Про помилки роботи з пам’яттю в Linux та засоби їх усунення

1. Теорія

1.1. Типові проблеми, які виникають при роботі з пам’яттю в Linux

Ці проблеми є частими причинами багів у продакшн-програмах, особливо в C/C++, де керування пам’яттю виконується вручну. Навіть такі незначні помилки, як a[3] замість a[2], можуть:

- руйнувати структури даних
- викликати аварійне завершення (segfault)
- спричинити витоки та повільне падіння продуктивності.

Таблиця 1. Значення термів

№	Англійський термін	Український переклад	Значення
1	Uninitialized variables(UV)	Неініціалізовані змінні	Змінна використовується до надання значення
2	Out-of-bounds access	Вихід за межі	Звернення до пам’яті поза масивом
3	Memory leakage	Витік пам’яті	Виділена пам’ять не звільняється

4	Undefined behavior(UB)	Невизначена поведінка	Наслідки не визначені стандартом
5	Fragmentation	Фрагментація пам’яті	Пам’ять розбита, що ускладнює нові а кації

1.2. Приклади з ASCII-візуалізаціями

1. Неініціалізовані змінні

```
int main() {
    int x;
    printf("x = %d\n", x); // Значення x не визначене
    return 0;
}
```

//Це не UB, але може виводити випадкові значення, оскільки стек містить "сміття".

2. Вихід за межі масиву (Out-of-bounds)

```
int a[3] = {1, 2, 3};
a[5] = 42; // Доступ за межами масиву
```

Схематичне поясненн коду

```
a[0] a[1] a[2] ??  ??  a[5]
  1    2    3   ?   ?   ← 42 тут — заборонений доступ
```

3. Витік пам’яті (Memory Leak)

```
void leak() {
    int *p = malloc(100 * sizeof(int));
    // жодного free() → витік
}
```

//Після завершення функції вказівник зникає, а виділена пам’ять – залишається зайнятою.

4. Невизначена поведінка (Undefined Behavior)

```
int a = 10;
int b = a++ + ++a; // UB: порядок виконання не визначено
```

Вираз a++ + ++a може дати різний результат на різних компіляторах.

5. Фрагментація пам’яті (Fragmentation)

```
char *a = malloc(8);
char *b = malloc(128);
char *c = malloc(8);
free(b); // Звільняємо великий блок між двома маленькими


[ a:8B ][ b:128B ][ c:8B ]
    ↓ free()
[ a:8B ][  F R E E ][ c:8B ]


// Після цього виділити великий блок НЕМОЖЛИВО → фрагментація
```

6. "Завислий" покажчик (Dangling pointer)

```
int *p = malloc(sizeof(int));
*p = 42;
free(p);
printf("%d", *p); // доступ до вже звільненої пам’яті
```

//Може випадково вивести правильне значення, але це undefined behavior.

7. Подвійне звільнення (Double Free)

```
int *p = malloc(sizeof(int));
free(p);
free(p); // Помилка: подвійне звільнення
 :
```

//Може призвести до краху або компрометації безпеки (heap poisoning).

8. Переповнення буфера (Buffer Overflow)

```
char *p = malloc(10);
free(p);
```

```
p[0] = 'A'; // використання звільненої пам’яті
```

Перезапис пам’яті, що може змінити змінні або return-адресу в стеку.

9. Use-after-free

```
char *p = malloc(10);
free(p);
p[0] = 'A'; // використання звільненої пам’яті
```

10. Пошкодження пам’яті (Memory Corruption)

```
int a[3] = {1, 2, 3};
int b = 5;
a[5] = 99; // змінює пам’ять змінної b або інші структури
```

//Може потайки зіпсувати дані в іншій частині програми.

2.2. Valgrind-практикум

Як запускати Valgrind:

bash

```
valgrind --leak-check=full ./a.out
```

Приклади використання

1. Витік пам’яті (Memory Leak)

C

```
// leak.c
#include <stdlib.h>

int main() {
    int *ptr = malloc(100 * sizeof(int)); // виділення пам'яті
    ptr[0] = 42;                          // використання
    return 0; // Немає free(ptr)
}
```

bash

```
gcc -g leak.c -o leak
```

```
valgrind --leak-check=full ./leak
```

Valgrind:

```
python-repl
==12345== HEAP SUMMARY:
==12345==    definitely lost: 400 bytes in 1 blocks
...
==12345== LEAK SUMMARY:
==12345==    definitely lost: 400 bytes in 1 blocks

...
```

//Valgrind виявив, що 400 байтів було "definitely lost", тобто програма точно втратила посилання на виділений блок пам’яті.

