4 задача

Адресное пространство процесса

Прежде всего немного теории, так как я делю эту лабу до того как была лекция по адресным пространствам .

1. Что такое виртуальная память?

Виртуальная память — это абстракция, которую предоставляет операционная система (ОС), чтобы программы "думали", что у них есть **собственное непрерывное адресное пространство**, даже если физическая память (RAM) фрагментирована или её не хватает.

- **Физическая память (RAM)** — реальные микросхемы памяти в компьютере (например, 8 ГБ).
-

Как это возможно?

- Операционная система (ОС) использует **жёсткий диск (SSD/HDD)** как "продолжение" RAM.
- Данные, которые не помещаются в RAM, временно выгружаются в **файл подкачки (swap)**.

Механизм подкачки (Swapping)

Когда RAM заканчивается:

1. **ОС выбирает "редко используемые" данные** из RAM.
2. **Записывает их на диск** в специальный файл или раздел (например, `/swapfile` в Linux).
3. **Освобождает место в RAM** для новых данных.
4. **Когда программе снова нужны эти данные** — ОС загружает их обратно в RAM (если нужно, выгружая что-то другое).

Пример:

- У вас есть ноутбук с **4 ГБ RAM**, а игра требует **6 ГБ**.
- ОС выгружает часть данных (например, фоновые приложения) на диск.
- Игра работает, но может немного **тормозить** (диск медленнее RAM).

Основные свойства:

- Каждый процесс считает, что у него есть **вся память компьютера** (например, 64-битное приложение "видит" 16 эксабайт памяти, даже если RAM всего 8 ГБ).
- Память процесса **изолирована** от других процессов (программа не может случайно повредить память другой программы).
- ОС подгружает данные из RAM на диск (в **swap**), если физической памяти не хватает.

---

2. Зачем нужна виртуальная память?

Без неё было бы **очень неудобно**:

1. **Программы мешали бы друг другу**
   - Если одна программа записала что-то по адресу `0x1234`, другая могла бы случайно перезаписать это.
2. **Не хватило бы RAM**
   - Современные программы часто используют гигабайты памяти, но физической RAM меньше.
3. **Память была бы фрагментирована**
   - Программе пришлось бы работать с "дырявыми" кусками RAM.

Что даёт виртуальная память?

✅ **Изоляция процессов** (один процесс не может сломать другой).
✅ **Безопасность** (например, `r-xp` код нельзя перезаписать).
✅ **Большие адресные пространства** (даже если RAM мало).
✅ **Лёгкость программирования** (программа видит линейную память, а не куски RAM).

---

3. Структура виртуальной памяти процесса

В вашем задании (`/proc/<PID>/maps`) видно, как память процесса делится на регионы:

Регион	Права	Для чего нужен?	Пример из вашего вывода
Программа (`.text`)	`r-xp`	Код программы (функции)	`606f5583a000-606f5583b000`
Константы (`.rodata`)	`r--p`	Глобальные `const` переменные	`606f5583b000-606f5583c000`
Глобальные переменные (`.data`, `.bss`)	`rw-p`	Статические/глобальные переменные	`606f5583d000-606f5583e000`
Куча (`[heap]`)	`rw-p`	Динамическая память (`malloc`)	`606f7bfc8000-606f7bfe9000`
Стек (`[stack]`)	`rw-p`	Локальные переменные, вызовы функций	`7ffd43ed7000-7ffd43ef9000`
Библиотеки (`libc.so`)	`r-xp` / `rw-p`	Код и данные библиотек	`77ec8e428000-77ec8e5b0000`

---

4. Как виртуальная память связана с вашим заданием?

В вашей программе:

- Вы смотрели адреса переменных (`&x`).
- Они попадали в разные регионы виртуальной памяти:
  - **Локальные переменные** → стек (`[stack]`).
  - **Глобальные/статические** → `.data` / `.bss`.
  - **Константы** → `.rodata`.

- `/proc/<PID>/maps` показывал, **какие права** у этих регионов и **откуда они загружены**.

Пример:

```
int global = 42;  // → .data (rw-p)
static int x = 10; // → .data (rw-p)
const int y = 100; // → .rodata (r--p)
int main() {
    int local = 5; // → [stack] (rw-p)
}
```

---

5. Что было бы, если бы не было виртуальной памяти?

1. **Программы бы ломали друг друга**
   - Если бы два процесса использовали один и тот же адрес `0x1234`, они бы конфликтовали.
2. **Нельзя было бы запускать несколько программ одновременно**
   - Каждая программа требовала бы **непрерывный кусок RAM**.
3. **Не было бы защиты от ошибок**
   - Если программа записывает в случайный адрес — могла бы убить систему.
4. **Не работал бы `malloc` и динамические библиотеки**
   - Память выделялась бы только на этапе загрузки программы.

---

6. Как это работает на практике?

1. **Программа пишет в "виртуальный" адрес** (например, `0x606f5583d010`).
2. **Процессор + ОС переводят его в физический адрес** через **таблицы страниц** (Page Tables).
3. **Если страницы нет в RAM** — происходит **Page Fault**, и ОС подгружает её с диска.

```
606f5583d000-606f5583e000 rw-p 00003000 103:0b 802072 /home/nekoie/.../addr_1
```

- Виртуальный адрес `0x606f5583d000` может быть **физически** где угодно (даже на диске в swap).
- ОС делает так, что программа **не видит** эту сложность.

Если бы её не было, программисты бы до сих пор писали в Assembler и вручную распределяли память....

Виртуальная память процесса в Linux организована в несколько ключевых сегментов, каждый из которых выполняет свою особую функцию.

1. Сегменты исполняемого файла (программы)

Сегмент	Права	Хранимые данные	Пример из `maps`
`.text`	`r-xp`	Исполняемый код программы	`606f5583a000-606f5583b000 r-xp`
`.rodata`	`r--p`	Константы (глобальные `const`)	`606f5583b000-606f5583c000 r--p`
`.data`	`rw-p`	Инициализированные глобальные переменные	`606f5583d000-606f5583e000 rw-p`

Сегмент	Права	Хранимые данные	Пример из `maps`
`.bss`	`rw-p`	Неинициализированные глобальные переменные (заполнены нулями)	Входит в `.data`

---

2. Динамически выделяемая память

Сегмент	Права	Описание	Пример из `maps`
`[heap]`	`rw-p`	Динамическая память (`malloc` / `new`)	`606f7bfc8000-606f7bfe9000 rw-p`
`[anon]`	`rw-p`	Анонимные отображения (`mmap`)	`77ec8e605000-77ec8e612000 rw-p`

---

3. Стек и управляющие структуры

Сегмент	Права	Назначение	Пример из `maps`
`[stack]`	`rw-p`	Стек (локальные переменные, вызовы функций)	`7ffd43ed7000-7ffd43ef9000 rw-p`
`[vvar]`	`r--p`	Виртуальные переменные ядра (время и др.)	`77ec8e66d000-77ec8e671000 r--p`
`[vdso]`	`r-xp`	Виртуальный системный вызов (ускорение)	`77ec8e671000-77ec8e673000 r-xp`
`[vsyscall]`	`--xp`	Устаревший механизм системных вызовов	`ffffffffff600000-ffffffffff601000 --xp`

---

4. Библиотеки

Сегмент	Права	Содержимое	Пример из `maps`
`libc.so`	`r-xp`	Код стандартной библиотеки C	`77ec8e428000-77ec8e5b0000 r-xp`
`ld-linux.so`	`r-xp`	Динамический загрузчик	`77ec8e674000-77ec8e69f000 r-xp`

---

5. Специальные сегменты

Сегмент	Права	Назначение	Пример из `maps`
`[stack:guard]`	`---p`	Защитная страница (от переполнения стека)	(Не всегда виден)
`[sigpage]`	`r--p`	Страница для обработки сигналов	(Зависит от архитектуры)

Для чего нужны все эти сегменты?

