**ЗАДАНИЕ**

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”

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define ERROR (-1)

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

int ret;

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

sleep(1);

ret = execv(argv[0], argv);

if (ret == ERROR) {

perror("execv");

return EXIT\_FAILURE;

}

printf("Hello world\n");

return EXIT\_SUCCESS;

}

**Что делает execv(argv[0], argv)?**

execv() — это системный вызов, который говорит ОС: «Замените весь код, данные и стек текущего процесса на другую программу. Но сам процесс (его ID и место в системе) оставьте тем же.»

Это не «запустить новую программу» как fork() — **это подмена текущего процесса на другую программу.**

**Что значит argv[0], argv?**

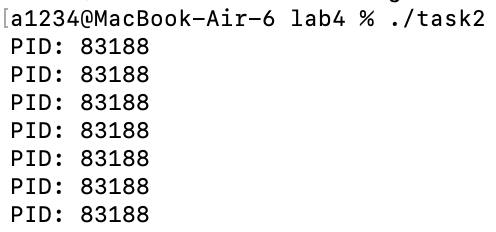
argv[0] — это имя текущего исполняемого файла (сама программа).

argv — это массив аргументов, которые мы хотим передать новой программе.

"Загрузи эту же программу заново и передай ей те же аргументы."

**Почему PID остаётся тем же?**

Потому что execv() не создаёт новый процесс. Он работает внутри текущего процесса. Просто удаляет старое содержимое (код, данные, стек) и загружает новое из файла. С точки зрения ядра: Таблица процесса, его PID, открытые файлы, права доступа — всё остаётся как было. Меняется только «что делает» этот процесс.



*b. Понаблюдайте за выводом программы и содержимым соответствующего*

*файла /proc/<pid>/maps. Объясните происходящее.*

**Почему printf("Hello world") не выводится?**

Потому что как только execv() успешно вызывается — старый код полностью уничтожается.

Загружается новая программа.

Ничего, что было после execv(), уже не выполняется.

Это значит: "Hello world" не выведется.

**Что происходит по шагам?**

1. Программа стартует.

2. Печатает PID.

3. Ждет 1 секунду — в это время ты можешь, например, запустить ps или top и посмотреть PID.

4. Через 1 секунду программа вызывает execv():

5. Загружается новая копия самой себя.

Все старые переменные, стек, heap, данные — сбрасываются.

6. Программа начинает с самого начала — печатает PID, ждет 10 секунд...

Это бесконечный цикл, если не остановить (каждый раз замещает себя новой копией).

**ПРИМЕЧАНИЕ**. Для следующих манипуляций удобно поменять значение в sleep(), например, на 20.

1) Локально у себя на компьютере создаем файл

2) Компилируем

gcc main.c -o main3) Запускаем

./main4) Нам вывелся pid процесса

5) Открываем НОВОЕ окно терминала, переходим в ту же директорию, пишем

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

6) Ждем, пока произойдет exec() (через то количество секунд, которое мы указали в sleep), нам опять вывелся тот же pid

7) Во втором окне опять пишем

vmmap <pid>  
**ИЛИ можно при запуске писать &**

**Как мы можем заметить, pid процесса остался одинаковым, но адреса поменялись.**

**Что происходит до exec()?**

Когда ты запускаешь программу, она получает память для своей работы. Вот, что происходит:

Сегмент \_\_TEXT — это сегмент с кодом программы. Он загружается в память с флагами r-x (чтение и выполнение).

Сегмент \_\_DATA\_CONST — содержит данные, такие как константы и глобальные переменные.

Сегмент \_\_LINKEDIT — это таблица символов и информации о динамических библиотеках.

На момент первого vmmap ты видишь эти данные, так как программа ещё не выполнила exec() и не изменила свой адресный пространство.

**Что происходит после вызова exec()?**

Когда ты вызываешь execv():

Программа заменяет себя новой копией той же самой программы или другой, указанной в execv().

Все данные, которые были в исходной программе (переменные, стек, сегменты), удаляются.

Вместо этого подгружается новая программа с новым набором сегментов в памяти.

Когда происходит exec(), память, которая была у программы до этого, перезаписывается:

Сегменты (например, \_\_TEXT, \_\_DATA\_CONST) будут загружены заново.

Адреса в памяти изменятся, потому что ядро ОС загрузит новый процесс в новые области памяти.