2. Використання після звільнення (Use-after-free)

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

int main() {
    int *p = malloc(sizeof(int));
    *p = 10;
    free(p);         // звільнення
    *p = 20;         // використання після free
    return 0;
}
```

```
bash
gcc -g uaf.c -o uaf
valgrind ./uaf
```

Valgrind:

```
arduino
```

```
==12346== Invalid write of size 4
==12346==    at 0x4005F6: main (uaf.c:9)
==12346==  Address 0x5204040 is 0 bytes inside a block of size 4 free'd
==12346==    by 0x4005ED: main (uaf.c:8)
```

//Valgrind повідомляє про "Invalid write" — запис у пам’ять, яка вже була звільнена (free).

3. Подвійне звільнення (Double Free)

```
// double_free.c
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof(int));
    free(p);
    free(p); // друга спроба — помилка
    return 0;
}
```

```
gcc -g double_free.c -o double_free
valgrind ./double_free
```

Valgrind:

```
==12347== Invalid free()
==12347==    at 0x... : free ...
==12347==  Address 0x5204040 was freed already
```

//Valgrind точно вказує, що адресу вже було звільнено, і друга спроба free() є недійсною.

Додаткові параметри Valgrind:

Параметр	Опис
--leak-check=full	Повний звіт про витоки пам’яті

--show-leak-kinds=	Показати всі типи витоків (definitely, indirectly, etc.)
--track-origins=yes	Показує, звідки взялася "сміттєва" пам’ять (use of nitialized mem)
--log-file=valgrind.l	Зберегти результат у файл

2. Практика

2.1. Варіанти завдань

1. Напишіть програму, яка демонструє використання змінної до її ініціалізації, але так, щоб на більшості систем це не викликало миттєвий крах.

2. Реалізуйте фрагмент коду, який зчитує за межами виділеного масиву без явного порушення доступу — і поясніть, чому це можливо.

3. Змоделюйте ситуацію, коли одна й та сама змінна випадково використовується як покажчик, хоча ним не є — і що з цього випливає.

4. Створіть сценарій, в якому втрата пам’яті (leak) стається лише при рідкісному гілкуванні логіки і не виявляється стандартними профайлерами.

5. Реалізуйте фрагмент програми з подвійним звільненням пам’яті, який не завжди викликає помилку при виконанні.

6. Побудуйте приклад, де програма виділяє пам’ять у циклі, не вивільняючи її, але виконується годинами без збою — поясніть чому.

7. Створіть випадок, коли фрагментація heap-пам’яті стає причиною відмови виділення нового блоку, навіть якщо загалом пам’яті ще вистачає.

8. Напишіть програму, яка звертається до стеку поза межами дозволеної області, але система не видає Segmentation Fault.

9. Побудуйте тест, де дві змінні випадково використовують ту саму адресу пам’яті (через обхід типів) — і виявити це можливо лише за допомогою логування.

10. Реалізуйте програму, яка перезаписує область пам’яті між двома структурами — і цей баг не проявляється на всіх компіляторах.

11. Створіть ситуацію, в якій використання realloc() призводить до втрати даних, якщо неправильно обробити результат виклику.

12. Побудуйте код, який динамічно виділяє пам’ять, але втрати трапляються лише при викиданні виключення або сигнала.

13. Напишіть фрагмент, в якому valgrind не може однозначно визначити утечку пам’яті, хоча вона є.

14. Побудуйте приклад "викрадення" покажчика, коли одна функція звільняє пам’ять, яка ще використовується в іншій.

15. Реалізуйте динамічну структуру даних, де циклічні посилання заважають звільненню всієї пам’яті.

16. Створіть приклад з memcpy(), який призводить до пошкодження пам’яті, але лише при вирівнюванні певного типу структури.

17. Продемонструйте приклад undefined behavior, пов’язаний з порядком обчислення аргументів у виклику функції.

18. Напишіть код, в якому неправильне приведення типів призводить до втрати частини даних при копіюванні у динамічну пам’ять.

19. Реалізуйте сценарій, де пам’ять не звільняється лише в разі виклику програми певним користувачем або групою.

20. Побудуйте програму, яка виділяє пам’ять за допомогою mmap() і втрачає її після fork() — без ознак у valgrind.

21. Напишіть програму, в якій помилка читання одного байта за межами буфера призводить до пошкодження іншого модуля.

22. Реалізуйте демонстрацію порушення через "use-after-free", коли об’єкт передається у бібліотечну функцію після звільнення.

23. Побудуйте приклад, в якому частина структури перезаписується через змінний union — і ця помилка не є очевидною при огляді коду.

24. Напишіть утиліту, яка перевіряє ефект фрагментації у великій кількості дрібних алокацій і спроб нових виділень.

25. Змоделюйте поведінку програми, яка поступово втрачає доступ до пам’яті через невірне зберігання покажчиків у стеку.

26. Створіть код, який спотворює значення, що зберігаються в heap, без доступу напряму до цих змінних — поясніть механізм.

27. Проведіть експеримент з malloc/free на структурі, яка включає вкладений масив покажчиків — і перевірте, які області реально звільнені.