1. `.text` — хранит код программы. Защищён от записи (`r-xp`), чтобы нельзя было случайно его испортить.

2. `.rodata` — содержит константы. Попытка записи сюда вызовет `Segmentation Fault`.
3. `.data` / `.bss` — хранят глобальные переменные. Отличаются тем, что `.bss` заполняется нулями при запуске.
4. `[heap]` — динамическая память. Растёт вверх (к большим адресам).
5. `[stack]` — растёт вниз (к меньшим адресам). Хранит локальные переменные и адреса возврата.
6. `[vvar]` / `[vdso]` — ускоряют работу с ядром (например, получение времени без переключения в режим ядра).
7. **Библиотеки (`libc.so`, `ld-linux.so`)** — содержат общий код, чтобы его не дублировать в каждой программе.

Что было бы, если бы этих сегментов не было?

1. **Без `.text` и `.rodata`** — код и константы можно было бы случайно перезаписать (краш программы).
2. **Без `.data` / `.bss`** — глобальные переменные пришлось бы хранить в куче (медленнее).
3. **Без `[heap]`** — нельзя было бы использовать `malloc` / `new` (только статическую память).
4. **Без `[stack]`** — не работали бы локальные переменные и вызовы функций.
5. **Без `[vdso]`** — системные вызовы (например, `gettimeofday`) работали бы медленнее.

Как это связано с заданием?

В вашем `/proc/<PID>/maps` видны все эти сегменты. Например:

- Глобальные переменные лежат в `rw-p` регионе (`.data` / `.bss`).
- Локальные переменные — в `[stack]`.
- Кучу (`[heap]`) вы не использовали, но она есть (хоть и пустая).

Если бы не было виртуальной памяти, все эти сегменты смешались бы в одной физической RAM, и программы постоянно бы ломали друг друга.

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>

void function_with_vars(){
```

```
int local_var = 12;

printf("Локальная переменная %d: по адресу = %p\n",local_var, (void*)&local_var);



static int static_var = 20;

printf("Статическая переменная %d: по адресу = %p\n",static_var, (void*)&static_var);



const int const_var = 34;

printf("Константная переменная %d: по адресу = %p\n",const_var,(void*) &const_var);

}

int global_initialized = 42;

int global_uninitialized;

const int global_const = 100;


int main() {

printf("=== Адреса переменных ===\n");

function_with_vars();

printf("Глобальная инициализированная переменная %d: адрес = %p\n",global_initialized,
(void*)&global_initialized);

printf("Глобальная неинициализированная переменная %d: адрес = %p\n",global_uninitialized,
(void*)&global_uninitialized);

printf("Глобальная константа %d: адрес = %p\n",global_const, (void*)&global_const);


return 0;

}
```

b. Сопоставьте адреса переменных с областями адресного пространства из соответствующего /proc//maps. Объясните увиденное.
нам нужно "поймать" программу, поэтому запускаем

Note

nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/4$ ./addr_1 &
[1] 14732
= Адреса переменных =
Локальная переменная 12: по адресу = 0×7ffd43ef77e0
Статическая переменная 20: по адресу = 0×606f5583d014
Константная переменная 34: по адресу = 0×7ffd43ef77e4
Глобальная инициализированная переменная 42: адрес = 0x606f5583d010

```
Глобальная неинициализированная переменная 0: адрес = 0x606f5583d01c
Глобальная константа 100: адрес = 0x606f5583b0e0

nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/4$ ps aux | grep addr_1
nekoie 14732 0.0 0.0 2684 1508 pts/1 S 23:17 0:00 ./addr_1
nekoie 14749 0.0 0.0 6576 2360 pts/1 S+ 23:17 0:00 grep --color=auto addr_1

nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/4$ cat /proc/14732/maps
606f55839000-606f5583a000 r--p 00000000 103:0b 802072 /home/nekoie/Рабочий
стол/programm_on_cpp/OCi/4/addr_1
606f5583a000-606f5583b000 r-xp 00001000 103:0b 802072 /home/nekoie/Рабочий
стол/programm_on_cpp/OCi/4/addr_1
606f5583b000-606f5583c000 r--p 00002000 103:0b 802072 /home/nekoie/Рабочий
стол/programm_on_cpp/OCi/4/addr_1
606f5583c000-606f5583d000 r--p 00002000 103:0b 802072 /home/nekoie/Рабочий
стол/programm_on_cpp/OCi/4/addr_1
606f5583d000-606f5583e000 rw-p 00003000 103:0b 802072 /home/nekoie/Рабочий
стол/programm_on_cpp/OCi/4/addr_1
606f7bfc8000-606f7bfe9000 rw-p 00000000 00:00 0 [heap]
77ec8e400000-77ec8e428000 r--p 00000000 103:0a 2925819 /usr/lib/x86_64-linux-gnu/libc.so.6
77ec8e428000-77ec8e5b0000 r-xp 00028000 103:0a 2925819 /usr/lib/x86_64-linux-gnu/libc.so.6
77ec8e5b0000-77ec8e5ff000 r--p 001b0000 103:0a 2925819 /usr/lib/x86_64-linux-gnu/libc.so.6
77ec8e5ff000-77ec8e603000 r--p 001fe000 103:0a 2925819 /usr/lib/x86_64-linux-gnu/libc.so.6
77ec8e603000-77ec8e605000 rw-p 00202000 103:0a 2925819 /usr/lib/x86_64-linux-gnu/libc.so.6
77ec8e605000-77ec8e612000 rw-p 00000000 00:00 0
77ec8e654000-77ec8e657000 rw-p 00000000 00:00 0
77ec8e66b000-77ec8e66d000 rw-p 00000000 00:00 0
77ec8e66d000-77ec8e671000 r--p 00000000 00:00 0 [vvar]
77ec8e671000-77ec8e673000 r-xp 00000000 00:00 0 [vdso]
77ec8e673000-77ec8e674000 r--p 00000000 103:0a 2925816 /usr/lib/x86_64-linux-gnu/ld-linux-x86-
64.so.2
77ec8e674000-77ec8e69f000 r-xp 00001000 103:0a 2925816 /usr/lib/x86_64-linux-gnu/ld-linux-x86-
64.so.2
77ec8e69f000-77ec8e6a9000 r--p 0002c000 103:0a 2925816 /usr/lib/x86_64-linux-gnu/ld-linux-x86-
64.so.2
77ec8e6a9000-77ec8e6ab000 r--p 00036000 103:0a 2925816 /usr/lib/x86_64-linux-gnu/ld-linux-x86-
64.so.2
77ec8e6ab000-77ec8e6ad000 rw-p 00038000 103:0a 2925816 /usr/lib/x86_64-linux-gnu/ld-linux-x86-
64.so.2
7ffd43ed7000-7ffd43ef9000 rw-p 00000000 00:00 0 [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

Переменная	Адрес	Регион памяти	Сегмент	Права
Локальная переменная	`0x7ffd43ef77e0`	`[stack]`	Стек	`rw-p`
Статическая переменная	`0x606f5583d014`	`.data` / `.bss`	Данные	`rw-p`
Константа в функции	`0x7ffd43ef77e4`	`[stack]`	Стек	`rw-p`
Глобальная инициализированная	`0x606f5583d010`	`.data`	Данные	`rw-p`
Глобальная неинициализированная	`0x606f5583d01c`	`.bss`	Данные	`rw-p`

Переменная	Адрес	Регион памяти	Сегмент	Права
Глобальная константа	`0x606f5583b0e0`	`.rodata`	Read-only	`r--p`

