**ОБЩАЯ ТЕОРИЯ**

**Адресное пространство процесса** — это логическая модель памяти, которую операционная система предоставляет каждому запущенному процессу. Оно представляет собой абстракцию физической памяти, позволяя каждому процессу «думать», что у него есть вся память в распоряжении.

**Адресное пространство** — это набор адресов, которые процесс может использовать для доступа к данным и инструкциям. Эти адреса логические (виртуальные), и они отображаются на физические адреса памяти через механизм, называемый **виртуальной памятью**.

**Виртуальная память** — это один из ключевых механизмов современных операционных систем. Она позволяет программам работать с виртуальными адресами, которые затем отображаются в физические адреса памяти (ОЗУ или диск). Это создаёт иллюзию у каждой программы, что у неё есть свой собственный, непрерывный участок памяти.

Ключевая идея — виртуальный адрес, с которым работает программа, не совпадает с физическим адресом в ОЗУ. Между ними есть «переводчик» — таблица страниц.

Этот перевод происходит при каждом обращении к памяти (чтение или запись), автоматически, на уровне железа — через MMU (Memory Management Unit).

**Что такое страница?**

Вся память (как физическая, так и виртуальная) делится на маленькие куски одинакового размера, которые называются:

Виртуальные страницы — от программы

Физические страницы (или кадры) — в RAM

Обычно размер страницы — 4 КБ (хотя бывают 2 МБ, 1 ГБ — «большие страницы»)

**Что такое таблица страниц?**

Таблица страниц — это структура, которую ОС создаёт для каждого процесса. Она говорит:

«Эта виртуальная страница X соответствует физической странице Y»

Когда программа обращается к памяти, MMU использует эту таблицу, чтобы найти нужный физический адрес.

**Как это происходит на практике?**

Пусть у нас есть:

Виртуальный адрес: 0x12345678

Размер страницы: 4 КБ (4096 байт)

Тогда:

Верхние биты адреса указывают номер страницы:

→ 0x12345 (это номер виртуальной страницы)

Нижние биты — смещение внутри страницы:

→ 0x678 (в каком месте внутри страницы находятся данные)

ОС по таблице находит:

виртуальная страница 0x12345 → физическая страница 0x0ABC3

Итоговый физический адрес:

0x0ABC3000 + 0x678 = 0x0ABC3678

Это всё делает MMU автоматически — для программы это просто один непрерывный массив памяти.

За трансляцию виртуальных адресов в физические отвечает специальное устройство, находящееся в процессоре!

**Зачем нужна** **виртуальная память?**

1. Изоляция процессов

Каждый процесс получает своё виртуальное пространство, недоступное другим.

Позволяет избежать ошибок и сбоев при попытках доступа к «чужой» памяти.

2. Упрощение управления памятью

Программист не должен заботиться о том, где именно в ОЗУ находится переменная или массив.

Операционная система сама решает, куда физически размещать данные.

3. Эффективное использование памяти

Не вся программа загружается в оперативную память сразу.

Подгружаются только те части, которые реально используются (это называется demand paging).

4. Виртуализация ресурсов

Программа может использовать больше памяти, чем реально доступно в ОЗУ (с помощью подкачки на диск — swap).

**Как программа использует больше памяти, чем есть в RAM?**

ОС не обязана загружать все страницы сразу в оперативную память. Она делает это по мере необходимости:

Механизм: Подкачка (Swapping)

1. Если в ОЗУ не хватает места:

ОС выбирает наименее используемую страницу.

Сохраняет её на диск (в файл подкачки / swap).

Освобождает место в ОЗУ.

2. Когда программе снова нужна та выгруженная страница:

Происходит Page Fault (ошибка страницы).

ОС читает страницу с диска обратно в ОЗУ.

Обновляет таблицу страниц.

Программа не знает, что данные гуляли на диск и обратно — для неё это прозрачно.

Пример:

Представь, у тебя 8 ГБ ОЗУ. Программа пытается занять 10 ГБ. Это физически невозможно. Но благодаря виртуальной памяти и подкачке, программа всё равно будет работать!

1. Программа запускается, ОС выделяет ей виртуальное адресное пространство.

Только часть данных реально загружается в RAM, остальное — пока нет.

2. Программа работает, память заполняется

Программа просит у ОС всё больше и больше памяти (например, массивы, объекты).

3. В какой-то момент ОЗУ заканчивается.

4. ОС ищет «спящие» страницы. Это страницы, которые давно не использовались. Например: глобальная переменная, к которой не обращались 10 минут.

5. ОС сохраняет страницу на диск (в swap-файл или раздел swap)

Страница выгружается из ОЗУ.

6. В таблице страниц ставится флаг: «Эта страница теперь на диске».

7. Программа вдруг хочет использовать эту страницу

MMU видит: в таблице страниц нет физического адреса. Срабатывает Page Fault — специальное прерывание. ОС: «О, эта страница нужна!». Считывает её обратно в RAM. Если нужно — выгоняет другую страницу.

