**Практичне заняття №2**

### **Коротка теоретична довідка**

#### **Сегменти**

Об'єктні файли та виконувані файли можуть мати кілька різних форматів. У більшості реалізацій Linux цей формат називається ELF (спочатку "Extensible Linker Format", тепер "Executable and Linking Format"). В інших системах використовується формат COFF (Common Object-File Format). Дізнатися більше про формат, який використовується у UNIX, можна через команду man a.out.

Усі ці формати мають спільну концепцію **сегментів**. Вони представляють собою області в двійковому файлі, де зберігається певний тип інформації (наприклад, таблиця символів). Також широко використовується термін "розділ", що є найменшою одиницею організації у файлі ELF. Сегмент зазвичай містить кілька розділів.

Варто не плутати **сегмент** у UNIX та **сегмент** у архітектурі Intel x86:

* У UNIX **сегмент** – це група пов'язаних даних у двійковому файлі.
* У Intel x86 **сегмент** – це область пам'яті, поділена на 64-кілобайтні блоки.

Тема сегментів архітектури Intel x86 заслуговує окремого обговорення. Коли ви запускаєте size для виконуваного файлу, він повідомляє вам розмір трьох сегментів, відомих як текст, дані та bss у виконуваному файлі. Наприклад:

**vadim@stoic ~/hspl/pract2**

**$ gcc -Wall hello.c**

**vadim@stoic ~/hspl/pract2**

**$ ls -l**

**total 169**

**-rwxr-xr-x 1 Vadim None 167993 Aug 25 20:46 a.exe**

**-rw-r--r-- 1 Vadim None 85 Aug 25 20:46 hello.c**

Команда size не виводить заголовки, тому використовуйте echo для їх створення:

**vadim@stoic ~/hspl/pract2**

**$ echo; echo "text data bss total"; size a.exe**

**text data bss total**

**text data bss dec hex filename**

**2989 1956 416 5361 14f1 a.exe**

Ще один спосіб перевірити вміст виконуваного файлу — це використання утиліт nm або dump.  
 Скомпілюйте вихідний код нижче й запустіть nm на отриманому a.out.