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

```
gcc memory_layout.c -o memory_layout
nm memory_layout
```

```
000000000000038c r __abi_tag
0000000000004018 B __bss_start
0000000000004018 b completed.0
                 w __cxa_finalize@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
00000000000010f0 t deregister_tm_clones
0000000000001160 t __do_global_dtors_aux
0000000000003da8 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003db0 d _DYNAMIC
0000000000004018 D _edata
0000000000004020 B _end
00000000000012e4 T _fini
00000000000011a0 t frame_dummy
0000000000003da0 d __frame_dummy_init_array_entry
0000000000002330 r __FRAME_END__
00000000000011a9 T function_with_vars
00000000000020e0 R global_const
0000000000004010 D global_initialized
0000000000003fa0 d _GLOBAL_OFFSET_TABLE_
000000000000401c B global_uninitialized
                 w __gmon_start__
0000000000002228 r __GNU_EH_FRAME_HDR
0000000000001000 T _init
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
0000000000001246 T main
                 U printf@GLIBC_2.2.5
                 U puts@GLIBC_2.2.5
0000000000001120 t register_tm_clones
                 U sleep@GLIBC_2.2.5
                 U __stack_chk_fail@GLIBC_2.4
00000000000010c0 T _start
0000000000004014 d static_var.0
0000000000004018 D __TMC_END__
```

1. Основные секции и их значение

Символ	Тип	Значение	Пример из вашего вывода
`T`	Text	Код функции (`.text`)	`0000000000001246 T main`
`t`	Локальный текст	Локальные функции (не экспортируемые)	`00000000000010f0 t deregister_tm_clones`

Символ	Тип	Значение	Пример из вашего вывода
`D`	Данные (`.data`)	Инициализированные глобальные переменные	`0000000000004000 D __data_start`
`B`	BSS (`.bss`)	Неинициализированные глобальные переменные	`0000000000004018 B __bss_start`
`R`	Read-only (`.rodata`)	Константы	`00000000000020e0 R global_const`
`d`	Локальные данные	Локальные символы (например, для инициализации)	`0000000000003da0 d __frame_dummy_init_array_entry`
`U`	Undefined	Функции из внешних библиотек (например, `libc`)	`U printf@GLIBC_2.2.5`
`r`	Read-only данные	Константы (аналогично `R`)	`000000000000038c r __abi_tag`

---

2. Важные символы в программе

Глобальные переменные:

- `global_initialized` (`D`) — инициализированная глобальная переменная (хранится в `.data`).

```
0000000000004010 D global_initialized
```

- `global_uninitialized` (`B`) — неинициализированная глобальная переменная (хранится в `.bss`).

```
000000000000401c B global_uninitialized
```

- `global_const` (`R`) — глобальная константа (хранится в `.rodata`).

```
00000000000020e0 R global_const
```

Функции:

- `main` (`T`) — точка входа программы (код в `.text`).

```
0000000000001246 T main
```

- `function_with_vars` (`T`) — ваша функция с локальными переменными.

```
00000000000011a9 T function_with_vars
```

- `_start` (`T`) — начальная точка программы (вызывается перед `main`).

```
00000000000010c0 T _start
```

Служебные символы:

- `__bss_start` (`B`) — начало секции `.bss`.

```
0000000000004018 B __bss_start
```

- `_edata` (`D`) — конец секции `.data` и начало `.bss`.

```
0000000000004018 D _edata
```

- `_end` (`B`) — конец секции `.bss`.

```
0000000000004020 B _end
```

3. Локальные (статические) переменные

- `static_var.0` (`d`) — статическая переменная внутри функции (хранится в `.data` или `.bss`).

```
0000000000004014 d static_var.0
```

---

4. Секции инициализации/финализации

- `__do_global_dtors_aux` — код для деинициализации глобальных объектов.

```
0000000000001160 t __do_global_dtors_aux
```

- `_fini` — секция завершения программы.

```
00000000000012e4 T _fini
```

- `_init` — секция инициализации программы.

```
0000000000001000 T _init
```

---

5. Таблица смещений (GOT/PLT)

- `_GLOBAL_OFFSET_TABLE_` (`d`) — используется для PIC-кода (Position Independent Code).

```
0000000000003fa0 d _GLOBAL_OFFSET_TABLE_
```

---

1. **Глобальные переменные**:
   - Инициализированные → `.data`.
   - Неинициализированные → `.bss`.
   - Константы → `.rodata`.
2. **Функции**:
   - Ваш код (`main`, `function_with_vars`) → `.text`.
   - Внешние функции (`printf`, `sleep`) → разрешаются через `libc`.
3. **Служебные символы**:
   - `_start`, `_init`, `_fini` — управляют жизненным циклом программы.
   - `__bss_start`, `_edata`, `_end` — границы секций.
4. **Статические переменные**:
   - Локальные для функций → хранятся в `.data` / `.bss` с уникальными именами (например, `static_var.0`).

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

```c
#include <stdio.h>// в каком случае будет сегментэйшен фэйл если возвращать из функции.

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <assert.h>



int* local_var_from_func(){

int local_var = 1;

printf("Локальная переменная %d адрес = %p\n",local_var,(void*)&local_var);

void * ptr = &local_var;

return ptr;

}

void overwrite_stack_heavy() {

char buffer[1024 * 1024 * 100];

memset(buffer, 0x41, sizeof(buffer));

}



int main() {

printf("\n=== Возврат адреса локальной переменной ===\n");
```

```
int* ptr = local_var_from_func();

printf("Адрес локальной переменной после вызова: %p \n", (void*)ptr);

overwrite_stack_heavy();

*ptr = 123;

printf("(значение: %d - неопределённое поведение!)\n", *ptr);

return 0;

}
```

Почему неопределеное поведение? давайте вспомним курс ЭВМ И ПУ ~~(ПУППУПУПУПУ)~~

Когда мы вызываем функцию из main , то переход в другую функцию предполагает использование стека(помните jmp всякие) , здесь все данные -локальные то есть эта переменная существует только в этом фрейме стека. А что мы еще про стек знаем?, когда функция выполнила код, то указатели rsp rbp перемещаются, тем самым затирая этот стековый фрейм, соответственно и переменную. Она нам попросту не нужна больше, мы все сделали и вернули результат.
Как только функция завершилась, её стековый фрейм становится **свободной областью памяти**.
Именно поэтому поведение не определенно!

В каком именно случае вылезет segmentation fault?
Segmentation fault произойдет, если система обнаружит попытку доступа к недействительной области памяти. Это зависит от состояния стека после завершения функции. - грубо говоря , если мы начнем использовать другую функцию, которая в памяти расположится там, где когда-то была вот эта, то будет как раз эта ошибка.

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;
f. Прокомментируйте работу предыдущего пункта.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <assert.h>
```

```
void heap_operation(){
char* buffer1 = (char*)malloc(100);
strcpy(buffer1,"Hello, world!");
printf("e.ii. Содержимое буфера1: %s\n", buffer1);
free(buffer1);
printf("e.v. Содержимое буфера1 после free: %s \n", buffer1);



char* buffer2 = (char*)malloc(100);
strcpy(buffer2, "Another hello!");
printf("e.viii. Содержимое буфера2: %s\n", buffer2);

char* mid_ptr = buffer2 + 50;
printf("e.x. Освобождение середины буфера \n");
free(mid_ptr);

printf("e.xi. Содержимое буфера2 после некорректного free: %s (мусор)\n", buffer2);


}
int main() {
printf("\n=== Работа с кучей ===\n");
heap_operations();
return 0;
}
```

Как работают `malloc()` и `free()` под капотом

`malloc()` и `free()` — это функции менеджера динамической памяти (кучи). Они управляют выделением и освобождением памяти в процессе работы программы. Давайте разберём их устройство.

`malloc` — это функция стандартной библиотеки C, которая выделяет блок памяти на куче (heap). Под капотом она взаимодействует с операционной системой через системные вызовы, такие как `brk` или `mmap`.

`free` освобождает ранее выделенную память, помечая её как **свободную** для повторного использования. Она также обновляет метаданные.

Как работает метка "свободной" памяти?

Когда вы вызываете `malloc`, он не просто выделяет память — он также хранит **метаданные** о каждом блоке памяти. Эти метаданные содержат информацию о размере блока, его состоянии (занят/свободен) и других атрибутах. Когда вы вызываете `free`, эти метаданные обновляются, чтобы пометить блок как свободный.