**Основные области адресного пространства**

Типичное адресное пространство процесса в современных операционных системахделится на следующие сегменты:

1. Текстовый сегмент (Text Segment)

Содержит машинный код программы (инструкции).

Обычно доступен только для чтения (чтобы защитить от случайных изменений кода).

Может быть разделяемым между процессами (если они исполняют один и тот же бинарник).

2. Сегмент данных (Data Segment)

Хранит глобальные и статические переменные.

Делится на две части:

Инициализированные данные — переменные с заданными значениями.

Неинициализированные данные (BSS) — переменные, которым не присвоены значения при старте (инициализируются нулями).

3. Куча (Heap)

Используется для динамического выделения памяти (через malloc, new, и т. п.).

Растет вверх (в сторону больших адресов).

Управляется программой через вызовы системных функций.

4. Стек (Stack)

Используется для хранения локальных переменных, адресов возврата и параметров функций.

Растет вниз (в сторону меньших адресов).

Каждая функция имеет свой стек-фрейм.

5. Сегмент памяти, отображённой в пространство процесса (Memory Mapped Region)

Сюда загружаются разделяемые библиотеки (.so, .dll).

Также используется при работе с файлами через mmap.

**Что такое регионы памяти (memory regions)?**

**Регион памяти** — это непрерывный участок виртуального адресного пространства, который:

имеет определённое назначение:

- обладает своими правами доступа (например, только чтение, только запись, исполнение)

- может отображаться на:

- физическую память (RAM),

- файл (через mmap),

- или быть пустым (будет выделена при обращении)

Это вывод типа *cat /proc/<pid>/maps* в Linux.

Каждая строка — один регион памяти.

0x00400000–0x0040b000 r-xp /bin/my\_program ← код (text)

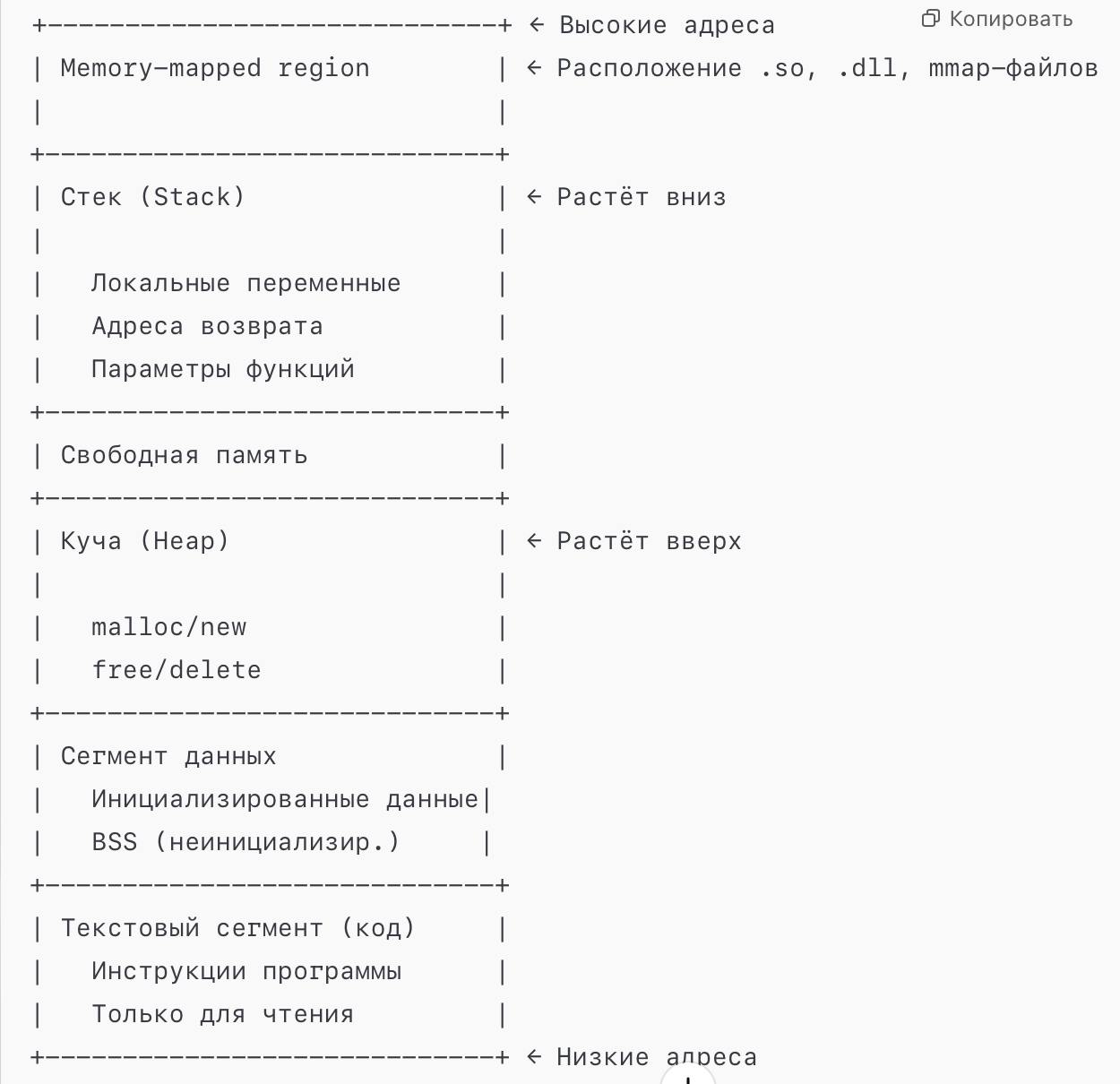
0x0060b000–0x0060d000 rw-p /bin/my\_program ← data