**Почему адреса изменяются?**

После exec() адреса в виртуальной памяти изменяются, потому что:

Каждый раз, когда программа запускается, ей выделяется новое пространство памяти.

Память, которая была раньше выделена процессу, освобождается, и программа получает новые адреса.

Это поведение происходит, потому что виртуальная память у процесса изолирована. Таким образом, даже если мы запускаем ту же программу, адресное пространство будет новым.

**В чём отличие exec() от обычного вызова программы?**

Если бы ты просто запустила программу (например, через терминал), она получила бы новое адресное пространство и новый PID. В случае с exec():

Программа не умирает.

Текущий процесс сохраняет свой PID, но его адресное пространство перезаписывается новой программой.

*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/<pid>/maps);*

*iii. напишите функцию, которая будет выделять на стеке массив*

*(например, 4096 байт) и рекурсивно вызывать себя;*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#define PAGE\_SIZE 4096

void AllocateOnStack() {

char buffer[PAGE\_SIZE];

printf("Allocation: %p\n", &buffer[0]);

sleep(1);

AllocateOnStack();

}

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

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

sleep(10);

AllocateOnStack();

return EXIT\_SUCCESS;

}

**Что такое стек и куча?**

**Стек (Stack)**: Это область памяти, которая используется для хранения локальных переменных и для управления вызовами функций. Память на стеке выделяется и освобождается автоматически при входе и выходе из функции.

**Куча (Heap)**: Это область памяти, в которой хранятся динамически выделяемые данные (например, при использовании функций malloc или calloc). Память на куче нужно освобождать вручную с помощью функции free.

**Как понять, что массив выделяется на стеке?**

Массив в функции AllocateOnStack выделяется с помощью синтаксиса char buff[PAGE\_SIZE];. Это означает, что массив является локальной переменной, и он будет выделен на стеке, а не на куче.

Когда мы пишем:

printf("Allocation: %p\n", &buff[0]);

Мы хотим вывести адрес первого элемента массива buff, а не сам массив в целом.

buff[0] — это первый элемент массива, т.е. переменная типа char.

&buff[0] — это адрес этого первого элемента, который является типом char \*.

Таким образом, когда мы пишем &buff[0], мы получаем адрес первого байта массива. Это именно то, что нам нужно, если мы хотим увидеть, где в памяти начинается этот массив.

**Что происходит в программе?**

Мы рекурсивно вызываем ту же функцию, и таким образом, каждый новый вызов будет выделять новый массив на стеке.

Важно понимать, что каждый новый вызов функции создает **новый фрейм стека**. Стек — это область памяти, которая используется для хранения локальных переменных, вызовов функций, а также адресов возврата. Когда мы рекурсивно вызываем функцию, мы добавляем новые фреймы на стек. Стек **растет вниз по памяти** (от более высоких адресов к более низким).

**Наблюдение за адресом стека.**

В процессе рекурсивного вызова стек будет расти, и каждый новый фрейм будет занимать место в памяти. Поскольку адреса в стеке будут изменяться по мере увеличения количества рекурсивных вызовов, важно отслеживать, как это происходит.

Мы выводим адрес первого элемента массива с помощью: printf("Allocation: %p\n", &buff[0]);

Этот вывод покажет вам адрес, на котором находится массив buff в текущем фрейме стека. Каждый новый вызов функции будет выделять новый блок памяти на стеке, и адреса этих блоков будут располагаться ниже, чем в предыдущем вызове.

**Что это нам даст?**

Понимание роста стека. Мы можем увидеть, как накапливаются данные в стеке с каждым вызовом функции. Адреса, по которым эти данные хранятся, будут показывать, как стек "растет" вниз по памяти.

