**Завдання 1**

**Текст завдання**

Скільки пам’яті може виділити malloc(3) за один виклик? Параметр malloc(3) є цілим числом типу даних size\_t, тому логічно максимальне число, яке можна передати як параметр malloc(3), — це максимальне значення size\_t на платформі (sizeof(size\_t)). У 64-бітній Linux size\_t становить 8 байтів, тобто 8 \* 8 = 64 біти. Відповідно, максимальний обсяг пам’яті, який може бути виділений за один виклик malloc(3), дорівнює 2^64. Спробуйте запустити код на x86\_64 та x86. Чому теоретично максимальний обсяг складає 8 ексабайт, а не 16?

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

int main(){

    size\_t max\_val = (size\_t) - 1;

    printf("max val of size\_t -> %zu \n", max\_val);

    void \*ptr = malloc(max\_val);

    if(ptr == NULL){

        printf("memory allocation failed\n");

        return 1;

    }

    else{

        printf("memory allocated without any trouble;P\n");

        free(ptr);

    }

    return 0;

}



size\_t на 64-бітній системі — 8 байтів (тобто 64 біти), отже, максимально можливе значення — 2^64 - 1 (≈ 18.4 ексабайт).

Але архітектура x86\_64 обмежена користувацьким простором адресації до 48 біт. Отже, реально виділити можна до 2^48 (≈ 256 ТБ). Саме тому не 16 ексабайт.

## Завдання 2

### Текст завдання

Що станеться, якщо передати malloc(3) від’ємний аргумент? Напишіть тестовий випадок, який обчислює кількість виділених байтів за формулою num = xa \* xb. Що буде, якщо num оголошене як цілочисельна змінна зі знаком, а результат множення призведе до переповнення? Як себе поведе malloc(3)? Запустіть програму на x86\_64 і x86.

#include <stdio.h>

#include <stdlib.h>

int main() {

    int xa = 100000;

    int xb = 100000;

    int num = xa \* xb;

    printf("Result of multiplication: %d\n", num);

    size\_t safe\_num = (size\_t)num;

    void \*ptr = malloc(safe\_num);

    if (ptr == NULL) {

        printf("Memory allocation failed due to overflow or insufficient memory.\n");

    } else {

        printf("Memory allocated successfully!\n");

        free(ptr);

    }

    return 0;

}



Якщо результат переповнення — від’ємне число, воно буде неявно перетворене до size\_t, тобто до дуже великого позитивного значення

## Завдання 3

### Текст завдання

Що станеться, якщо використати malloc(0)? Напишіть тестовий випадок, у якому malloc(3) повертає NULL або вказівник, що не є NULL, і який можна передати у free(). Відкомпілюйте та запустіть через ltrace. Поясніть поведінку програми.

#include <stdio.h>

#include <stdlib.h>

int main() {

    void \*ptr = malloc(0);

    if (ptr == NULL) {

        printf("malloc(0) returned NULL\n");

    } else {

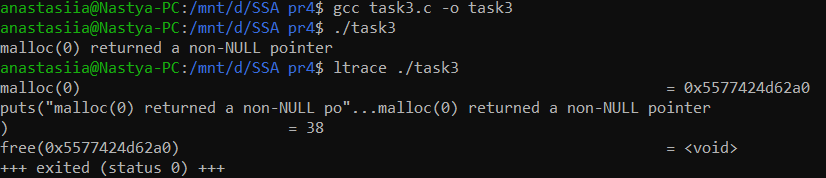
        printf("malloc(0) returned a non-NULL pointer\n");

        free(ptr);

    }

    return 0;

}



malloc(0) може повернути не-NULL, тобто валідний вказівник — і це абсолютно нормальна поведінка згідно зі стандартом C.

## Завдання 4

### Текст завдання

Чи є помилки у такому коді? void \*ptr = NULL; while () { if (!ptr) ptr = malloc(n); [... <використання 'ptr'> ...] free(ptr); }

#include <stdio.h>

#include <stdlib.h>

int main(){

    void \*ptr = NULL;

    while(1){

    if (!ptr){

        free(ptr);

        ptr = malloc(sizeof(void \*));

        printf("ptr locates at %p\n", ptr);

    }

    }

    free(ptr);

    return 0;

}



ptr виділяється тільки один раз, але звільняється кожну ітерацію. На наступному циклі — ptr уже не валідний, але не оновлюється (бо if (!ptr) вже не істинне)

## Завдання 5

### Текст завдання

Що станеться, якщо realloc(3) не зможе виділити пам’ять? Напишіть тестовий випадок, що демонструє цей сценарій.

#include <stdio.h>

#include <stdlib.h>

int main() {

    size\_t size = 10;

    void \*ptr = malloc(size);

    if (!ptr) {

        printf("step 1 -> initial allocation failed;(\n");

        return 1;

    }

    size\_t new\_size = size \* ((size\_t) - 1);

    void \*new\_ptr = realloc(ptr, new\_size);

    if (new\_ptr == NULL) {

        printf("step 2 -> realloc failed;(\n");

        free(ptr);

    } else {

        printf("realloc succeeded!\n");

        free(new\_ptr);

    }

    return 0;

}



Перший етап виділення памʼяті пройшов успішно, що є очевидним, адже 10 - "адекватний" розмір для подібної операції. Під час "перевиділення" памʼяті ми задали надзвичайно великий параметр, що спричинило порушення роботи програми, про що нам каже стрічка "step 2 -> realloc failed;"

## Завдання 6

### Текст завдання

Якщо realloc(3) викликати з NULL або розміром 0, що станеться? Напишіть тестовий випадок.