0x0060d000–0x0062d000 rw-p [heap] ← куча

0x7ffeef2dd000–0x7ffeef2ff000 rw-p [stack] ← стек

0x7f2a64000000–0x7f2a64200000 rw-p [anon] ← mmap/анонимная

0x7f2a65000000–0x7f2a65100000 r-xp /lib/libc.so.6 ← библиотека



**Заключение: как всё работает вместе?**

Когда ты запускаешь программу:

ОС загружает код в text segment.

Загружает глобальные переменные в data segment.

Выделяет heap (будет использоваться по мере вызовов malloc).

Создаёт стек, где будет происходить вызов main() и всех других функций.

Загружает библиотеки в memory-mapped регион.

Все эти сегменты отображаются в виртуальное адресное пространство процесса, изолированное от других.

**ЗАДАНИЕ**

*1. Структура адресного пространства.*

*a. Напишите программу, которая создает переменные и выводит их адреса:*

*i. локальные в функции;*

*ii. статические в функции;*

*iii. константы в функции;*

*iv. глобальные инициализированные;*

*v. глобальные неинициализированные;*

*vi. глобальные константы.*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

int global\_not\_inited\_var;

int global\_inited\_var = 1;

const int global\_const\_var = 2;

int main() {

int local\_var;

static int static\_var;

const int local\_const\_var = 3;

printf("pid: %d\n\n", getpid());

printf("global\_not\_inited\_var: %p, %d\n", &global\_not\_inited\_var, global\_not\_inited\_var);

printf("global\_inited\_var: %p, %d\n", &global\_inited\_var, global\_inited\_var);

printf("global\_const\_var: %p, %d\n", &global\_const\_var, global\_const\_var);

printf("local\_var: %p, %d\n", &local\_var, local\_var);

printf("static\_var: %p, %d\n", &static\_var, static\_var);

printf("local\_const\_var: %p, %d\n", &local\_const\_var, local\_const\_var);

sleep(60);

return EXIT\_SUCCESS;

}

*b. Сопоставьте адреса переменных с областями адресного пространства из соответствующего /proc/<pid>/maps. Объясните увиденное.*

1) Локально у себя на компьютере создаем файл

2) Компилируем

gcc main1.c -o main13) Запускаем

./main14) Пока программа работает (блогадря sleep(60) она "виснет" 60 секунд) открываем новое окно терминала в нем запускаем

/proc/<pid>/maps**НА MacOS ДРУГАЯ КОМАНДА:**

vmmap <pid>где вместо <pid> указываем номер процесса, с которым запустился наш main1.c

5) Сравниваем адреса памяти, по которым лежат наши переменные (первое окно терминала) с регионами (второе окно терминала), сравниваем сегменты (текст/дата/др.)

**Пояснение к vmmap-выводу**

Когда ты запустишь vmmap, ты увидишь блоки вроде:

\_\_TEXT — код программы (функции)

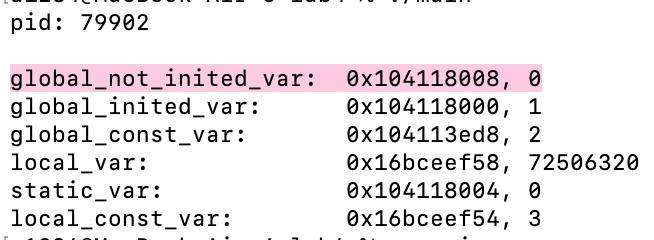
\_\_DATA — инициализированные глобальные и статические переменные