```
typedef struct block {
    size_t size;          // Размер блока данных (без учёта метаданных)
    int free;             // Флаг: 1 = свободен, 0 = занят
    struct block* next;   // Указатель на следующий блок
} block_t;
```

Контракт `malloc` / `free`

Контракт между `malloc` и `free` включает несколько важных правил:

1. **Правильный указатель** :
   - Указатель, передаваемый в `free`, должен быть получен из `malloc` или одной из его производных (`calloc`, `realloc`).
   - Нельзя освобождать указатель дважды (double free).
   - Нельзя освобождать указатель, указывающий на середину блока.
2. **Неопределённое поведение** :
   - Если нарушить контракт, поведение программы становится неопределённым (undefined behavior).
   - Например, если освободить уже освобождённый блок, это может привести к ошибке или повреждению данных.
3. **Управление памятью** :
   - После вызова `free` память больше не принадлежит программе. Любые попытки использовать этот указатель приводят к неопределённому поведению.

1. Основные концепции

Куча (Heap)

- Это область памяти, где динамически выделяются блоки (например, через `malloc`).
- Управляется **аллокатором** (часть стандартной библиотеки C, например, `glibc`).
- **Аллокатор** (от англ. *allocator* — «распределитель») — это компонент программы или операционной системы, который управляет **динамическим выделением и освобождением памяти** (обычно в куче — *heap*).
- Растёт в сторону увеличения адресов (в отличие от стека).

Метаданные

- Аллокатор хранит скрытую информацию о каждом выделенном блоке:
  - Размер блока.
  - Статус (свободен/занят).
  - Ссылки на соседние блоки (в некоторых реализациях).

---

2. Как работает `malloc()`

Когда вы вызываете `malloc(100)`:

Шаг 1: Проверка кеша свободных блоков

- Аллокатор сначала проверяет **список свободных блоков** (free list), чтобы найти подходящий по размеру.
- Если есть свободный блок ≥ 100 байт — он используется (может быть разделён, если слишком большой).

Шаг 2: Запрос памяти у ОС (если нужно)

- Если нет подходящего свободного блока, аллокатор запрашивает память у ядра через:
  - `brk()` — увеличивает границу кучи (работает с одним непрерывным регионом).
  - `mmap()` — выделяет отдельные страницы (для больших блоков).
- ОС выделяет **целые страницы** (обычно 4 КБ), даже если запрошено меньше.

Шаг 3: Разметка блока

- Аллокатор сохраняет **метаданные** перед выделенной областью (например, размер):

```
struct malloc_chunk {
    size_t size;       // Размер блока (включая метаданные)
    struct chunk *next;// Ссылка на следующий блок (в free list)
    int is_free;       // Флаг свободен/занят
};
```

- Пользователь получает указатель **на данные** (после метаданных).

Шаг 4: Возврат указателя

- `malloc()` возвращает адрес **первого байта после метаданных**.
- Например:

```
void *malloc(size_t size) {
    // ...
    return (void*)((char*)chunk + sizeof(struct malloc_chunk));
}
```

```
# Упрощенная реализация malloc
```

```c
      #include <unistd.h>
#include <stdio.h>

typedef struct block {
    size_t size;          // Размер блока
    int free;             // Флаг: свободен ли блок
    struct block* next;   // Указатель на следующий блок
} block_t;

#define BLOCK_SIZE sizeof(block_t)

void* malloc(size_t size) {
    static block_t* head = NULL; // Голова списка блоков
    block_t* current = head;

    // Ищем свободный блок подходящего размера
    while (current) {
        if (current->free && current->size >= size) {
            current->free = 0; // Помечаем блок как занятый
            return (void*)(current + 1); // Возвращаем указатель на данные
        }
        current = current->next;
    }

    // Если свободного блока нет, запрашиваем память у ОС
    size_t total_size = size + BLOCK_SIZE;
    block_t* new_block = sbrk(total_size);
    if (new_block == (void*)-1) {
        return NULL; // Ошибка выделения памяти
    }

    // Инициализируем новый блок
    new_block->size = size;
    new_block->free = 0;
    new_block->next = NULL;

    // Добавляем блок в список
    if (!head) {
        head = new_block;
    } else {
        block_t* last = head;
        while (last->next) {
            last = last->next;
        }
        last->next = new_block;
    }

    return (void*)(new_block + 1); // Возвращаем указатель на данные
}
```

Немного важных определений!!
Аллокаторы (или менеджеры памяти) — это компоненты, которые управляют выделением и освобождением памяти в программе.

Что такое аллокатор?

Аллокатор — это механизм или библиотека, которая предоставляет функциональность для:

1. **Выделения памяти** (например, через `malloc`).
2. **Освобождения памяти** (например, через `free`).
3. **Управления фрагментацией памяти** .
4. **Оптимизации производительности** при работе с памятью.

Аллокаторы работают на уровне кучи (heap) и обеспечивают абстракцию над системными вызовами, такими как `brk` и `mmap`, которые взаимодействуют с операционной системой.

Для чего нужны аллокаторы?

1. **Абстракция над системными вызовами** :
   - Программисту не нужно напрямую работать с низкоуровневыми системными вызовами, такими как `brk` или `mmap`. Аллокатор предоставляет удобный интерфейс (`malloc`, `free`).
2. **Управление фрагментацией памяти** :
   - При частом выделении и освобождении памяти могут возникать "дыры" (свободные блоки), которые невозможно использовать. Аллокаторы оптимизируют использование памяти, объединяя соседние свободные блоки.
3. **Оптимизация производительности** :
   - Аллокаторы могут кэшировать ранее выделенные блоки памяти, чтобы уменьшить количество обращений к ОС.
   - Они также могут предоставлять специализированные стратегии выделения памяти для разных задач (например, маленькие объекты или большие массивы).
4. **Безопасность** :
   - Современные аллокаторы могут обнаруживать ошибки использования памяти, такие как двойное освобождение (double free) или использование освобождённой памяти.

Какие бывают аллокаторы?

Аллокаторы можно классифицировать по нескольким параметрам:

1. По уровню управления:

- **Стандартный аллокатор** :
  - Реализован в стандартной библиотеке C (`malloc`, `free`).
  - Подходит для большинства задач, но может быть неэффективен в специализированных случаях.
- **Пользовательские аллокаторы** :
  - Разработаны под конкретные задачи (например, для работы с маленькими объектами или многопоточными приложениями).

2. По стратегии выделения памяти:

- **Первый подходящий (First Fit)** :
  - Ищет первый свободный блок достаточного размера.
  - Простой и быстрый, но может привести к фрагментации.
- **Лучший подходящий (Best Fit)** :
  - Ищет самый подходящий свободный блок (минимальный размер, который покрывает запрос).
  - Уменьшает фрагментацию, но медленнее.

- **Худший подходящий (Worst Fit)** :
  - Ищет самый большой свободный блок.
  - Редко используется из-за высокой вероятности фрагментации.

3. По области применения:

- **Общий аллокатор** :
  - Обрабатывает запросы на выделение памяти любого размера.
  - Например, стандартный `malloc`.
- **Специализированный аллокатор** :
  - Оптимизирован для определённых типов данных или сценариев.
  - Например:
    - **Arena Allocator** : Выделяет память большими блоками и позволяет освободить всё сразу.
    - **Pool Allocator** : Предназначен для работы с объектами одного размера (например, маленькие структуры данных).
    - **Slab Allocator** : Используется в ядрах операционных систем для управления памятью в критически важных секциях.

---

3. Как работает `free()`

Когда вы вызываете `free(ptr)` :

Шаг 1: Поиск метаданных

- Аллокатор вычисляет адрес метаданных:

```
struct malloc_chunk *chunk = (struct malloc_chunk*)((char*)ptr - sizeof(struct
malloc_chunk));
```

- Проверяет, что блок действительно был выделен (защита от двойного `free`).

Шаг 2: Помещение в free list

- Блок помечается как **свободный** и добавляется в список свободных блоков.
- Может быть объединён с соседними свободными блоками (coalescing), чтобы избежать фрагментации.

Шаг 3: Возврат памяти ОС (иногда)

- Если освобождён большой блок, аллокатор может вернуть память ядру через:
  - `brk()` — уменьшает границу кучи.
  - `munmap()` — освобождает страницы, выделенные через `mmap`.

---

5. Реализация аллокатора (упрощённая)

Пример структуры и функций:

```c
struct block {
    size_t size;
    struct block *next;
    int free;
};

void *heap_start = NULL;

void* my_malloc(size_t size) {
    struct block *curr = heap_start;
    while (curr) {
        if (curr->free && curr->size >= size) {
            curr->free = 0;
            return (void*)(curr + 1); // Пользовательские данные
        }
        curr = curr->next;
    }
    // Если нет свободного блока — запросить у ОС через sbrk()
    struct block *new_block = sbrk(sizeof(struct block) + size);
    new_block->size = size;
    new_block->free = 0;
    new_block->next = NULL;
    if (!heap_start) heap_start = new_block;
    else {
        curr = heap_start;
        while (curr->next) curr = curr->next;
        curr->next = new_block;
    }
    return (void*)(new_block + 1);
}

void my_free(void *ptr) {
    if (!ptr) return;
    struct block *block_ptr = (struct block*)ptr - 1;
    block_ptr->free = 1;
    // Можно добавить coalescing (объединение свободных блоков)
}
```

Упрощенный free

```c
void free(void* ptr) {
    if (!ptr) return; // Защита от free(NULL)

    block_t* block = (block_t*)ptr - 1; // Находим заголовок блока
    block->free = 1; // Помечаем блок как свободный

    // Здесь можно добавить логику объединения соседних свободных блоков
}
```

---

g. Заведите переменную окружения.

```c
export MY_VAR="Hello, environment!"
```

`export` — это команда в Unix-подобных системах (Linux, macOS), которая делает переменную окружения доступной для дочерних процессов.

Как это работает?

1. Переменные окружения могут быть:
   - **Локальными** : видны только в текущей оболочке (shell).
   - **Экспортированными** : видны и в текущей оболочке, и во всех дочерних процессах (например, программах, запущенных из этой оболочки).

h. Добавьте в вашу программу код, который:
i.распечатывает ее значение;
ii.изменяет его значение;
iii.повторно распечатывает ее значение.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    // i. Распечатываем значение переменной окружения
    const char* env_var = getenv("MY_VAR");
    if (env_var) {
        printf("Initial value of MY_VAR: %s\n", env_var);
    } else {
        printf("MY_VAR is not set.\n");
    }

    // ii. Изменяем значение переменной окружения
    if (setenv("MY_VAR", "New value", 1) != 0) {
        perror("setenv failed");
        return 1;
    }

    // iii. Повторно распечатываем значение
    env_var = getenv("MY_VAR");
    if (env_var) {
        printf("Updated value of MY_VAR: %s\n", env_var);
    } else {
        printf("MY_VAR is not set after update.\n");
    }

    return 0;
}
```

Переменные окружения: связь с виртуальной памятью, назначение и особенности

---

1. Что такое переменные окружения?

Переменные окружения (environment variables) — это строковые пары `ключ=значение`, которые:

- Устанавливаются **внутри процесса** или передаются ему **от родительского процесса** (например, из терминала).
- Хранят настройки, пути, параметры запуска и другую конфигурацию.
- Примеры: `PATH`, `HOME`, `USER`, `MY_ENV_VAR` (как в вашем коде).

2. Где хранятся переменные окружения в памяти?

Расположение в виртуальной памяти процесса

Переменные окружения находятся **в виртуальной памяти процесса** в двух местах:

1. **Стек процесса** (или рядом с ним):
   - При запуске программы ОС помещает переменные окружения **в верхнюю часть виртуальной памяти процесса**, рядом с аргументами командной строки (`argv`).
   - Обычно это область **выше** `argv` и **ниже стека** (stack).
2. **Куча (heap)**:
   - Если переменные изменяются через `setenv()` , они могут копироваться в кучу.

Как это выглядит в памяти?

Вот схема виртуальной памяти процесса:

```
Высокие адреса
+---------------------+
| Аргументы (argv)    | ← Например, "./program arg1"
| Переменные окружения| ← MY_ENV_VAR=value
+---------------------+
| Стек (stack)        | ← Растёт вниз
| ...                 |
| Куча (heap)         | ← Растёт вверх
| ...                 |
| Данные программы    | ← .data, .bss
| Код программы (.text)|
Низкие адреса
```

---

3. Как работают функции для работы с переменными окружения?

`getenv("VAR")`

1. Ищет переменную `VAR` в **области окружения** процесса.
2. Возвращает указатель на её значение (или `NULL` , если переменной нет).

`int setenv(const char *name, const char *value, int overwrite);`

1. - `name` : имя переменной.
   - `value` : новое значение.
   - `overwrite` : если `1` , существующее значение будет перезаписано; если `0` , значение не изменится.

Зачем нужен параметр `overwrite` в `setenv`?

Что делает `overwrite`?

Параметр `overwrite` определяет, нужно ли перезаписывать существующее значение переменной окружения:

- Если `overwrite = 1`: существующее значение будет заменено новым.
- Если `overwrite = 0`: существующее значение останется без изменений.

Зачем это нужно?

Представьте ситуацию, когда вы хотите установить значение переменной окружения, но только если она ещё не была задана. Например:

```
// Установить значение, только если переменная не существует
setenv("MY_VAR", "New value", 0);
```

1. **Повторное чтение переменной:**
   - После изменения переменной окружения с помощью `setenv`, мы снова вызываем `getenv`, чтобы проверить обновлённое значение.

Особенности:

- Переменные окружения **наследуются** от родительского процесса (например, из терминала).
- Изменения через `setenv()` **видны только текущему процессу** (не влияют на родительский процесс).

---

4. Связь с виртуальной памятью

1. **Первоначальное хранение**:
   - При запуске процесса ОС копирует переменные окружения **в его виртуальную память** (обычно рядом со стеком).
   - Это **read-only** область — напрямую менять её нельзя (только через `setenv`).
2. **Динамическое изменение (`setenv`)**:
   - Новые переменные попадают в **кучу (heap)**, так как требуют выделения памяти.
   - Указатель на них сохраняется в **структуре процесса** (в виртуальной памяти).
3. **Доступ через `getenv`**:
   - Функция ищет переменные **в куче** (если они изменялись) или **в исходной read-only области**.

---

5. Для чего используются переменные окружения?

Основные сценарии:

1. **Конфигурация программ**:
   - Например, `PATH` хранит список путей для поиска исполняемых файлов.

- Ваш код может проверять `MY_ENV_VAR` для включения режима отладки.

2. **Передача параметров**:
   - Запуск программы с разными настройками без изменения кода:

```
MY_ENV_VAR=debug ./program
```

3. **Безопасность**:
   - Хранение паролей, API-ключей (но не безопасно — лучше использовать секреты в Docker/Kubernetes).
4. **Управление поведением ОС**:
   - `LANG=ru_RU.UTF-8` задаёт язык программы.
   - `DISPLAY=:0` указывает, куда выводить графику.

---

6. Свойства и особенности

Глобальность vs Локальность

- **Глобальные**: Установлены в системе (например, в `.bashrc`).
- **Локальные**: Установлены только для текущего процесса (через `setenv`).

Жизненный цикл

- Существуют **только во время работы процесса**.
- Родительский процесс (например, терминал) **не видит** изменения через `setenv`.

Безопасность

- **Не подходят для секретов** (видны в `ps -e` и `/proc/<PID>/environ`).
- Вместо этого используйте:
  - Файлы конфигурации с правами `chmod 600`.
  - Механизмы вроде `AWS Secrets Manager`.