#include <stdio.h>

#include <stdlib.h>

int main() {

    void \*ptr1 = realloc(NULL, 100);

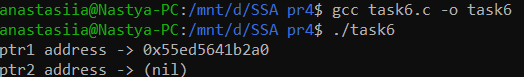
    void \*ptr2 = realloc(ptr1, 0);

    printf("ptr1 address -> %p\n", ptr1);

    printf("ptr2 address -> %p\n", ptr2);

    return 0;

}



По факту, realloc(NULL, 100) спрацювало як malloc, бо ми "на рівному місці" виділили памʼять.А realloc(ptr1, 0) - виділення 0 комірок. А як ми памʼятаємо із завдання 3, таке можливо.

## Завдання 7

### Текст завдання

Перепишіть наступний код, використовуючи reallocarray(3): struct sbar \*ptr, newptr; ptr = calloc(1000, sizeof(struct sbar)); newptr = realloc(ptr, 500sizeof(struct sbar));

#include <stdlib.h>

#include <stdio.h>

#include <malloc.h>

struct sbar {

    int a;

    char b;

};

int main() {

    struct sbar \*ptr = calloc(1000, sizeof(struct sbar));

    struct sbar \*newptr = reallocarray(ptr, 500, sizeof(struct sbar));

    if (!newptr) {

        printf("reallocarray\n");

        free(ptr);

        return 1;

    }

    printf("new pointer address -> %p \n", newptr);

    free(newptr);

    return 0;

}



Логіка полягає у тому, що кщо reallocarray не змогла виділити нову пам’ять, виводиться помилка, і стару пам’ять звільняють.У разі успіху стара пам’ять копіюється в нову область, і стара звільняється автоматично.

Судячи з повідомлення про адресу вказівника, програма була виконана успішно, тому що reallocarray() вважається безпечнішою за realloc() — особливо в контексті захисту від переповнення цілого числа при множенні, бо reallocarray перевіряє переповнення перед виділенням памʼяті.

## Завдання 8

### Текст завдання

Напишіть кастомний memory allocator на базі freelist.

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#define BLOCK\_SIZE 64

#define POOL\_SIZE  1024

typedef struct FreeBlock {

    struct FreeBlock\* next;

} FreeBlock;

typedef struct {

    uint8\_t\* memory\_pool;

    FreeBlock\* free\_list;

    size\_t block\_size;

    size\_t total\_blocks;

} Allocator;

void allocator\_init(Allocator\* allocator, size\_t block\_size, size\_t pool\_size) {

    allocator->block\_size = block\_size;

    allocator->total\_blocks = pool\_size / block\_size;

    allocator->memory\_pool = malloc(pool\_size);

    if (!allocator->memory\_pool) {

        printf("malloc");

        exit(1);

    }

    allocator->free\_list = NULL;

    for (size\_t i = 0; i < allocator->total\_blocks; i++) {

        FreeBlock\* block = (FreeBlock\*)(allocator->memory\_pool + i \* block\_size);

        block->next = allocator->free\_list;

        allocator->free\_list = block;

    }

}

void\* allocate(Allocator\* allocator) {

    if (!allocator->free\_list) {

        return NULL;  // Пам’ять закінчилася

    }

    FreeBlock\* block = allocator->free\_list;

    allocator->free\_list = block->next;

    return (void\*)block;

}

void deallocate(Allocator\* allocator, void\* ptr) {

    if (!ptr) return;

    FreeBlock\* block = (FreeBlock\*)ptr;

    block->next = allocator->free\_list;

    allocator->free\_list = block;

}

void allocator\_destroy(Allocator\* allocator) {

    free(allocator->memory\_pool);

    allocator->memory\_pool = NULL;

    allocator->free\_list = NULL;

}

int main() {

    Allocator allocator;

    allocator\_init(&allocator, BLOCK\_SIZE, POOL\_SIZE);

    void\* ptr1 = allocate(&allocator);

    void\* ptr2 = allocate(&allocator);

    printf("Allocated block at %p\n", ptr1);

    printf("Allocated block at %p\n", ptr2);

    deallocate(&allocator, ptr1);

    deallocate(&allocator, ptr2);

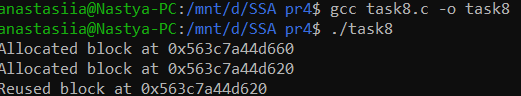
    void\* ptr3 = allocate(&allocator); // reuse!

    printf("Reused block at %p\n", ptr3);

    allocator\_destroy(&allocator);

    return 0;

}



У цьому завданні ми реалізували власний менеджер пам’яті — простий аллокатор на основі списку вільних блоків, або freelist. Ідея полягає в тому, щоб заздалегідь виділити великий шматок пам’яті, розбити його на менші блоки однакового розміру і керувати ними вручну. Для цього ми створюємо однозв’язний список вільних блоків: кожен вільний блок містить вказівник на наступний. Коли потрібно виділити пам’ять, ми просто забираємо блок з початку списку. Коли пам’ять більше не потрібна, ми повертаємо блок назад у список. Це дозволяє дуже швидко розподіляти і звільняти пам’ять без викликів malloc і free кожного разу.