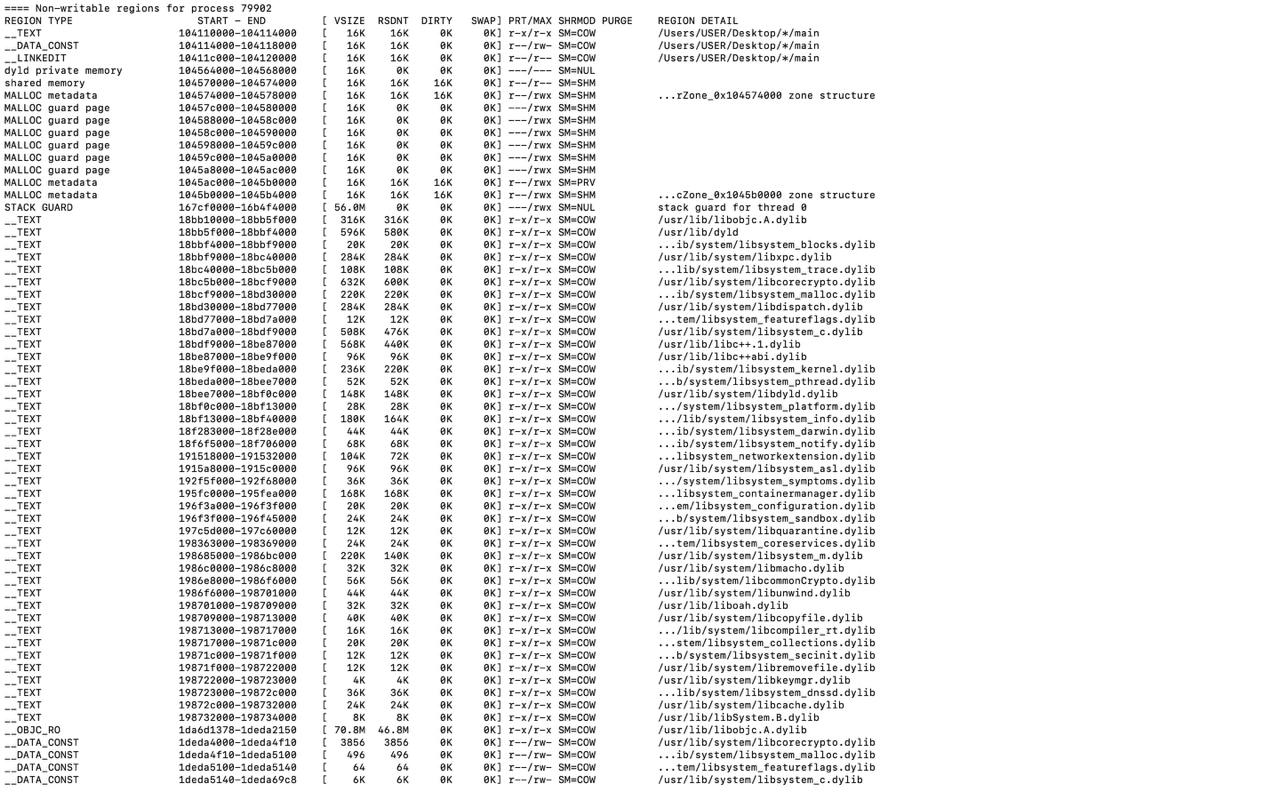
\_\_BSS — неинициализированные глобальные/статические

\_\_STACK — стек (локальные переменные)

\_\_LINKEDIT — данные от компоновщика

MALLOC — куча (heap)







Просто запоминаем, что

1) локальные перменные, находящиеся в функциях, хранятся на СТЕКЕ

2) глобальные переменные хранятся в СЕКЦИИ ДАННЫХ (неинициализированные в ПОДсекции BSS)

3) .rodata - область ТОЛЬКО для чтения (обычно в ней хранятся КОНСТАНТЫ, которые нельзя изменить иили перезаписать)

**Кстати!**

Здесь можно обратить внимание на то, что

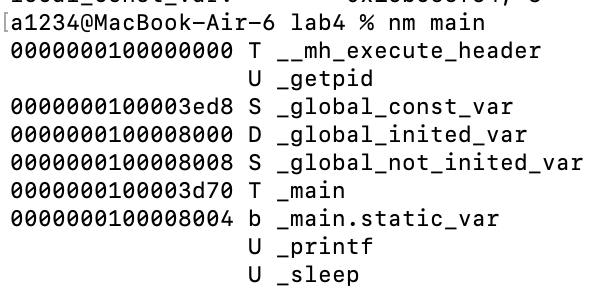
1) Неинициализированная ГЛОБАЛЬНАЯ переменная всегда инициализируется значением 0 (она хранится в сегменте bss и ОС сама присваивает ей значение 0)

2) Неинициализированная ЛОКАЛЬНАЯ переменная (например, в данной программе - local\_var) инициализируется МУСОРОМ (так как она лежит на стеке, мы не знаем в какое именно место она попадет)





*c. Используя утилиту nm (или readelf) определите в каких секциях находятся выделенные переменные переменные.*



**Заглавные буквы - глобальные пермеенные**

**Маленькие буквы - локальные переменные**

**ВАЖНО!**

Команда nm отображает символы, связанные с глобальными и статическими переменными, а также функциями, которые находятся в объектных файлах или исполнимых файлах. Но она не отображает локальные переменные.

