Лабораторна робота №4

<https://github.com/comradeFreeman/csc.git> ( директорія lab4 )

# 1. Підготовка середовища розробки

Для виконання лабораторної роботи вам знадобиться комп'ютер (віртуальний або фізичний) архітектури AMD64/EM64T із встановленим дистрибутивом ОС Linux (будь-яким).

На систему необхідно встановити GCC, GDB, GNU Make та GNU Binutils.

Створіть окремий каталог, який будете використовувати для виконання лабораторної роботи.

Завантажте в нього файл із символами та програму-заготовку:

* [defs.h](http://tilde.slu.kiev.ua/cs/asm/defs.h)
* [exit.s](http://tilde.slu.kiev.ua/cs/asm/exit.s)



Виконайте асемблювання програми-заготовки та зв'язування:

* as -o exit.o -c exit.s
* ld -static -o exit exit.o

Пересвідчіться у тому, що виконуваний файл працездатний. Програма повинна нічого не робити і не виводити жодних помилок.





# 2. Автоматизація збірки

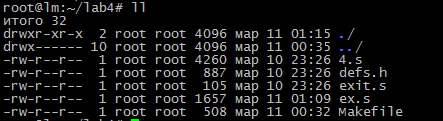
Створіть Makefile, який за командою make exit та make all виконає збірку, а за командою make clean очистить об'єктні та виконувані файли.

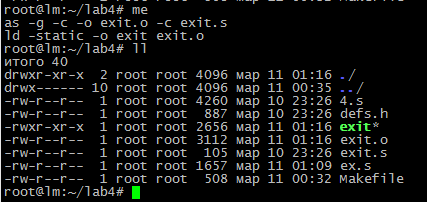
Модифікуйте Makefile так, щоб опції асемблера та лінкера задавалися змінними ASFLAGS та LDFLAGS.

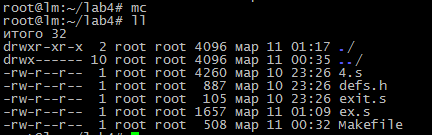
Додайте опцію асемблера для генерації відлагоджувальних символів DWARF.

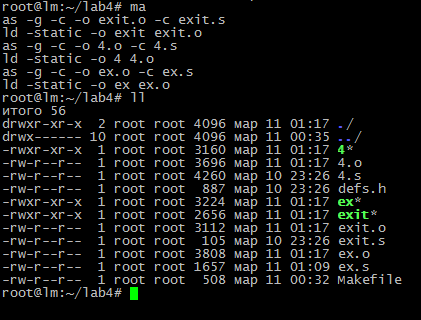
Використайте шаблонні правила так, щоб можна було збирати декілька асемблерних файлів в окремі виконувані файли. Це знадобиться при виконанні індивідуального завдання.

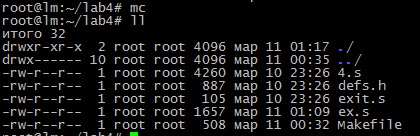
У репозиторії лабораторної – файл **Makefile**











# 3. Навички відлbагоджування

Завантажте одержаний виконуваний файл у відлагоджувач за допомогою команди:

gdb ./exit

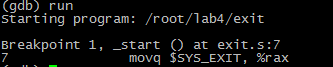
Встановіть точку зупинки на початок програми (мітка \_start):

b \_start



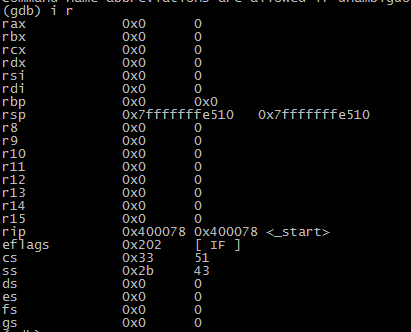
Запустіть програму

run



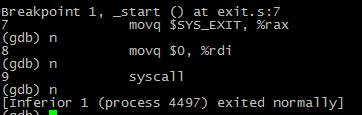
Після зупинки виконання програми перегляньте вміст регістрів:

info registers або i r



Переходьте до виконання наступної команди:

next або n



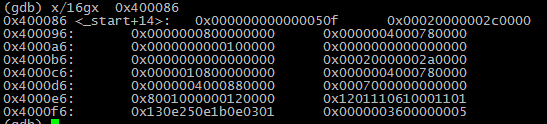
Для виходу із режиму покрокового виконання використовуйте команду

continue або с

Програма працюватиме до наступної точки зупинки або до повного чи аварійного завершення.

