Київський національний університет імені Тараса Шевченка

**Лабораторна робота №4**

**З навчального курсу «Unix-подібні операційні системи»**

**«System Call API. Hooks & Event Handlers.** **»**

Виконав:

студент 3 курсу

факультету кібернетики

спеціальність «Комп’ютерні науки»

групи ТТП-32

Чебан Богдан Володимирович

**Київ 2024**

**Постановка задачі:**

Скористатись можливостями ядра системи (System Call API) для наступних задач:  
1) прочитати пам'ять іншого процесу (*наприклад*, в 1 процесі в змінну записуємо певне значення (наприклад, вводом з клавіатури), а 2й процес слідкує за цією ділянкою пам'ята 1го процесу та показує зміни, якщо вони відбуваються) або альтернативно – зробити сторінку **wired / non-paged**   
2) перехопити (**hook**) клавіатуру або мишку із **неактивного** застосунку (тобто, *наприклад*, виводити у вікні те, що набирають на клавіатурі у іншому application, "перехопити клавіатурне введення") + продумати сценарій демонстрації.  
  
Hints:   
\* див. /proc/[id]/mem - доступ до пам'яті  
\* /dev/input... - доступ до I/O-пристроїв

**Хід роботи:**

1) прочитати пам'ять іншого процесу (*наприклад*, в 1 процесі в змінну записуємо певне значення (наприклад, вводом з клавіатури), а 2й процес слідкує за цією ділянкою пам'ята 1го процесу та показує зміни, якщо вони відбуваються) або альтернативно – зробити сторінку **wired / non-paged**   
  
**Крок 1: Підготовка**

1. **Створення TargetProcess**

Написання програми **TargetProcess.cpp**, яка резервує динамічно виділену пам'ять під рядок даних та виводить PID процесу і адресу цієї динамічно виділеної пам'яті.

Компіляція **TargetProcess.cpp** з використанням **g++** для створення виконуваного файлу.

1. **Створення ObserverProcess**

Написання програми **ObserverProcess.cpp**, яка читає специфічну область пам'яті іншого процесу за допомогою системних викликів.

Компіляція **ObserverProcess.cpp** з використанням **g++** для створення виконуваного файлу.

**Крок 2: Запуск TargetProcess**

Запуск **TargetProcess**, який виведе PID і адресу пам'яті, де зберігається рядок.

Запис PID і адреси пам'яті для подальшого використання.

**Крок 3: Використання ObserverProcess**

Запуск **ObserverProcess** з аргументами PID і адресою пам'яті, отриманими від **TargetProcess**.

Спостереження за виводом **ObserverProcess**, щоб перевірити, чи правильно відображається вміст пам'яті цільового процесу.

**Крок 4: Перевірка результатів**

Використання **ObserverProcess** для читання даних з пам'яті процесу **TargetProcess** і перевірка, чи виведені дані співпадають з тим, що було записано в **TargetProcess**.

**Крок 5: Висновки**

Аналіз отриманих даних та визначення, чи програма **ObserverProcess** коректно читає пам'ять іншого процесу.

Розгляд можливих проблем з безпекою та стабільністю при доступі до пам'яті інших процесів.

Підготовка висновків про можливості та обмеження такого методу читання пам'яті.

TargetProcess.cpp:

#include <unistd.h>

#include <iostream>

#include <new>

int main() {

const char\* data = new (std::nothrow) char[28]{"I love Polishuk and EPAM"};

if (!data) {

std::cerr << "Memory allocation failed." << std::endl;

return 1;

}

std::cout << "pid: " << getpid() << "\nData address: " << static\_cast<const void\*>(data)

<< "\nData size: " << sizeof("I love Polishuk and EPAM\n") << std::endl;

std::cin.get();

delete[] data;

return 0;

}

ObserverProcess.cpp:

include <iostream>

#include <iomanip>

#include <fcntl.h>

#include <unistd.h>

#include <cerrno>

#include <cstring>

#include <vector>

// Usage: ./process\_reader pid address size

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

if (argc != 4) {

std::cerr << "Usage: " << argv[0] << " <pid> <address> <size>" << std::endl;

return -1;

}

std::string pid{argv[1]};

auto start = std::stoull(argv[2], nullptr, 16);

auto size = std::stoull(argv[3]);

auto filename = "/proc/" + pid + "/mem";

std::cout << "Filename: " << filename << "\nAddr is: " << (void\*)start << "\nSize: " << size << std::endl;

int fd = open(filename.c\_str(), O\_RDONLY);

if (fd == -1) {

std::cerr << "Error opening file: " << strerror(errno) << std::endl;

return -2;

