>
```

Посмотрите информацию о всех установленных точках останова:

```
(gdb) i b
```

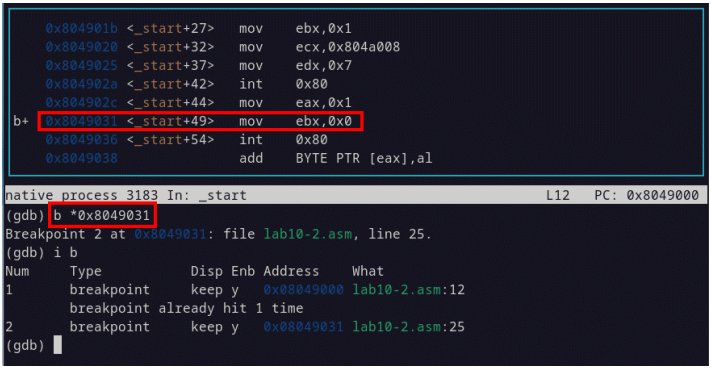

Рис. 10.3. Установка точки останова по адресу инструкции

10.4.2.2. Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Выполните 5 инструкций с помощью команды `stepi` (или `si`) и проследите за изменением значений регистров. Значения каких регистров изменяются?

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

```
(gdb) info registers
```

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.

Посмотрите значение переменной `msg1` по имени

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

Посмотрите значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрите инструкцию mov ecx,msg2 которая записывает в регистр ecx адрес перемененной msg2 (рис. 10.4).

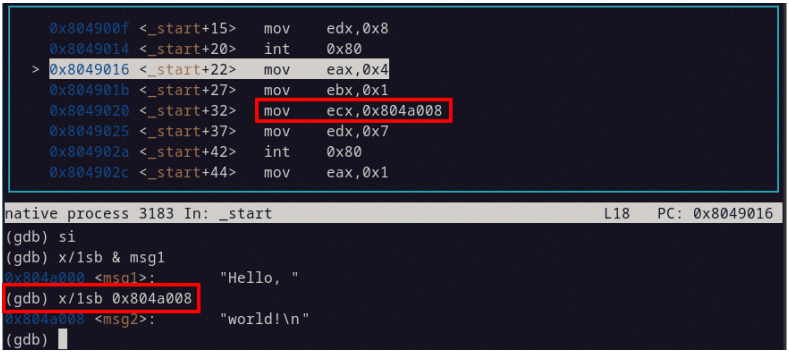

Рис. 10.4. Отображение содержимого памяти

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). Измените первый символ переменной msg1 (рис. 10.5):

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

Рис. 10.5. Примеры использования команды `set`

Замените любой символ во второй переменной `msg2`.

Чтобы посмотреть значения регистров используется команда `print /F <val>` (перед именем регистра обязательно ставится префикс $) (рис. 10.6):

```
p/F $<регистр>
```

```
Register group: general
eax            0x4                4
ecx            0x804a000          134520832
edx            0x8                8
ebx            0x1                1
esp            0xffffd220         0xffffd220
ebp            0x0                0x0
esi            0x0                0
edi            0x0                0
eip            0x804901b          0x804901b <_start+27>
native process 3183 In: _start
(gdb) p/s $eax
$2 = 4
(gdb) p/t $eax
$3 = 100
(gdb) p/s $ecx
$4 = 134520832
(gdb) p/x $ecx
$5 = 0x804a000
(gdb)
```

Рис. 10.6. Примеры использования команды `print`

Выведете в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра `edx`.

С помощью команды `set` измените значение регистра `ebx`:

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

```
(gdb) p/s $ebx
$4 = 2
(gdb)
```

Объясните разницу вывода команд `p/s $ebx`.

Завершите выполнение программы с помощью команды `continue` (сокращенно `c`) или `stepi` (сокращенно `si`) и выйдите из GDB с помощью команды `quit` (сокращенно `q`).

10.4.2.3. Обработка аргументов командной строки в GDB

Скопируйте файл `lab9-2.asm`, созданный при выполнении лабораторной работы №9, с программой выводящей на экран аргументы командной строки (Листинг 9.2) в файл с именем `lab10-3.asm`:

```
cp ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
```

Создайте исполняемый файл.

```
nasm -f elf -g -l lab10-3.lst lab10-3.asm
ld -m elf_i386 -o lab10-3 lab10-3.o
```

Для загрузки в `gdb` программы с аргументами необходимо использовать ключ `--args`. Загрузите исполняемый файл в отладчик, указав аргументы:

```
gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 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.

```
(gdb) b _start
(gdb) run
```

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы):

```
(gdb)  x/x $esp
0xffffd200:     0x05
```

Как видно, число аргументов равно 5 – это имя программы lab10-3 и непосредственно аргументы: аргумент1, аргумент, 2 и 'аргумент 3'.

Посмотрите остальные позиции стека – по адесу [esp+4] располагается адрес в памяти где находиться имя программы, по адесу [esp+8] храниться адрес первого аргумента, по аресу [esp+12] – второго и т.д.

```
(gdb) x/s *(void**)($esp + 4)
0xffffd358:     "~/lab10-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd3bc:     "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xffffd3ce:     "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xffffd3df:     "2"
(gdb) x/s *(void**)($esp + 20)
0xffffd3e1:     "аргумент 3"
(gdb) x/s *(void**)($esp + 24)
0x0:    <error: Cannot access memory at address 0x0>
(gdb)
```

Объясните, почему шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12] и т.д.).

10.5. Задание для самостоятельной работы

1. Преобразуйте программу из лабораторной работы №9 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму.
2. В листинге 10.3 приведена программа вычисления выражения $(3+2)*4+5$. При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

Листинг 10.3. Программа вычисления выражения $(3+2)*4+5$

```
%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
```

10.6. Содержание отчёта

Отчёт должен включать:

- Титульный лист с указанием номера лабораторной работы и ФИО студента.
- Формулировка цели работы.
- Описание результатов выполнения лабораторной работы:
  - описание выполняемого задания;
  - скриншоты (снимки экрана), фиксирующие выполнение заданий лабораторной работы;
  - комментарии и выводы по результатам выполнения заданий.
- Описание результатов выполнения заданий для самостоятельной работы:
  - описание выполняемого задания;
  - скриншоты (снимки экрана), фиксирующие выполнение заданий;
  - комментарии и выводы по результатам выполнения заданий;
  - листинги написанных программ (текст программ).
- Выводы, согласованные с целью работы.

Отчёт по выполнению лабораторной работы оформляется в формате `Markdown`. В качестве отчёта необходимо предоставить отчёты в 3 форматах: `pdf`, `docx` и `md`. А также файлы с исходными текстами написанных при выполнении лабораторной работы программ (файлы `*.asm`). Файлы необходимо загрузить на странице курса в ТУИС в задание к соответствующей лабораторной работе и загрузить на Github.

10.7. Вопросы для самопроверки

1. Какие языковые средства используются в ассемблере для оформления и активизации подпрограмм?
2. Объясните механизм вызова подпрограмм.
3. Как используется стек для обеспечения взаимодействия между вызывающей и вызываемой процедурами?
4. Каково назначение операнда в команде `ret`?
5. Для чего нужен отладчик?
6. Объясните назначение отладочной информации и как нужно компилировать программу, чтобы в ней присутствовала отладочная информация.
7. Расшифруйте и объясните следующие термины: breakpoint, watchpoint, checkpoint, catchpoint и call stack.
8. Назовите основные команды отладчика `gdb` и как они могут быть использованы для отладки программ.