Локальные переменные, объявленные внутри функций, существуют только в рамках вызова функции, и их адреса размещаются в стеке (stack). Они не являются глобальными переменными, и компилятор обычно не включает их в таблицу символов для отображения с помощью nm. Локальные переменные существуют только в контексте текущего выполнения функции, а после завершения работы функции их память освобождается. Поэтому локальные переменные не имеют символов в исполнимом файле или объектном файле.

**Разбор вывода:** (только нужное для лабы)

1. T (Text section)

Указывает на символ, который находится в секции кода (Text section). Это обычно функции или код программы.

Например, если перед функцией стоит T, это означает, что функция размещена в сегменте с кодом программы.

2. S (Static data section)

Указывает на символ, который находится в секции данных, но обычно это глобальные или статические переменные.

В случае глобальных и статических переменных S означает, что переменная размещена в секции данных, и ее значение задано в коде программы.

3. D (Data section)

Указывает на инициализированную глобальную переменную в секции данных (Data section). Эти переменные имеют значение, которое присваивается им в коде программы.

4. b (BSS section)

Указывает на переменную, которая находится в секции BSS. Это неинициализированные глобальные или статические переменные. Во время выполнения программы им будет присвоено значение по умолчанию (например, 0).

*d. Напишите функцию, которая создает и инициализирует локальную переменную и возвращает ее адрес. Прокомментируйте результат и дайте оценку происходящему.*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

int global\_not\_inited\_var;

int global\_inited\_var = 1;

const int global\_const\_var = 2;

int\* InitLocalVar() {

int local\_inited\_var = 4;

printf("local\_inited\_var: %d\n", local\_inited\_var);

return &local\_inited\_var;

}

int main() {

int local\_var;

static int static\_var;

const int local\_const\_var = 3;

printf("pid: %d\n\n", getpid());

printf("global\_not\_inited\_var: %p, %d\n", &global\_not\_inited\_var, global\_not\_inited\_var);

printf("global\_inited\_var: %p, %d\n", &global\_inited\_var, global\_inited\_var);

printf("global\_const\_var: %p, %d\n", &global\_const\_var, global\_const\_var);

printf("local\_var: %p, %d\n", &local\_var, local\_var);

printf("static\_var: %p, %d\n", &static\_var, static\_var);

printf("local\_const\_var: %p, %d\n", &local\_const\_var, local\_const\_var);

InitLocalVar();

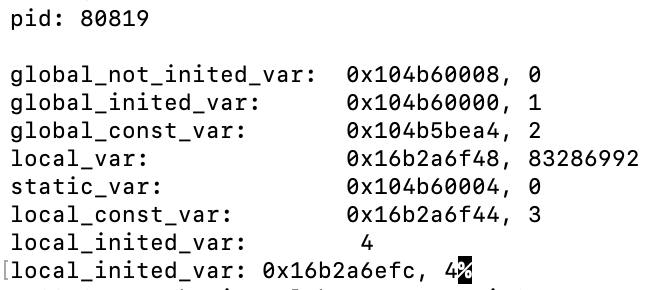
int \*local\_inited\_var\_address = InitLocalVar(); // разыменовываем адрес

printf("local\_inited\_var: %p, %d\n", local\_inited\_var\_address, \*local\_inited\_var\_address);

sleep(60);

return EXIT\_SUCCESS;

}



Видим, что программа вывела значение переменной local\_inited\_var (4) и адрес в стеке, по которому она хранится

**ДОП ВОПРОС (ВАЖНО РАЗОБРАТЬ)**

Если мы немного изменим main() и переместим вызов функции наверх:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

int global\_not\_inited\_var;

int global\_inited\_var = 1;

const int global\_const\_var = 2;

int\* InitLocalVar() {

int local\_inited\_var = 4;

printf("local\_inited\_var: %d\n", local\_inited\_var);

return &local\_inited\_var;

}

int main() {

int local\_var;

static int static\_var;

const int local\_const\_var = 3;

int \*local\_inited\_var\_address = InitLocalVar();

printf("pid: %d\n\n", getpid());

printf("global\_not\_inited\_var: %p, %d\n", &global\_not\_inited\_var, global\_not\_inited\_var);

printf("global\_inited\_var: %p, %d\n", &global\_inited\_var, global\_inited\_var);

printf("global\_const\_var: %p, %d\n", &global\_const\_var, global\_const\_var);

printf("local\_var: %p, %d\n", &local\_var, local\_var);

printf("static\_var: %p, %d\n", &static\_var, static\_var);

printf("local\_const\_var: %p, %d\n", &local\_const\_var, local\_const\_var);