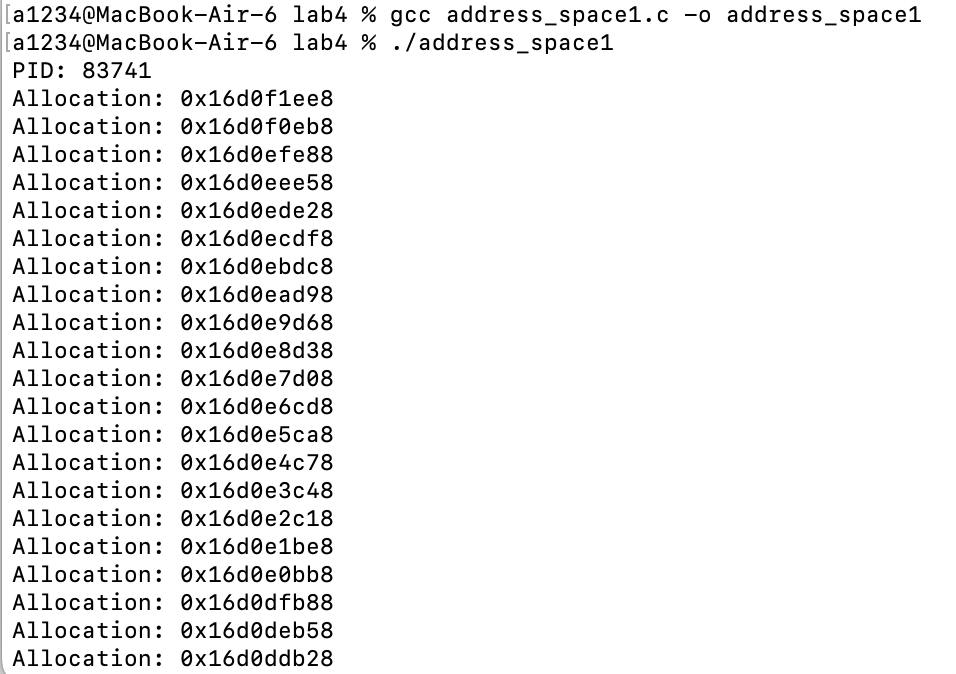
Демонстрация переполнения стека. Если вы будете слишком глубоко рекурсировать, вы столкнетесь с ошибкой переполнения стека (stack overflow), что является важным практическим аспектом работы с памятью в языке C.

**Почему создается новый фрейм, а не повторно используется старый?**

Это связано с тем, как работает стек в операционной системе. Каждый раз, когда функция вызывает саму себя (или другую функцию), она должна иметь свой собственный набор локальных данных и свой адрес возврата. Если бы мы использовали один и тот же фрейм стека для всех вызовов, то каждый новый вызов перезаписал бы данные из предыдущего, что привело бы к непредсказуемому поведению программы.

*iv. понаблюдайте как изменяется адресное пространство процесса*

*(стек);*



Потом будет segmentation fault когда стек заполнится (можно убрать строку sleep(1) и увидеть это)

*v. напишите цикл, в котором на каждой итерации будет выделяться*

*память на куче (подберите размер буфера сами). Используйте*

*секундную паузу между итерациями.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#define PAGE\_SIZE 4096

void AllocateOnHeap() {

while (1) {

char \*buffer = malloc(PAGE\_SIZE);

if (buffer == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

printf("Allocation: %p\n", &buffer[0]);

sleep(1);

}

}

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

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

sleep(10);

AllocateOnHeap();

return EXIT\_SUCCESS;