**vadim@stoic ~/hspl/pract2**

**$ cat fruts.c**

**#include**

**#include char pear[40];**

**static double peach;**

**int mango = 13;**

**static long melon = 2001;**

**int main() {**

**int i=3, j, \*ip; ip = malloc(sizeof(i));**

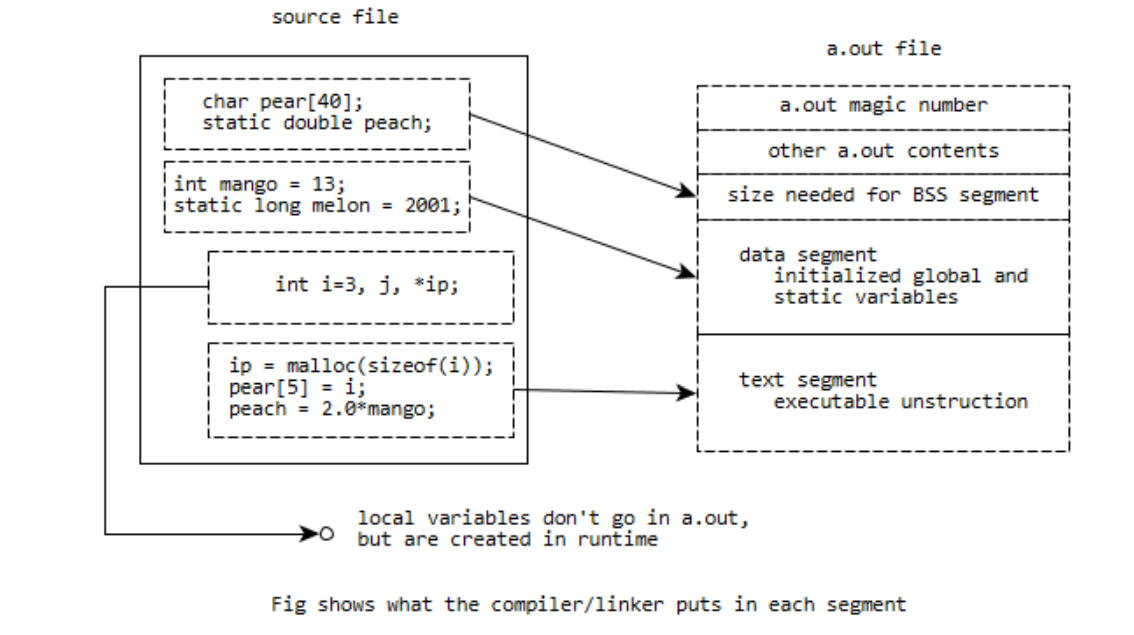
**pear[5] = i;**

**peach = 2.0\*mango;**

**return 0; }**

**Vadim@Stoic ~/hspl/pract2**

**$ gcc fruts.c Vadim@Stoic ~/hspl/pract2 Vadim@Stoic ~/hspl/pract2 $ nm -... a.exe . . .**



### **Відомий факт:**

У програмі на C неініціалізовані глобальні та статичні змінні автоматично ініціалізуються нулем. А як щодо ініціалізованих глобальних об'єктів?

**Ініціалізований сегмент даних** – це область адресного простору, де зберігаються явно ініціалізовані глобальні та статичні змінні.

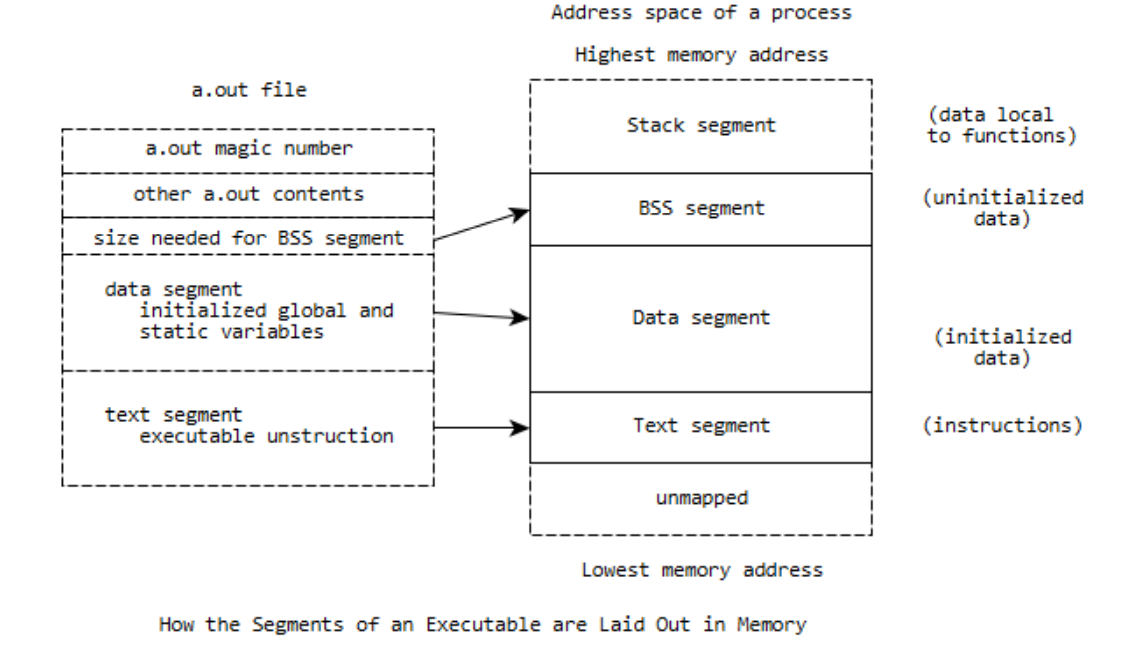
**Сегмент неініціалізованих даних** – це область адресного простору, у якій розташовані неініціалізовані глобальні та статичні змінні. Насправді вони неявно ініціалізуються нулем (через memset із нульовим значенням). Варто пам’ятати, що в старій літературі цей сегмент часто згадується як **BSS**!