InitLocalVar();

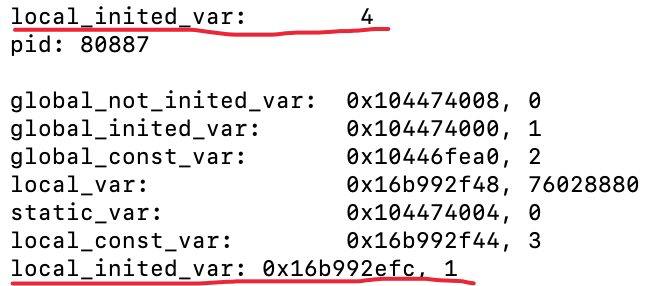
printf("local\_inited\_var: %p, %d\n", local\_inited\_var\_address, \*local\_inited\_var\_address);

sleep(60);

return EXIT\_SUCCESS;

}

то программа теперь выведет следующее:



Сначала она в вызове функции InitLocalVar выведет правильное значение (4) local\_inited\_var, а в самом конце там уже будет не 4, а 1.

Это происходит потому, что в первом варианте программы мы после return сразу вызываем printf, и наш стек еще не успел очиститься. А во втором варианте после работы функции InitLocalVar мы делаем еще кучу других printf, и наш стек очищается (можно еще дополнительно прочитать про то, как очищается стек (меняется регистр rsp)), теперь там лежит мусор.

*e. Напишите функцию, которая:*

*i. выделяет на куче буфер (например, размером 100 байт);ii. записывает в него какую-либо фразу (например, hello world);*

*iii. выводит содержимое буфера;*

*iv. освобождает выделенную память;*

*v. снова выводит содержимое буфера;*

*vi. выделяет еще один буфер;*

*vii. записывает в них какую-либо фразу (например, hello world);*

*viii. выводит содержимое буфера;*

*ix. перемещает указатель на середину буфера;*

*x. освобождает память по этому указателю.*

*xi. выводит содержимое буфера;*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

int global\_not\_inited\_var;

int global\_inited\_var = 1;

const int global\_const\_var = 2;

int\* InitLocalVar() {

int local\_inited\_var = 4;

printf("local\_inited\_var: %d\n", local\_inited\_var);

return &local\_inited\_var;

}

void InitBufferInHeap() {

int buffer\_size = 100;

char \*local\_buffer = malloc(buffer\_size \* sizeof(char));

if (local\_buffer == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

strcpy(local\_buffer, "Hello, World!");

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

free(local\_buffer);

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

local\_buffer = malloc(buffer\_size \* sizeof(char));

if (local\_buffer == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

strcpy(local\_buffer, "Hello, World!");

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

char \*mid\_buffer = local\_buffer + 50;

printf("mid\_buffer: %p\n", mid\_buffer);

free(mid\_buffer);

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

}

int main() {

int local\_var;

static int static\_var;

const int local\_const\_var = 3;

printf("pid: %d\n\n", getpid());

printf("global\_not\_inited\_var: %p, %d\n", &global\_not\_inited\_var, global\_not\_inited\_var);

printf("global\_inited\_var: %p, %d\n", &global\_inited\_var, global\_inited\_var);

printf("global\_const\_var: %p, %d\n", &global\_const\_var, global\_const\_var);

printf("local\_var: %p, %d\n", &local\_var, local\_var);

printf("static\_var: %p, %d\n", &static\_var, static\_var);

printf("local\_const\_var: %p, %d\n", &local\_const\_var, local\_const\_var);

InitLocalVar();

int \*local\_inited\_var\_address = InitLocalVar();

printf("local\_inited\_var: %p, %d\n", local\_inited\_var\_address, \*local\_inited\_var\_address);

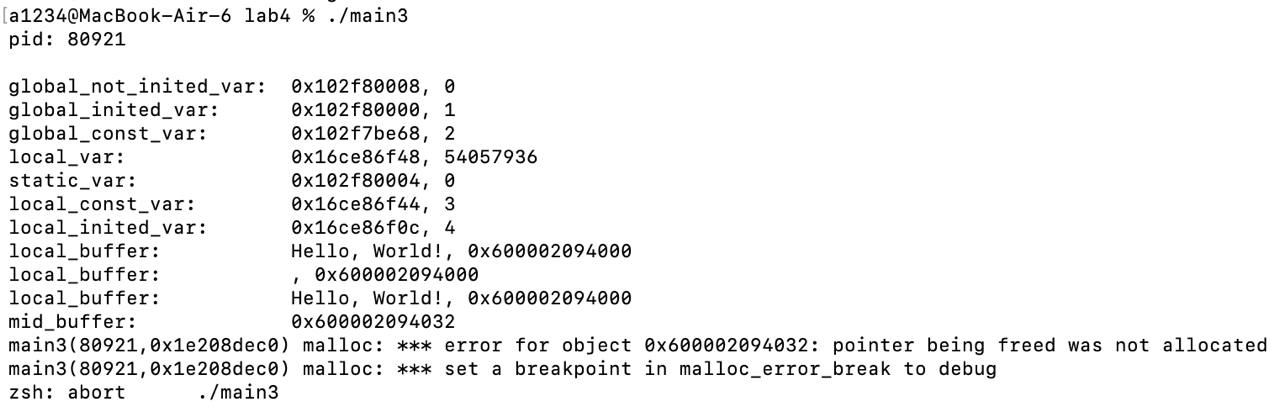
InitBufferInHeap();

sleep(60);

return EXIT\_SUCCESS;