}

off\_t res = lseek(fd, (off\_t)start, SEEK\_SET);

if (res == (off\_t)-1) {

std::cerr << "Error seeking in file: " << strerror(errno) << std::endl;

close(fd);

return -3;

}

std::vector<char> buffer(size);

ssize\_t read\_res = read(fd, buffer.data(), buffer.size());

if (read\_res == -1) {

std::cerr << "Error reading file: " << strerror(errno) << std::endl;

close(fd);

return -4;

}

std::string output(buffer.begin(), buffer.begin() + read\_res);

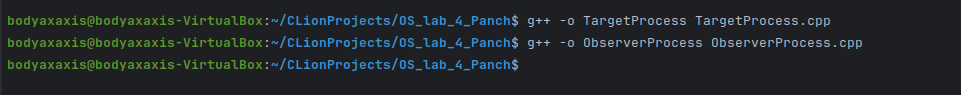
std::cout << "Data is: " << std::quoted(output) << "\n";

close(fd);

return 0;

}

Виконання коду:

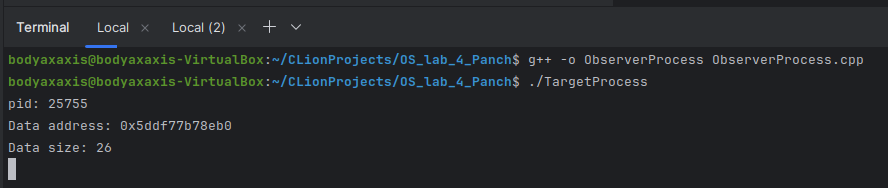


g++ -o TargetProcess TargetProcess.cpp

g++ -o ObserverProcess ObserverProcess.cpp

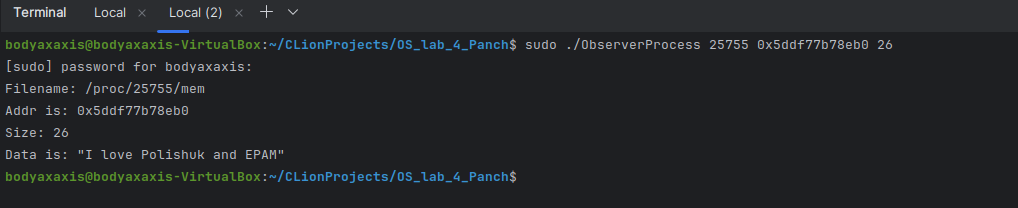
….

./TargetProcess



…

./ObserverProcess <PID> <address>



Опис коду:

**TargetProcess.cpp**

Ця програма призначена для виведення інформації про змінну, що зберігає певну строку, яка розташована у динамічній пам'яті.

**Залучення необхідних заголовочних файлів**:

**<unistd.h>**: для функції **getpid()**, що отримує PID (идентифікатор процесу).

**<iostream>**: для введення-виведення.

**<new>**: для оператора **new** без виключень (nothrow).

**Головна функція (main)**:

Створюється змінна **data** типу **const char\***, яка використовує оператор **new** для резервування пам'яті під рядок "I love Polishuk and EPAM".

Перевіряється, чи пам'ять була успішно резервована. Якщо ні, виводиться повідомлення про помилку і програма завершується з кодом помилки **1**.

Виводиться PID поточного процесу, адреса динамічно виділеної пам'яті та розмір данних.

Програма очікує на введення з клавіатури, щоб утримувати консоль відкритою.

Використовується **delete[]** для звільнення резервованої пам'яті перед завершенням програми.

**ObserverProcess.cpp**

Ця програма призначена для читання пам'яті іншого процесу на основі заданих аргументів: PID, адреси пам'яті та розміру даних.

**Перевірка аргументів командного рядка**:

Якщо кількість аргументів не дорівнює чотирьом, програма виводить інструкції з використання та завершується з кодом помилки **-1**.

**Відкриття файлу пам'яті процесу**:

Створюється шлях до файлу пам'яті цільового процесу, використовуючи **/proc/[PID]/mem**.

Відкривається файловий дескриптор до цього файлу.

**Читання пам'яті**:

Здійснюється перехід до вказаної адреси пам'яті з використанням **lseek**.

Читається задана кількість байтів у вектор **buffer**.