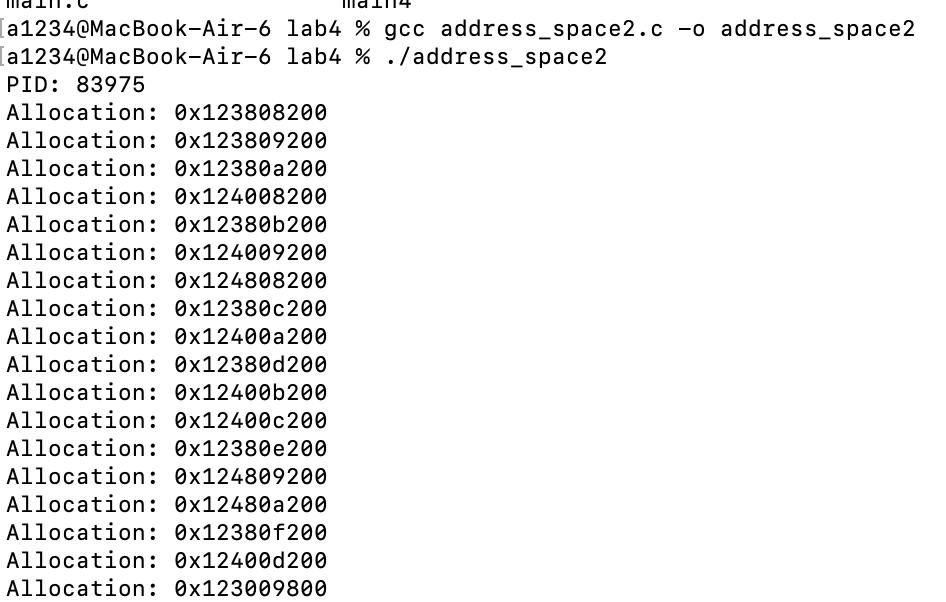
}

**Что происходит с адресным пространством?**

Каждый раз, когда вы вызываете malloc(), выделяется новый блок памяти на куче. Память будет выделяться по мере выполнения программы, и вы будете видеть, как адресное пространство растет с каждой итерацией. Со временем программа может исчерпать доступную память, что приведет к ошибке выделения памяти, если система не может выделить достаточно памяти.

*vi. понаблюдайте как изменится адресное пространство процесса*

*(heap);*



**Почему происходит "перепрыгивание" между адресами?**

Когда операционная система или распределитель памяти выделяет блоки памяти, она может использовать разные механизмы, чтобы уменьшить фрагментацию памяти и эффективно использовать доступные области памяти.

Эти блоки могут выделяться в разных регионах памяти, и это нормально. Так как память на куче управляется виртуально, адреса не обязательно будут подряд, как это бывает с переменными на стеке.

Адреса с разными префиксами:

Когда вы видите, что одни адреса начинаются с 0x123..., а другие с 0x124..., это значит, что система выделяет память в разных частях виртуального адресного пространства.

Это может быть связано с тем, как операционная система управляет памятью и как процесс использует разные сегменты адресного пространства для динамического выделения памяти.

**В чем причина разности в адресах?**

1. Виртуальное адресное пространство: Каждый процесс имеет своё виртуальное адресное пространство, и для разных процессов операционная система может предоставлять разные диапазоны виртуальных адресов. Таким образом, адреса, начинающиеся с разных префиксов (0x123 и 0x124), могут указывать на разные области памяти в виртуальном адресном пространстве.

2. Фрагментация кучи: В случае динамического выделения памяти с помощью malloc система может выделять память в местах, которые не являются соседними, чтобы уменьшить фрагментацию и эффективно использовать пространство.

*viii. присоедините к процессу еще один регион адресов размером в 10 страниц (используйте mmap(2) с флагом ANONYMOUS).*

*ix. понаблюдайте за адресным пространством.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#define SUCCESS 0

#define ERROR (-1)

#define PAGE\_SIZE 4096

#define NO\_FD (-1)

void AllocateAddressRegion() {

int ret = SUCCESS;

char \*address\_region;

address\_region = mmap(NULL, 10 \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

if (address\_region == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

printf("Created address region: %p\n", &address\_region[0]);

sleep(10);

}

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

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

sleep(10);

AllocateAddressRegion();

return EXIT\_SUCCESS;

}

**КАК ЗАПУСКАТЬ:**

1) Открываем первое окно терминала, переходим в папку, создаем файл

2) Компилируем программу

3) Открываем второе окно терминала, переходим в ту же папку

4) В первом окне терминала запускаем программу, дожидаемся пока нам выведет PID

5) Во втором окне терминала ПИШЕМ (но пока НЕ отправляем!!!)

vmmap <pid>где вместо <pid> указываем PID, который нам вывело ранее

6) Когда в первом окне терминала появилась строчка

Created address region: ...во втором окне нажимаем ENTER (отправляем наш vmmap)

7) Смотрим на выделенную область памяти (ищем строку VM\_ALLOCATE или MALLOC (admin) или Mapped (anonymous))

**Объяснение кода:**

1) addr\_region = mmap(NULL, 10 \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

**mmap используется для выделения области памяти.**

NULL — указывает, что операционная система сама выберет адрес в виртуальном адресном пространстве.

10 \* PAGE\_SIZE — размер региона памяти, который мы выделяем. В данном случае это 10 страниц по 4096 байт (40960 байт).

PROT\_READ | PROT\_WRITE | PROT\_EXEC — права доступа: разрешено читать, писать и выполнять код.

MAP\_PRIVATE | MAP\_ANONYMOUS — флаги, указывающие, что память будет анонимной (не связана с файлом), и доступна только для текущего процесса.