}

*f. Прокомментируйте работу предыдущего пункта.*



**Объяснение вывода:**

1. local\_buffer: Hello, World!, 0x600002094000

Вызывается malloc, память выделяется, и туда копируется строка "Hello, World!".

2. local\_buffer: , 0x600002094000

Вы освобождаете память (free(local\_buffer);) и тут же пытаетесь её снова вывести — UB (undefined behavior), поэтому строка пустая.

3. local\_buffer: Hello, World!, 0x600002094000

Вы повторно выделяете память, возможно, она попадает по тому же адресу (куча — управляется аллокатором, это вполне допустимо).

**4. mid\_buffer: 0x600002094032**

**main3(80921,0x1e208dec0) malloc: \*\*\* error for object 0x600002094032: pointer being freed was not allocated**

**main3(80921,0x1e208dec0) malloc: \*\*\* set a breakpoint in malloc\_error\_break to debug**

**zsh: abort ./main3**

Вы пытаетесь освободить не тот указатель, который вы получили от malloc.

Вы выделили память: char \*local\_buffer = malloc(...) → допустим, получили 0x600002094000.

А освобождаете указатель внутри блока памяти: mid\_buffer = local\_buffer + 50 → 0x600002094032.

Ошибка! free() разрешается вызывать только на оригинальный указатель, возвращённый malloc/realloc/calloc. Иначе возникает ошибка, потому что система управления памятью не знает, как корректно освободить "середину" блока.

**Почему, раз мы освобождаем память с помощью free, мы еще можем по этому адресу получить local\_buffer?**

Что делает free()?

Освобождает участок памяти, на который указывает указатель (возвращённый malloc, calloc или realloc).

НЕ обнуляет указатель, т.е. local\_buffer всё ещё указывает на ту же самую область, даже если она уже может быть:

- Переписана,

- Выделена под другой объект,

- Недоступна (и доступ к ней — undefined behavior).

ТО ЕСТЬ ОС/аллокатор помечает блок как "свободный", но:

НЕ затирает данные,

НЕ изменяет значение переменной local\_buffer.

local\_buffer всё ещё содержит тот же адрес: 0x600002094000.

*g. Заведите переменную окружения.*

*h. Добавьте в вашу программу код, который:*

*i. распечатывает ее значение;*

*ii. изменяет его значение;*

*iii. повторно распечатывает ее значение.*

*i. Запустите вашу программу и убедитесь что переменная окружения*

*имеет требуемое значение.*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

#define SUCCESS 0

#define ERROR (-1)

int global\_not\_inited\_var;

int global\_inited\_var = 1;

const int global\_const\_var = 2;

int\* InitLocalVar() {

int local\_inited\_var = 4;

printf("local\_inited\_var: %d\n", local\_inited\_var);

return &local\_inited\_var;

}

void InitBufferInHeap() {

int buffer\_size = 100;

char \*local\_buffer = malloc(buffer\_size \* sizeof(char));

if (local\_buffer == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

strcpy(local\_buffer, "Hello, World!");

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

free(local\_buffer);

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

local\_buffer = malloc(buffer\_size \* sizeof(char));

if (local\_buffer == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

strcpy(local\_buffer, "Hello, World!");

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

char \*mid\_buffer = local\_buffer + 50;

printf("mid\_buffer: %p\n", mid\_buffer);

free(mid\_buffer);

printf("local\_buffer: %s, %p\n", local\_buffer, local\_buffer);

}

void ChangeEnvVar() {

int ret = SUCCESS;

char \*env\_var\_name = "ENV\_VAR";

char \*env\_var\_value;

char \*new\_env\_var\_value = "response";

env\_var\_value = getenv(env\_var\_name);

printf("env\_var: %s\n", env\_var\_value);

ret = setenv(env\_var\_name, new\_env\_var\_value, true);

if (ret == ERROR) {

perror("setenv");

exit(EXIT\_FAILURE);

}

env\_var\_value = getenv(env\_var\_name);

printf("env\_var: %s\n", env\_var\_value);