Якщо виникає помилка, виводиться повідомлення про помилку і програма завершується з відповідним кодом помилки.

**Виведення прочитаних даних**:

З прочитаних байтів створюється рядок **output**.

Виводиться рядок, оточений лапками для відображення у вигляді цитати.

Після завершення, файловий дескриптор закривається, і програма завершується нормально.

2) перехопити (**hook**) клавіатуру або мишку із **неактивного** застосунку (тобто, *наприклад*, виводити у вікні те, що набирають на клавіатурі у іншому application, "перехопити клавіатурне введення") + продумати сценарій демонстрації.

ХІД РОБОТИ

Я написав програму на C++, яка перехоплює події клавіатури в Linux за допомогою інтерфейсу **/dev/input**. Моя мета була відстежити натискання клавіш в іншому додатку та вивести цю інформацію.

Спочатку я підготував середовище розробки, забезпечивши доступ до файлової системи **/dev/input** та переконавшись, що маю необхідні дозволи для читання з пристрою клавіатури.

Я відкрив термінал і виконав наступні команди для компіляції та запуску моєї програми:

shCopy code

sudo /home/bodyaxaxis/CLionProjects/OS\_lab\_4\_Panch/cmake-build-debug/OS\_lab\_4\_Panch

Я запустив програму з правами суперкористувача (sudo), щоб забезпечити необхідний доступ до пристрою вводу. Програма **OS\_lab\_4\_Panch** була створена для читання подій клавіатури із файлу пристрою клавіатури Linux.

У програмі я визначив масив рядків **evval**, що представляє стан клавіш (натиснута, відпущена, повторна). Я використав системний виклик **open** для отримання дескриптора файлу, що відповідає пристрою клавіатури.

Далі я розпочав безкінечний цикл для читання подій клавіатури з пристрою вводу. Для кожного прочитаного івенту я перевіряв, чи він є подією клавіатури (**EV\_KEY**), та виводив статус клавіші та її код. Якщо читання завершувалося помилкою, я виводив повідомлення про помилку і завершував цикл.

Після перехоплення подій клавіатури я міг аналізувати отримані дані і виводити інформацію на екран. Це дозволило мені спостерігати за вводом клавіатури в реальному часі.

На завершення, я закрив дескриптор файлу і закінчив роботу програми. Ця робота дала мені більш глибоке розуміння взаємодії з пристроями Linux на низькому рівні і можливостей перехоплення та обробки системних подій.

…

Команди в термінал:

sudo /home/bodyaxaxis/CLionProjects/OS\_lab\_4\_Panch/cmake-build-debug/OS\_lab\_4\_Panch

…

#include <iostream>

#include <fcntl.h>

#include <unistd.h>

#include <linux/input.h>

#include <cstring>

#include <cerrno>

#include <array>

int main() {

const char\* dev = "/dev/input/by-path/platform-i8042-serio-0-event-kbd";

std::array<const char\*, 3> evval = {

"RELEASED",

"PRESSED ",

"REPEATED"

};

int fd = open(dev, O\_RDONLY);

if (fd == -1) {

std::cerr << "Cannot open " << dev << ": " << strerror(errno) << ".\n";

return EXIT\_FAILURE;

}

input\_event ev;

while (true) {

ssize\_t n = read(fd, &ev, sizeof(ev));

if (n == (ssize\_t)-1) {

if (errno == EINTR) {

continue; // Interrupted system call

} else {

std::cerr << "Read error: " << strerror(errno) << ".\n";

break;

}

} else if (n != sizeof(ev)) {

errno = EIO; // Input/output error

std::cerr << "Mismatched read size: " << strerror(errno) << ".\n";

break;

}

if (ev.type == EV\_KEY && ev.value >= 0 && ev.value <= 2) {

std::cout << evval[ev.value] << " 0x" << std::hex << ev.code

<< " (" << std::dec << ev.code << ")\n";