NO\_FD и 0 — параметры, которые не используются в данном случае, так как память не привязана к файлу.

Если mmap не удается выделить память, возвращается MAP\_FAILED. В этом случае выводится ошибка и программа завершает выполнение.

**Общая сигнатура mmap:**

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

**Параметры:**

1. addr: Ожидаемый адрес, с которого начнется отображение. Если указать NULL, то система выберет адрес автоматически.

2. length: Размер области памяти, которую необходимо отобразить. Обычно это кратно размеру страницы (например, 4096 байт).

3. prot: Права доступа к памяти. Может быть одно или несколько значений из следующего списка:

PROT\_READ — разрешает чтение.

PROT\_WRITE — разрешает запись.

PROT\_EXEC — разрешает выполнение.

PROT\_NONE — запрещает доступ.

4. flags: Опции для отображения. Несколько флагов могут быть комбинированы через побитовую операцию OR (|).

MAP\_PRIVATE — создается частная копия памяти (изменения не будут видны другим процессам).

MAP\_SHARED — разделяемая память (изменения видны другим процессам).

MAP\_ANONYMOUS — память не связана с файлом (не используется дескриптор файла).

MAP\_FIXED — указывает, что отображение должно происходить строго по адресу, указанному в параметре addr.

5. fd: Дескриптор файла, с которым связана память. Если используется флаг MAP\_ANONYMOUS, то значение этого параметра должно быть -1, так как память не будет связана с файлом.

6. offset: Смещение в файле, с которого начинается отображение. Обычно указывается как 0 для отображения всего файла. Если используется флаг MAP\_ANONYMOUS, этот параметр обычно также равен 0.

2) printf("Created address region: %p\n", &addr\_region[0]);

printf выводит адрес начала выделенной области памяти.

*x. измените права доступа к созданному региону и проверьте какая будет реакция, если их нарушить:*

*2. запретите писать и попробуйте записать.*

*xi. попробуйте перехватить сигнал SIGSEGV.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#include <signal.h>

#define SUCCESS 0

#define ERROR (-1)

#define PAGE\_SIZE 4096

#define NO\_FD (-1)

void SegfaultHandler(int signum) {

printf("Caught signal %d: Segmentation fault\n", signum);

exit(EXIT\_FAILURE);

}

void AllocateAddressRegion() {

int ret = SUCCESS;

char \*address\_region;

address\_region = mmap(NULL, 10 \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

if (address\_region == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

ret = mprotect(address\_region, 10 \* PAGE\_SIZE, PROT\_READ); // права только на чтение

if (ret == ERROR) {

perror("mprotect");

exit(EXIT\_FAILURE);

}

printf("Changed memory protection: PROT\_READ\n");

\*address\_region = 'a'; // пытаемся записать

sleep(10);

}

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

signal(SIGSEGV, SegfaultHandler);

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

sleep(10);

AllocateAddressRegion();

return EXIT\_SUCCESS;

}

Объяснение кода:

1.

void SegfaultHandler(int signum) {

printf("Caught signal %d: Segmentation fault\n", signum);

exit(EXIT\_FAILURE);

}Это обработчик сигнала. Её будет вызывать операционная система, если в процессе произойдёт ошибка сегментации (то есть SIGSEGV).

Параметр signum — это номер сигнала:

Например:

SIGSEGV → это 11 на большинстве Unix-систем.

Если ты попытаешься прочитать или записать в память, к которой у тебя нет доступа — система отправит сигнал SIGSEGV.

Этот обработчик печатает сообщение, завершает программу через exit(EXIT\_FAILURE) (это означает "завершить с ошибкой").

2.

signal(SIGSEGV, SegfaultHandler);sighandler\_t signal(int signum, sighandler\_t handler);

signum: номер сигнала, например SIGSEGV, SIGINT, SIGTERM и т.д.

handler: указатель на функцию-обработчик сигнала.

Ты ОС говоришь: "Если произойдёт сигнал SIGSEGV, не убивай процесс, а вызови мою функцию SegfaultHandler(int)."