Сегмент **BSS** отримав свою назву від абревіатури *«Block Started by Symbol»* – псевдооперації зі старого асемблера IBM 704, що була перенесена в UNIX і збереглася до наших днів. Дехто запам’ятовує це як *«Better Save Space»*, оскільки сегмент BSS містить лише змінні, які ще не мають жодного значення.

Розмір, необхідний для BSS під час виконання програми, записується в об'єктний файл, але сам сегмент BSS (на відміну від сегмента даних) не займає реального місця в об'єктному файлі.

Ще раз: **BSS – це стара директива асемблера («Block Started by Symbol»), яку можна ігнорувати; сьогодні BSS – це сегмент неініціалізованих даних у віртуальному адресному просторі (VAS) процесу.**

Тепер зрозуміло, чому файл a.out розбитий на сегменти. Сегменти зручно відображати у об'єкти, які компоновщик середовища виконання використовує для розміщення змінних у пам’яті.виконання може завантажувати безпосередньо. Завантажувач бере кожен образ сегмента у файлі та розміщує його в пам’яті. Сегменти стають областями пам'яті виконуваної програми, кожен із яких має певне призначення. Це зображено на малюнку нижче



При виконанні команди size для виконуваного файлу можна отримати інформацію про три основні сегменти: **текстовий**, **даних** та **BSS**. Наприклад:

**$ gcc -Wall hello.c**

**$ ls -l**

**total 169**

**-rwxr-xr-x 1 user user 167993 Aug 25 20:46 a.out**

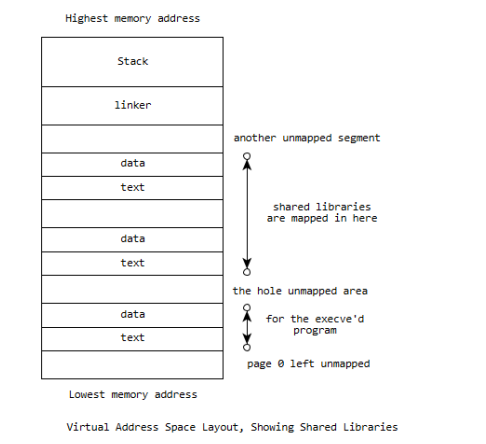
**-rw-r--r-- 1 user user 85 Aug 25 20:46 hello.c**

**$ echo "text data bss total" ; size a.out**

**text data bss total**

**2989 1956 416 5361**

Додатково можна перевірити вміст виконуваного файлу за допомогою nm або objdump.



### **Завдання ЗАГАЛЬНЕ ДЛЯ ВСІХ**

1. Напишіть програму для визначення моменту, коли time\_t закінчиться.Дослідіть, які зміни відбуваються в залежності від 32- та 64-бітної архітектури.Дослідіть сегменти виконуваного файлу.

### **Завдання 2.2.**

Розгляньте сегменти у виконуваному файлі.

1. Скомпілюйте програму "hello world", запустіть ls -l для виконуваного файлу, щоб отримати його загальний розмір, і запустіть size, щоб отримати розміри сегментів всередині нього.
2. Додайте оголошення глобального масиву із 1000 int, перекомпілюйте й повторіть вимірювання. Зверніть увагу на відмінності.
3. Тепер додайте початкове значення в оголошення масиву (пам’ятайте, що C не змушує вас вказувати значення для кожного елемента масиву в ініціалізаторі). Це перемістить масив із сегмента BSS у сегмент даних. Повторіть вимірювання. Зверніть увагу на різницю.
4. Тепер додайте оголошення великого масиву в локальну функцію. Оголосіть другий великий локальний масив з ініціалізатором. Повторіть вимірювання. Дані розташовуються всередині функцій, залишаючись у виконуваному файлі? Яка різниця, якщо масив ініціалізований чи ні?
5. Які зміни відбуваються з розмірами файлів і сегментів, якщо ви компілюєте для налагодження? Для максимальної оптимізації?