Производительность

- Частые вызовы `setenv()` могут **фрагментировать кучу**.
- Лучше читать переменные при старте и сохранять их в кэш.

---

7. Примеры

Установка переменной в терминале

```
export MY_ENV_VAR="Hello"  # Устанавливается для всех дочерних процессов
./program                  # Программа увидит MY_ENV_VAR
```

Чтение в программе

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    char *env = getenv("MY_ENV_VAR");
    printf("MY_ENV_VAR = %s\n", env ? env : "(not set)");
    return 0;
}
```

Изменение в программе (только для неё)

```c
setenv("MY_ENV_VAR", "New value", 1);  // 1 = перезаписать, если есть
```

---

8. Как посмотреть переменные окружения процесса?

В Linux:

```
printenv                  # Все глобальные переменные
cat /proc/<PID>/environ   # Переменные конкретного процесса (замените <PID>)
```

В коде на C

```c
extern char **environ;  // Глобальный массив переменных окружения

for (char **env = environ; *env; env++) {
    printf("%s\n", *env);
}
```

---

9. Что будет, если переменных окружения не будет?

- Программы не смогут получать **гибкие настройки** без перекомпиляции.
- Придётся передавать всё через аргументы командной строки (`argv`), что неудобно для сложных конфигов.
- Невозможно будет **наследовать настройки** (например, `PATH` для поиска бинарников).

- Переменные окружения хранятся **в виртуальной памяти процесса** (сначала в read-only области, затем в куче).
- Используются для **гибкой настройки программ** без изменения кода.
- Изменяются через `setenv()` **только для текущего процесса**.
- Не подходят для хранения секретов (альтернативы: файлы с ограниченным доступом, Vault).

g. Переменная окружения

```
```

```
export MY_ENV_VAR="Initial value"
```

h. Изменение переменной окружения

- `getenv` читает переменную.
- `setenv` изменяет её **только для текущего процесса** (не влияет на родительскую оболочку).

j. Проверка после завершения

Функции `getenv()` и `setenv()`: работа с переменными окружения

Эти функции позволяют программам на C **читать** и **изменять** переменные окружения во время выполнения. Разберём их подробно.

---

1. `getenv()` — чтение переменной окружения

Синтаксис

```
char *getenv(const char *name);
```

- **`name`** — имя переменной (например, `"PATH"`).
- **Возвращает**:
  - Указатель на строку со значением переменной.
  - `NULL`, если переменной не существует.

Пример

```
char *path = getenv("PATH");
if (path) {
    printf("PATH: %s\n", path); // Выведет список путей, например "/usr/bin:/bin"
} else {
    printf("Переменная PATH не установлена.\n");
}
```

!!Эти переменные хранятся в специальной области памяти нового процесса (не на стеке, а в отдельном регионе адресного пространства, который называется **environment block**).!!

Как работает?

1. Ищет переменную в **области памяти процесса**, где хранятся переменные окружения.
2. Возвращает **указатель на статическую строку** (не нужно освобождать память!).

Особенности

- **Только чтение**: нельзя изменять строку, которую вернул `getenv()`.
- **Не потокобезопасна** в некоторых реализациях (лучше не использовать в многопоточных программах без синхронизации).

---

2. `setenv()` — установка переменной окружения

Синтаксис

```
int setenv(const char *name, const char *value, int overwrite);
```

- **`name`** — имя переменной.
- **`value`** — новое значение.
- **`overwrite`**:
  - `1` (или `true`): перезаписать переменную, если она уже существует.
  - `0` (или `false`): оставить старую переменную без изменений.
- **Возвращает**:
  - `0` — успех.
  - `-1` — ошибка (например, не хватило памяти).

Пример

```
setenv("MY_VAR", "Hello", 1); // Установит MY_VAR=Hello (перезапишет, если было)
setenv("MY_VAR", "World", 0); // Проигнорируется, если MY_VAR уже существует
```

Как работает?

1. Если `overwrite=1` и переменная существует — **удаляет старую**.
2. **Выделяет память в куче** под новую строку `name=value`.
3. Добавляет запись в **список переменных окружения процесса**.

Особенности

- Изменения **видны только текущему процессу** (не влияют на родительский процесс, например, терминал).
- **Память** под новые переменные выделяется в **куче (heap)**.
- Если `value` пустая строка (`""`), переменная будет установлена, но с пустым значением.

---

3. Разница между `setenv()` и `putenv()`

- **`setenv("VAR", "value", 1)`**:
  Копирует строки `VAR` и `value` в свою память (безопасно).
- **`putenv("VAR=value")`**:
  Принимает строку в формате `"VAR=value"` и **не копирует её**, а просто сохраняет указатель.
  ⚠ Опасность: если переданная строка будет изменена или освобождена — UB!

```
char buffer[] = "MY_VAR=test";
putenv(buffer); // OK (buffer не будет удалён)
// strcpy(buffer, "XXX"); // Опасно! Может повредить переменную.

setenv("MY_VAR", "test", 1); // Лучше: делает копию строки.
```

---

4. Удаление переменной: `unsetenv()`

```c
int unsetenv(const char *name);
```

- Удаляет переменную `name` из окружения процесса.
- Всегда возвращает `0` (даже если переменной не было).

Пример

```c
unsetenv("MY_VAR"); // Теперь getenv("MY_VAR") вернёт NULL
```

---

5. Когда использовать?

`getenv()`

- Проверка настроек (например, режим отладки):

```c
if (getenv("DEBUG")) {
    printf("Режим отладки включён!\n");
}
```

- Получение системных путей (`PATH` , `HOME`).

`setenv()`

- Динамическое изменение поведения программы:

```c
setenv("LANG", "ru_RU.UTF-8", 1); // Установить язык
```

- Передача параметров в дочерние процессы (через `fork()` + `exec()`).

---

6. Где хранятся переменные?

- **Виртуальная память процесса**:
  - Изначально — в **стеке** (близко к `argv`).
  - После `setenv()` — в **куче**.
- Посмотреть все переменные:

```bash
printenv          # В терминале
cat /proc/$PID/environ  # Для процесса (замените $PID)
```

---

7. Важные предупреждения

1. **Не храните секреты** в переменных окружения (видны в `ps -e`).
2. **Не злоупотребляйте** `setenv()` — частая смена переменных может фрагментировать кучу.
3. **Потокобезопасность**: `getenv()` / `setenv()` могут быть небезопасны в многопоточных программах (используйте мьютексы).

Пример из вашего кода

```
char* env_value = getenv("MY_ENV_VAR"); // Чтение
setenv("MY_ENV_VAR", "New value!", 1);  // Изменение (только для текущего процесса)
```

- После `setenv()` последующие вызовы `getenv("MY_ENV_VAR")` вернут `"New value!"`.
- **Но** если запустить программу снова — переменная вернётся к исходному значению (если не установлена в оболочке).

- `getenv()` — читает переменные окружения.
- `setenv()` — изменяет их **для текущего процесса**.
- `unsetenv()` — удаляет переменную.
- Изменения **не затрагивают родительский процесс** (например, терминал).

i. Запустите вашу программу и убедитесь что переменная окружения
имеет требуемое значение.

j. Выведите значение переменной окружения после того как ваша
программа завершилась.

