Практична робота 4

Завдання ЗАГАЛЬНЕ ДЛЯ ВСІХ

Завдання 4.1

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?

Завдання 4.2

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.

Завдання 4.3

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.

Завдання 4.4

Чи є помилки у такому коді?

```
void *ptr = NULL;
while (<some-condition-is-true>) {
   if (!ptr)
      ptr = malloc(n);
   [... <використання 'ptr'> ...]
   free(ptr);
}
```

Напишіть тестовий випадок, який продемонструє проблему та правильний варіант коду.

Завдання 4.5

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

Завдання 4.6

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

Завдання 4.7

Перепишіть наступний код, використовуючи reallocarray(3):

```
struct sbar *ptr, *newptr;
ptr = calloc(1000, sizeof(struct sbar));
newptr = realloc(ptr, 500*sizeof(struct sbar));
```

Порівняйте результати виконання з використанням ltrace.

Завдання по ВАРІАНТАХ

1. Реалізуйте власну функцію malloc, використовуючи mmap() та brk().
2. Напишіть програму для імітації витоку пам’яті та засічіть її за допомогою valgrind.
3. Досліджуйте поведінку malloc у багатопоточному середовищі.
4. Напишіть тестовий випадок, де malloc повертає однакові адреси після послідовного виділення та звільнення.
5. Використайте mprotect для створення області пам’яті, що неможливо змінювати.
6. Дослідіть поведінку malloc під час високого навантаження (виділення великого обсягу пам’яті).
7. Напишіть програму, яка перевіряє ефективність cache locality для розподіленої пам’яті.
8. Використайте aligned_alloc для виділення пам’яті з певним вирівнюванням.
9. Напишіть кастомний memory allocator на базі freelist.
10. Дослідіть розподіл пам’яті в структурі heap Linux.
11. Створіть сценарій, що перевіряє фрагментацію heap при виділенні/звільненні пам’яті.
12. Напишіть тест на використання calloc у порівнянні з malloc + memset.
13. Використайте mallopt() для налаштування malloc та перевірте ефект.
14. Перепишіть malloc на основі jemalloc.
15. Дослідіть різницю між malloc у glibc та musl.
16. Реалізуйте memory pool для часто використовуваних об’єктів.
17. Напишіть тестовий випадок, що досліджує різницю між виділенням великої та малої кількості пам’яті.
18. Використайте mmap() для створення пам’яті, яку не звільняє ОС після завершення програми.
19. Напишіть код, що примусово викликає out-of-memory killer у Linux.
20. Використайте pthreads для конкурентного доступу до heap та перевірте ефективність.
21. Перевірте роботу malloc у середовищі з обмеженим обсягом пам’яті.
22. Використайте системні виклики sbrk() та brk() для маніпуляції heap.
23. Реалізуйте обгортку навколо malloc, яка веде журнал виділеної пам’яті.
24. Використайте sanitizers для перевірки витоків пам’яті.

25. Дослідіть, як працює lazy allocation у Linux та його вплив на malloc.