Для перегляду адресного простору процесу скористайтесь командою x, наприклад:

x/16gx 0x12345678



# Довідка для роботи над індивідуальним завданням

### Довідка по системним викликам

<http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/>

### Довідка по використанню GDB

<https://habr.com/ru/post/491534/>

### Створення та використання глобальних змінних

Ініціалізованих нулями -- секція .bss

Ініціалізованих будь-якими значеннями -- секція .data

Константи -- секція коду .text

Мітка створює символ, значення якого відповідає її адресі.

Наприклад:

.section .bss

envp: .quad 0 /\* long envp = 0; \*/

...

.section .data

nforks: .byte 9 /\* byte nforks = 9; \*/

...

.section .text

Звернення до значення змінної:

movq envp, %rax

decb nforks

### 

### Застосування значення символа у коді

Якщо в інструкції вказується назва символа -- то це означає ***значення за його адресою***.

Наприклад:

movq var, %rax -- запише число за адресою var (значення змінної або константи) в RAX.

Якщо ж в інструкції перед іменем стоїть $, то це означає ***безпосереднє значення*** символа. Наприклад:

movq $STDIN, %rax -- запише число 0 (значення STDIN) в RAX.

movq $1, %rax -- запише в RAX число 1

syscall

# 4. Індивідуальні завдання

## 4. Правда про своє оточення

Створіть програму, яка виводить вміст змінних оточення власного процесу на стандартний потік виведення.

### Псевдокод

int len;

byte \*p;

const char \*newline = "\n";

main() {

int argc = \*(%rsp)

char \*\*envp = %rsp + 8 \* (argc + 2);

while(envp != NULL) {

len = 0;

p = \*envp;

while(\*p != '\0') {

p++;

len++;

}

write(stdout, len, \*envp);

write(stdout, 1, newline);

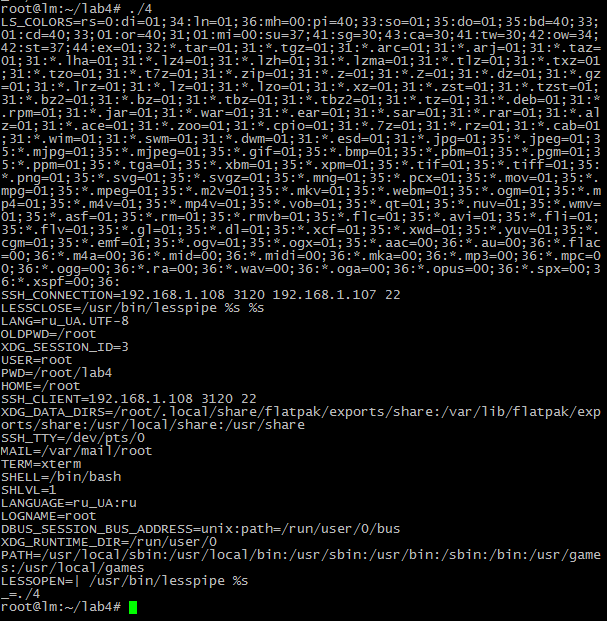
}

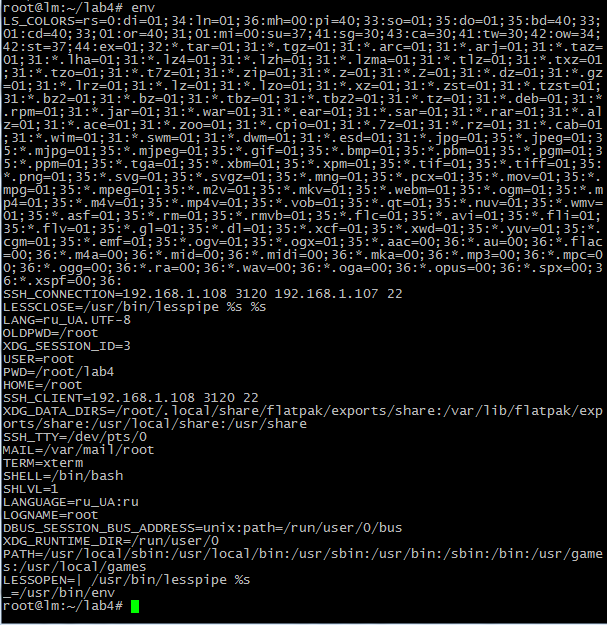
}

Код наведено у файлі **4.s** репозиторію!

**Перевірка**

Порівняйте результат виконання із результатом команди env.





Автори найкоротшого бінарного файлу (для кожного варіанту) та найбільш універсального Makefile отримають додаткові бонуси.