```
echo $MY_ENV_VAR  # Будет "Initial value" (программа меняла свою копию)
```

k. Объясните произошедшее.

- Переменные окружения копируются в процесс при запуске.
- `setenv` меняет их только внутри процесса, не затрагивая родительскую оболочку.

2. Управление адресным пространством:

a. Напишите программу, которая:
i.
выводит pid процесса;
ii.
ждет одну секунду;
iii.
делает exec(2) самой себя;
iv.
выводит сообщение “Hello world”

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    // i. Выводим PID процесса
    printf("PID: %d\n", getpid());

    // ii. Ждём одну секунду
    sleep(1);

    // iii. Выполняем exec самой себя
    printf("Executing self...\n");
    execl("/proc/self/exe", "/proc/self/exe", NULL);

    // iv. Выводим сообщение (никогда не выполнится)
    printf("Hello world\n");

    return 0;
}
```

1. **`getpid()`** :
   - Возвращает идентификатор текущего процесса (PID).
   - Это системный вызов, который предоставляет информацию о процессе.
2. **`sleep(1)`** :
   - Приостанавливает выполнение программы на 1 секунду.
   - Это нужно, чтобы вы успели проверить содержимое `/proc/<pid>/maps`.
3. **`execl`** :
   - Заменяет текущий процесс новым образом программы.
   - После вызова `exec` текущая программа полностью перезаписывается новой программой.
   - Код после `exec` не выполняется, если только вызов не завершился ошибкой.
4. **Файл `/proc/<pid>/maps`** :
   - Содержит информацию об адресном пространстве процесса.
   - Вы можете наблюдать, как меняются регионы памяти до и после вызова `exec`.

Наблюдения:

- Перед вызовом `exec` в `/proc/<pid>/maps` будут видны регионы для текстового сегмента (код), данных (глобальные переменные) и стека.
- После вызова `exec` адресное пространство будет полностью перезаписано новым образом программы.

---

b. Понаблюдайте за выводом программы и содержимым соответствующего файла /proc//maps. Объясните происходящее.

Запустите программу и одновременно отслеживайте изменения в `/proc/<pid>/maps`:

```
watch -n 1 cat /proc/<pid>/maps
```

- Вы увидите, как адресное пространство изменяется после вызова `exec`.
- Все старые регионы памяти будут заменены новыми.

c. Напишите программу, которая:
i.
выводит pid процесса;
ii.
ждет 10 секунд (подберите паузу чтобы успеть начать мониторить адресное пространство процесса, например, watch cat /proc//maps);
iii.
напишите функцию, которая будет выделять на стеке массив (например, 4096 байт) и рекурсивно вызывать себя;
iv.
понаблюдайте как изменяется адресное пространство процесса (стек);
v.
напишите цикл, в котором на каждой итерации будет выделяться память на куче (подберите размер буфера сами). Используйте секундную паузу между итерациями.
vi.
понаблюдайте как изменится адресное пространство процесса (heap);
vii.
освободите занятую память.
viii.
присоедините к процессу еще один регион адресов размером в 10 страниц (используйте mmap(2) с флагом ANONYMOUS).
ix.
понаблюдайте за адресным пространством.x.
измените права доступа к созданному региону и проверьте какая будет реакция, если их нарушить:

3. запретите читать данные и попробуйте прочитать из региона.
4. запретите писать и попробуйте записать.
   xi.
   попробуйте перехватить сигнал SIGSEGV.
   xii.
   отсоедините страницы с 4 по 6 в созданном вами регионе.
   xiii.
   понаблюдайте за адресным пространством.

```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/mman.h>

#include <signal.h>

#include <string.h>
```

```c
#define PAGE_SIZE 4096


volatile sig_atomic_t segv_caught = 0;


void recursive_stack_allocation(int depth) {

char stack_array[PAGE_SIZE * 100];

printf("Stack depth: %d, Address: %p\n", depth, (void*)stack_array);


if (depth < 10){

sleep(1);

recursive_stack_allocation(depth + 1);

}

}


void sigsegv_handler(int signum) {

printf("Caught SIGSEGV signal\n");

segv_caught = 1;

}



int main(){

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

printf("watch -d -n1 cat /proc/%d/maps\n", getpid());


sleep(15);

printf("Смотрим на регион стека!\n");

recursive_stack_allocation(0);

printf("Смотрим на регион кучи!\n");

void* heap_buffers[5];

for (int i = 0; i < 10; ++i) {

heap_buffers[i] = malloc(PAGE_SIZE * 1024);
```

```
printf("Heap allocation %d: %p\n", i, heap_buffers[i]);
sleep(1);
}

printf("Освобождаем память из под кучи...\n");
for (int i = 0; i < 5; ++i) {
free(heap_buffers[i]);
printf("Freed buffer %d: %p\n", i, heap_buffers[i]);
sleep(1);
}

printf("Присоединяем анонимный регион 2 страниц\n");
void* anonymous_region = mmap(NULL, 2 * PAGE_SIZE, PROT_READ | PROT_WRITE,
MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
if (anonymous_region == MAP_FAILED) {
perror("mmap failed");
return 1;
}
printf("Смотрим на присоединение региона %p\n", anonymous_region);
sleep(1);
printf("Изменяем права достпа к региону и смортрим реакцию\n");
if (mprotect(anonymous_region, PAGE_SIZE, PROT_NONE) == -1) {
perror("mprotect failed");
return 1;
}
sleep(2);

signal(SIGSEGV, sigsegv_handler);

printf("Попытаемся прочитать из этого региона\n");
char value;
```

```
if (!segv_caught) {
value = *((char*)anonymous_region);
printf("Read value: %c\n", value);
}

printf("Пытаемся записать в регион\n");
printf("Trying to write to protected region...\n");
if (!segv_caught) {
*((char*)anonymous_region) = 'A';
}

printf("Отсоединим с 4 по 6 страницу\n");
if (munmap(anonymous_region + 4 * PAGE_SIZE, 3 * PAGE_SIZE) == -1) {
perror("munmap failed");
return 1;
}

printf("Pages 4-6 unmapped successfully\n");
sleep(15);
}

// gcc cdd_2.c -o cdd
// ./cdd
```

`sleep` — это системный вызов, который приостанавливает выполнение текущего процесса на указанное количество секунд.

- `sleep` использует системный вызов `nanosleep`, который работает с таймерами ядра.
- Если процесс получает сигнал во время сна (например, `SIGINT`), он может быть пробуждён досрочно.

`execl` — это функция из семейства `exec`, которая заменяет текущий образ программы новым образом (исполняемым файлом).

1. Когда вызывается `execl`, ядро уничтожает текущее адресное пространство процесса (текстовый сегмент, данные, стек, куча и т.д.).

- Затем ядро загружает новый образ программы в память.

2. **Загрузка нового образа** :
   - Ядро читает исполняемый файл (например, ELF-формат в Linux).
   - Оно загружает сегменты кода, данных и другие части файла в память.
3. **Инициализация нового процесса** :
   - Стек инициализируется новыми аргументами командной строки и переменными окружения.
   - Указатель инструкции (instruction pointer) устанавливается на точку входа нового образа.
4. **Выполнение нового кода** :
   - После завершения `exec` , процесс начинает выполнять новый код.
   - Если `exec` завершился успешно, старый код больше не выполняется.

`SIGSEGV` (Segmentation Fault) — это сигнал, который отправляется процессу, когда он пытается выполнить недопустимую операцию с памятью.

Когда возникает `SIGSEGV` ?

1. **Обращение к недействительной памяти** :
   - Попытка чтения или записи по недействительному адресу (например, `NULL`).
2. **Нарушение прав доступа** :
   - Попытка записи в регион, доступный только для чтения.
   - Попытка чтения из регионов, помеченных как недоступные (например, после `mprotect`).
3. **Работа с освобождённой памятью** :
   - Использование указателя после вызова `free` .

d. Чтобы было удобнее наблюдать за адресным пространством подберите
удобные паузы между операциями изменяющими его.
e. Объясните что происходит с адресным пространством в данной задаче.

5. Самодельная куча
   a. Реализуйте свою кучу над анонимным регионом адресов:
   i.
   присоедините анонимный регион (mmap(2));
   ii.
   реализуйте функцию my_malloc(), которая:
6. принимает размер памяти в байтах;
7. резервирует буфер запрошенного размера и возвращает
   указатель на его начало;
8. при недостатке памяти возвращает NULL.
   iii.
   реализуйте функцию my_free(), которая:
9. принимает указатель на буфер, возвращенный ранее
   функцией my_malloc();
10. помечает буфер свободным;
    iv.
    Рекомендация. Для отладки можно присоединить регион
    связанный с файлом. Это позволит наблюдать за состоянием
    вашей кучи при выделении-освобождении памяти.