Проаналізуйте результати, щоб переконатися, що:

* сегмент даних зберігається у виконуваному файлі;
* сегмент BSS не зберігається у виконуваному файлі (за винятком примітки щодо його вимог до розміру часу виконання);
* текстовий сегмент більшою мірою піддається перевіркам оптимізації;
* на розмір файлу a.out впливає компіляція для налагодження, але не сегменти.

### **Завдання 2.3.**

Скомпілюйте й запустіть тестову програму, щоб визначити приблизне розташування стека у вашій системі:

**#include <stdio.h>**

**int main() {**

**int i;**

**printf("The stack top is near %p\n", &i);**

**return 0;**

**}**

Знайдіть розташування сегментів даних і тексту, а також купи всередині сегмента даних, оголосіть змінні, які будуть поміщені в ці сегменти, і виведіть їхні адреси.

Збільшіть розмір стека, викликавши функцію й оголосивши кілька великих локальних масивів. Яка зараз адреса вершини стека?

**Примітка:** стек може розташовуватися за різними адресами на різних архітектурах та різних ОС. Хоча ми говоримо про вершину стека, на більшості процесорів стек зростає вниз, до пам’яті з меншими значеннями адрес.

### **Завдання 2.4**

Ваше завдання – дослідити стек процесу або пригадати, як це робиться. Ви можете:

* Автоматично за допомогою утиліти gstack.
* Вручну за допомогою налагоджувача GDB.

Користувачі Ubuntu можуть зіткнутися з проблемою: на момент написання (Ubuntu 18.04) gstack, схоже, не був доступний (альтернативою може бути pstack). Якщо gstack не працює, використовуйте другий метод – через GDB, як показано нижче.

Спочатку подивіться на стек за допомогою gstack(1). Нижче наведений приклад стека bash (аргументом команди є PID процесу):

**$ gstack 14654**

**#0 0x00007f359ec7ee7a in waitpid () from /lib64/libc.so.6**

**#1 0x000056474b4b41d9 in waitchild.isra ()**

**#2 0x000056474b4b595d in wait\_for ()**

**#3 0x000056474b4a5033 in execute\_command\_internal ()**

**#4 0x000056474b4a5c22 in execute\_command ()**

**#5 0x000056474b48f252 in reader\_loop ()**

**#6 0x000056474b48dd32 in main ()**

**$**

Розбір стека:

* Номер кадру стека відображається ліворуч перед символом #.
* Кадр #0 – це найнижчий кадр. Читайте стек знизу вверх (тобто від main() – кадр #6 – до waitpid() – кадр #0).
* Якщо процес багатопотоковий, gstack покаже стек кожного потоку окремо.

### Аналіз стека в режимі користувача через GDB

Щоб переглянути стек процесу вручну, використовуйте GDB, приєднавшись до процесу.

Нижче наведена невелика тестова програма на C, що виконує кілька вкладених викликів функцій. Граф викликів виглядає так:

**main() --> foo() --> bar() --> bar\_is\_now\_closed() --> pause()**

Системний виклик pause() – це приклад блокуючого виклику. Він переводить викликаючий процес у сплячий режим, очікуючи (або блокуючи) сигнал. У цьому випадку процес блокується, поки не отримає будь-який сигнал.