*x. измените права доступа к созданному региону и проверьте какая будет реакция, если их нарушить:*

*1. запретите читать данные и попробуйте прочитать из региона.*

*xi. попробуйте перехватить сигнал SIGSEGV.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#include <signal.h>

#define SUCCESS 0

#define ERROR (-1)

#define PAGE\_SIZE 4096

#define NO\_FD (-1)

void SegfaultHandler(int signum) {

printf("Caught signal %d: Segmentation fault\n", signum);

exit(EXIT\_FAILURE);

}

void AllocateAddressRegion() {

int ret = SUCCESS;

char \*address\_region;

address\_region = mmap(NULL, 10 \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

if (address\_region == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

ret = mprotect(address\_region, 10 \* PAGE\_SIZE, PROT\_WRITE); // права только на запись

if (ret == ERROR) {

perror("mprotect");

exit(EXIT\_FAILURE);

}

printf("Changed memory protection: PROT\_WRITE\n");

printf("%c\n", \*address\_region); // пытаемся читать

sleep(10);

}

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

signal(SIGSEGV, SegfaultHandler);

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

sleep(10);

AllocateAddressRegion();

return EXIT\_SUCCESS;

}

**Что такое сигнал SIGSEGV?**

SIGSEGV — это сигнал, который посылается операционной системой процессу, когда он пытается обратиться к памяти, к которой у него нет доступа. Название расшифровывается как "Segmentation Violation" — нарушение сегмента памяти.

**Когда возникает SIGSEGV?**

Этот сигнал возникает в ситуациях, когда:

- процесс чита́ет или пи́шет по "непринадлежащему" адресу;

- обращается к "нулевому указателю" (например, \*NULL);

- нарушает права доступа к защищённой области памяти: например, если доступ разрешён только на чтение (PROT\_READ), а вы попробуете записать;

- или если вы удалили область (munmap), но продолжили к ней обращаться.

**Что происходит при нарушении прав доступа?**

Процесс делает системный вызов, чтобы записать в память.

ОС проверяет таблицу страниц — есть ли у этого процесса право записи по этому адресу.

Если права отсутствуют — ОС посылает сигнал SIGSEGV.

Если этот сигнал не перехвачен (обработчик не установлен), процесс немедленно завершается с ошибкой.

*xii. отсоедините страницы с 4 по 6 в созданном вами регионе.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#define SUCCESS 0

#define ERROR (-1)

#define PAGE\_SIZE 4096

#define NO\_FD (-1)

void AllocateAddressRegion() {

int ret = SUCCESS;

// char \*address\_region;

// address\_region = mmap(NULL, 10 \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

// if (address\_region == MAP\_FAILED) {

// perror("mmap");

// exit(EXIT\_FAILURE);

// }

// ret = munmap(address\_region + 3 \* PAGE\_SIZE, 3 \* PAGE\_SIZE);

// if (ret == ERROR) {

// perror("munmap");

// exit(EXIT\_FAILURE);

// }

char \*region[10];

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

region[i] = mmap(NULL, PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, NO\_FD, 0);

if (region[i] == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

}

for (int i = 4; i < 7; ++i) {

if (munmap(region[i], PAGE\_SIZE) == -1) {

perror("munmap");

exit(EXIT\_FAILURE);

}

}

printf("Deallocated address region\n");

sleep(10);

}

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

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

sleep(10);

AllocateAddressRegion();

return EXIT\_SUCCESS;

}

Объяснение кода:

ret = munmap(address\_region + 3 \* PAGE\_SIZE, 3 \* PAGE\_SIZE);

address\_region + 3 \* PAGE\_SIZE — это 4-я страница (индексация с нуля).

3 \* PAGE\_SIZE — от этой точки освобождаем 3 страницы: 4-6-ю.

munmap() — удаляет указанный участок из адресного пространства процесса.

В munmap() второй аргумент — это размер удаляемого блока, а не конечный адрес!

*xiii. понаблюдайте за адресным пространством.*

**КАК ЗАПУСКАТЬ:**

1) Открываем первое окно терминала, переходим в папку, создаем файл

2) Компилируем программу

3) Открываем второе окно терминала, переходим в ту же папку

4) В первом окне терминала запускаем программу, дожидаемся пока нам выведет PID

5) Во втором окне терминала ПИШЕМ (но пока НЕ отправляем!!!)

vmmap <pid>где вместо <pid> указываем PID, который нам вывело ранее

6) Когда в первом окне терминала появилась строчка

Deallocated address regionво втором окне нажимаем ENTER (отправляем наш vmmap)