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <sys/mman.h>

#define HEAP_SIZE (1024 * 1024) // Размер кучи: 1 МБ

typedef struct block {
    size_t size;
    int free;
    struct block* next;
} block_t;

static block_t* heap_start = NULL;

void init_heap() {
    // Присоединяем анонимный регион
    heap_start = mmap(NULL, HEAP_SIZE, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1,
0);
    if (heap_start == MAP_FAILED) {
        perror("mmap failed");
        exit(1);
    }

    // Инициализируем первый блок
    heap_start->size = HEAP_SIZE - sizeof(block_t);
    heap_start->free = 1;
    heap_start->next = NULL;
}

void* my_malloc(size_t size) {
    block_t* current = heap_start;

    // Ищем свободный блок подходящего размера
    while (current) {
        if (current->free && current->size >= size) {
            current->free = 0;
            return (void*)(current + 1);
        }
        current = current->next;
    }

    return NULL; // Недостаточно памяти
}

void my_free(void* ptr) {
    if (!ptr) return;

    // Находим заголовок блока
    block_t* block = (block_t*)ptr - 1;
    block->free = 1;
}

int main() {
    init_heap();

    // Выделяем память
    void* ptr1 = my_malloc(100);
    void* ptr2 = my_malloc(200);

    printf("Allocated pointers: %p, %p\n", ptr1, ptr2);

    // Освобождаем память
    my_free(ptr1);
```

```
    my_free(ptr2);

    return 0;
}
```

Вот программа на языке C, которая реализует все пункты задачи (a и c), а также объяснения в конце.

📌 Часть a: exec(2) самой себя

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = getpid();
    printf("PID процесса: %d\n", pid);
    sleep(1);
    execl("./self_exec", "./self_exec", NULL);
    // Если мы здесь, то exec не удался
    perror("exec");
    return 1;
}
```

Сохраните как `self_exec.c`, скомпилируйте:

```
gcc self_exec.c -o self_exec
```

При запуске программа будет бесконечно перезапускать себя через `exec`. Каждый раз адресное пространство сбрасывается заново. Это можно проверить командой:

```
watch cat /proc/<pid>/maps
```

📌 Часть c: работа с адресным пространством

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/mman.h>
#include <string.h>
#include <errno.h>

void recursive_stack(int depth);
void handle_sigsegv(int sig, siginfo_t *si, void *unused);

int main() {
    pid_t pid = getpid();
    printf("PID процесса: %d\n", pid);

    // Ждем 10 секунд перед началом изменений
    printf("Жду 10 секунд...\n");
```

```
    sleep(10);

    // Рекурсия для изменения стека
    printf("Начинаю рекурсивные вызовы (стек)\n");
    recursive_stack(0);

    // Выделение памяти на куче
    printf("Выделяю память на куче...\n");
    #define BUF_SIZE 4096
    char* buffers[10];
    for (int i = 0; i < 10; ++i) {
        buffers[i] = malloc(BUF_SIZE);
        if (!buffers[i]) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        printf("Выделен буфер #%d по адресу: %p\n", i, (void*)buffers[i]);
        sleep(1);
    }

    // Освобождаем память
    for (int i = 0; i < 10; ++i) {
        free(buffers[i]);
        printf("Освобожден буфер #%d\n", i);
        sleep(1);
    }

    // mmap — выделяем 10 страниц
    const size_t PAGE_SIZE = getpagesize();
    const size_t MAP_SIZE = 10 * PAGE_SIZE;

    void* mapped = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (mapped == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    printf("mmap выделен по адресу: %p\n", mapped);

    // Изменяем права доступа: только запись
    if (mprotect(mapped, MAP_SIZE, PROT_WRITE) == -1) {
        perror("mprotect");
        exit(EXIT_FAILURE);
    }
    printf("Права изменены: только запись\n");

    // Устанавливаем обработчик SIGSEGV
    struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handle_sigsegv;
    if (sigaction(SIGSEGV, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    // Попытка чтения из региона без прав чтения
    printf("Попытка чтения из региона без прав:\n");
    char val = *(char*)mapped; // Ожидается SIGSEGV

    // Попытка записи в регион после запрета записи
    printf("Меняю права: только чтение\n");
```

```
    mprotect(mapped, MAP_SIZE, PROT_READ);
    printf("Попытка записи в регион без прав:\n");
    *(char*)mapped = 'A'; // Ожидается SIGSEGV

    // Отсоединяем страницы 4–6
    void* unmap_start = (char*)mapped + 3 * PAGE_SIZE;
    size_t unmap_len = 3 * PAGE_SIZE; // Страницы 4,5,6 (индекс начинается с 0)
    if (munmap(unmap_start, unmap_len) == -1) {
        perror("munmap");
        exit(EXIT_FAILURE);
    }
    printf("Страницы 4-6 отсоединены\n");

    printf("Программа завершена.\n");
    return 0;
}

// Рекурсивная функция для изменения стека
void recursive_stack(int depth) {
    if (depth >= 10) return;
    char stack_array[4096]; // Выделяем массив на стеке
    printf("Рекурсия глубины %d, адрес массива: %p\n", depth, stack_array);
    sleep(1);
    recursive_stack(depth + 1);
}

// Обработчик сигнала SIGSEGV
void handle_sigsegv(int sig, siginfo_t *si, void *unused) {
    printf("Получен сигнал %d (%s), адрес нарушения: %p\n",
           sig, strsignal(sig), si->si_addr);
    exit(EXIT_FAILURE);
}
```

✅ Как компилировать:

```
gcc full_memory_test.c -o memtest -Wall -Wextra -g
```

✅ Как наблюдать за `/proc/<pid>/maps`:

Откройте второй терминал и выполните:

```
watch -n 1 cat /proc/<pid>/maps
```

🧠 Объяснение происходящего:

a. `exec`

- При каждом `exec` текущее адресное пространство заменяется новой программой.
- Файлы `/proc/<pid>/maps` показывают новый layout памяти: код, данные, стек и т.д.

- PID остается тем же, но содержимое памяти полностью меняется.

---

c. Изменения адресного пространства

1. Стек

- При рекурсивных вызовах выделяются фреймы стека.
- В `/proc/maps` это видно по увеличению диапазона `[stack]`.

2. Куча

- `malloc()` использует `brk()` или `mmap()` внутренне.
- При выделении памяти появляются новые области в куче (`heap`).
- При освобождении памяти эти области могут быть слиты или оставлены в пуле.

3. mmap

- Создает новый регион виртуальной памяти.
- Отображается в `/proc/maps` как `[anon]`.
- Размер 10 страниц (обычно 4096 * 10 = 40960 байт).

4. mprotect

- Меняет права доступа к региону памяти.
- При попытке нарушить права (например, чтение при только записи), генерируется сигнал `SIGSEGV`.

5. SIGSEGV

- Обрабатываем сигнал, чтобы увидеть, где произошло нарушение.
- Без обработки программа аварийно завершится.

6. munmap

- Отключает определенные страницы из регионов.
- Эти участки исчезают из `/proc/maps`.

🔁 Паузы между операциями

- Использованы `sleep(1)` между выделением памяти и рекурсией.
- Перед началом активности — `sleep(10)` для удобства наблюдения.

🧪 Что происходит с адресным пространством?

Этап	Что происходит
`exec`	Полная замена адресного пространства
`malloc`	Увеличение heap

Этап	Что происходит
`free`	Освобождение памяти (не всегда уменьшение heap)
`mmap`	Добавление нового anon-региона
`mprotect`	Изменение прав доступа к региону
`SIGSEGV`	Защита от недозволенного доступа
`munmap`	Удаление части регионов

Если нужно — могу предоставить bash-скрипт для автоматического мониторинга `/proc/<pid>/maps`.

Хочешь версию программы с комментариями на русском?