}

int main() {

int local\_var;

static int static\_var;

const int local\_const\_var = 3;

printf("pid: %d\n\n", getpid());

printf("global\_not\_inited\_var: %p, %d\n", &global\_not\_inited\_var, global\_not\_inited\_var);

printf("global\_inited\_var: %p, %d\n", &global\_inited\_var, global\_inited\_var);

printf("global\_const\_var: %p, %d\n", &global\_const\_var, global\_const\_var);

printf("local\_var: %p, %d\n", &local\_var, local\_var);

printf("static\_var: %p, %d\n", &static\_var, static\_var);

printf("local\_const\_var: %p, %d\n", &local\_const\_var, local\_const\_var);

InitLocalVar();

int \*local\_inited\_var\_address = InitLocalVar();

printf("local\_inited\_var: %p, %d\n", local\_inited\_var\_address, \*local\_inited\_var\_address);

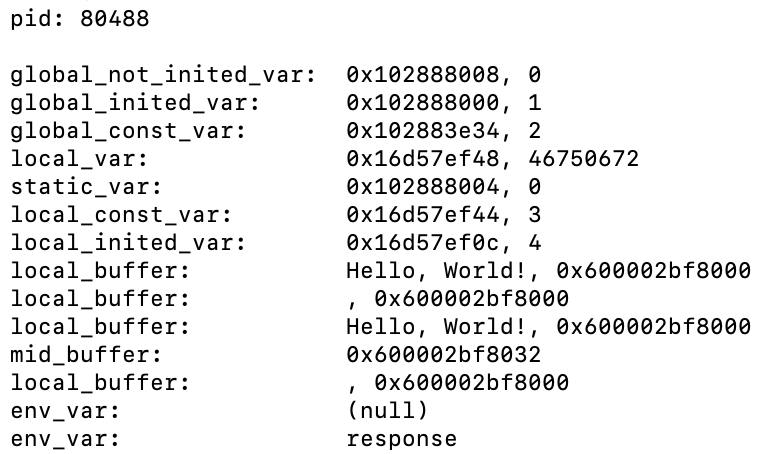
InitBufferInHeap();

ChangeEnvVar();

sleep(60);

return EXIT\_SUCCESS;

}



**Что такое переменные окружения?**

Переменные окружения (environment variables) — это:

Пары имя = значение, которые существуют в процессе (в пространстве памяти).

Используются для передачи конфигурационной информации в программы без изменения кода.

Например:

export PATH="/usr/bin:/bin"

export ENV\_VAR="test"

Вот самые типичные применения:

1. PATH Где искать исполняемые файлы

2. HOME Домашняя директория пользователя

3. USER Имя пользователя

4. LD\_LIBRARY\_PATH Где искать динамические библиотеки

5. пользовательские (MY\_APP\_CONFIG, ENV\_VAR) Конфигурация для своих программ

**Где и как они хранятся?**

Внутри процесса!!!:

После запуска программы каждый процесс получает копию окружения в виде массива строк:

extern char \*\*environ;

или передается как параметр main:

int main(int argc, char \*argv[], char \*envp[])

**Где в памяти?**

В Linux/macOS/Unix переменные окружения хранятся в сегменте данных (data segment) или в stack/heap, в зависимости от реализации.

Но обычно: в начале stack, рядом с argv.

**Могут ли несколько процессов рабоать с одной переменной окружения?**

Нет, переменные окружения не являются общими между процессами.

Переменные окружения — индивидуальны для каждого процесса

Как это работает:

Когда вы запускаете программу из оболочки (например, ./prog), оболочка копирует своё окружение в новый процесс. Это означает:

У каждого процесса — своя копия окружения.

Если один процесс меняет значение переменной окружения с помощью setenv(), другие процессы этого не увидят.

**Почему переменные окружения не общие?**

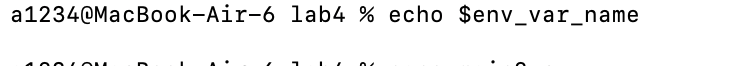
При запуске нового процесса (через fork() и/или exec()), среда окружения копируется.

То есть: это приватная область памяти, как стек или куча.

Это сделано намеренно: переменные окружения — способ передать конфигурацию, а не средство IPC (межпроцессного взаимодействия).

*j. Выведите значение переменной окружения после того как ваша*

*программа завершилась.*



*k. Объясните произошедшее.*

**Почему переменной окружения нет после завершения программы?**

Когда ты вызываешь setenv("ENV\_VAR", "response", 1); внутри своей программы, ты на самом деле:

Изменяешь окружение только в этом процессе.

Эти изменения живут только до завершения программы.

После выхода из main() — весь процесс, включая его окружение, уничтожается.

**Почему echo $env\_var\_name ничего не показывает?**

Потому что env\_var\_name — это переменная C, а не shell.

Если ты хотела проверить значение ENV\_VAR, нужно написать:

echo $ENV\_VARНо даже это не поможет — потому что твоя программа не может изменить переменные окружения оболочки, из которой запущена.