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#define MSG "In function %20s; &localvar = %p\n"**

**static void bar\_is\_now\_closed(void) {**

**int localvar = 5;**

**printf(MSG, \_\_FUNCTION\_\_, &localvar);**

**printf("\n Now blocking on pause()...\n");**

**pause();**

**}**

**static void bar(void) {**

**int localvar = 5;**

**printf(MSG, \_\_FUNCTION\_\_, &localvar);**

**bar\_is\_now\_closed();**

**}**

**static void foo(void) {**

**int localvar = 5;**

**printf(MSG, \_\_FUNCTION\_\_, &localvar);**

**bar();**

**}**

**int main(int argc, char \*\*argv) {**

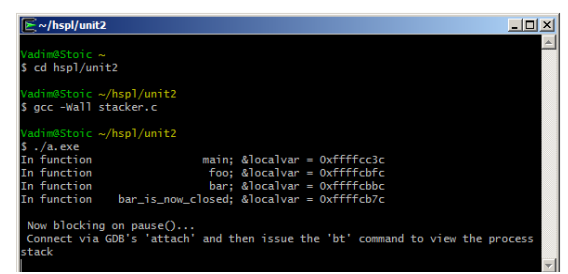
**int localvar = 5;**

**printf(MSG, \_\_FUNCTION\_\_, &localvar);**

**foo();**

**exit(EXIT\_SUCCESS);**

**}**



Тепер відкрийте GDB

У ньому підключіться (**attach**) до процесу (в наведеному прикладі **PID = 24957**) і дослідіть стек за допомогою команди backtrace (bt):

**$ gdb --quiet**

**(gdb) attach 24957**

**Attaching to process 24957**

**Reading symbols from <...>/hspl/unit2/stacker...done.**

**Reading symbols from /lib64/libc.so.6...Reading symbols from**

**/usr/lib/debug/usr/lib64/libc-2.26.so.debug...done.**

**done.**

**Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols**

**...**

**(gdb) bt**

**...**

Примітка: В Ubuntu, через питання безпеки, GDB не дозволяє підключатися до довільного процесу. Це можна обійти, запустивши GDB від імені користувача root.

### Аналіз того ж процесу через gstack

$ gstack 24957

...

gstack — це, по суті, оболонковий скрипт (wrapper shell script), який неінтерактивно викликає GDB і запускає команду backtrace, яку ви щойно використали.

Завдання: Ознайомтеся з виводом gstack і порівняйте його з GDB.

### **Завдання 2.5**

Відомо, що при виклику процедур і поверненні з них процесор використовує стек.Чи можна в такій схемі обійтися без лічильника команд (IP), використовуючи замість нього вершину стека? Обґрунтуйте свою відповідь та наведіть приклади.

**Завдання ПО ВАРІАНТАХ**

1. Напишіть програму, що визначає максимально можливий розмір сегмента кучі.
2. Дослідіть вплив використання mmap() замість malloc().
3. Реалізуйте власний менеджер пам’яті з аналогами malloc() та free().
4. Дослідіть техніки розподілу пам’яті в багатопотокових програмах.
5. Реалізуйте стекову машину, що використовує сегмент стека для обчислень.
6. Визначте мінімально необхідний стек для виконання глибокої рекурсії.
7. Створіть програму, що змінює права доступу до сегментів пам’яті (mprotect).
8. Реалізуйте експлойт для переповнення буфера.
9. Дослідіть використання сегмента BSS для збереження великих масивів.
10. Аналізуйте поведінку динамічних бібліотек під час завантаження.
11. Реалізуйте програму, що перевіряє випадкове розташування сегментів ASLR.
12. Напишіть власний завантажувач ELF-файлів.
13. Дослідіть вплив оптимізації -Os на розмір сегментів.
14. Реалізуйте програму для аналізу вмісту /proc/self/maps.
15. Виміряйте час доступу до різних сегментів пам’яті.
16. Реалізуйте зміну сегмента коду під час виконання (mremap).
17. Дослідіть витік пам’яті у довготривалих процесах.
18. Використайте ptrace() для перегляду пам’яті іншого процесу.
19. Напишіть програму для перевірки залежностей ELF-файлів.
20. Реалізуйте стековий аналізатор відстеження викликів.
21. Використайте valgrind для профілювання пам’яті.
22. Напишіть програму для обчислення перевантаження кешу.
23. Дослідіть поведінку програми при сегментації пам’яті.
24. Реалізуйте власний механізм розмітки пам’яті.
25. Аналізуйте використання пам’яті різними мовами програмування.