}

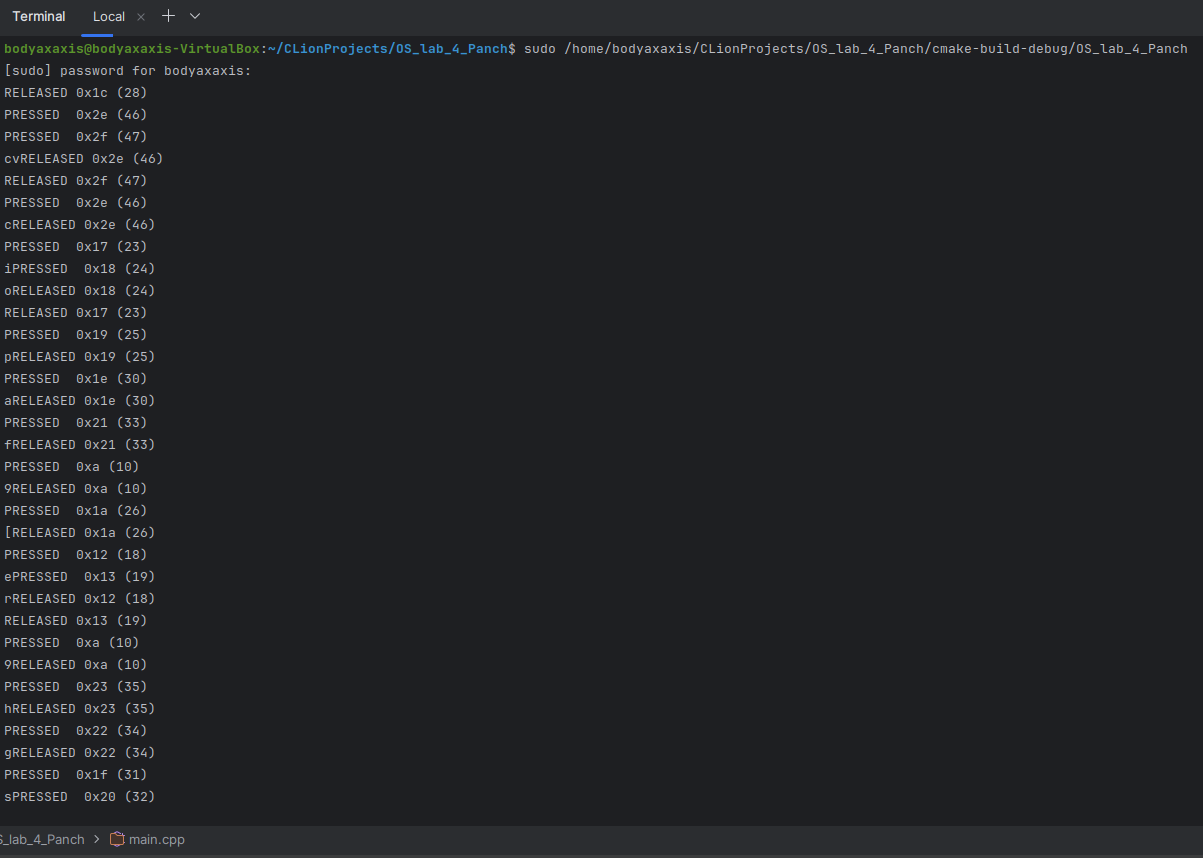
}

close(fd);

return EXIT\_FAILURE;

}

Вивід(буде нижче):



Опис коду:

1. **Підключення бібліотек**:

**<iostream>**: для роботи з потоками вводу/виводу.

**<fcntl.h>**: для використання функції **open**.

**<unistd.h>**: для системних викликів **read** і **close**.

**<linux/input.h>**: для використання структури **input\_event**, яка описує події вводу.

**<cstring>** і **<cerrno>**: для роботи з строками та помилками.

**<array>**: для використання контейнера масиву.

1. **Шлях до пристрою**:

Ініціалізується змінна **dev** із шляхом до файлу подій клавіатури. У Linux шлях **/dev/input/by-path/** містить символічні посилання на файли пристроїв вводу.

1. **Масив станів клавіш**:

Створюється масив **evval**, який містить три строки, що представляють можливі стани клавіш: "RELEASED" (відпущена), "PRESSED" (натиснута) і "REPEATED" (затиснута).

1. **Відкриття файлу пристрою**:

За допомогою системного виклику **open** відкривається файловий дескриптор **fd** для файла пристрою клавіатури у режимі тільки для читання. Якщо файл не вдається відкрити, виводиться повідомлення про помилку і програма завершується із статусом **EXIT\_FAILURE**.

1. **Основний цикл**:

У безкінечному циклі відбувається читання подій клавіатури.

Функція **read** намагається прочитати структуру **input\_event** з файлового дескриптора **fd**.

Якщо читання переривається сигналом (**errno** встановлено в **EINTR**), читання відновлюється.

Якщо **read** повертає **-1**, це означає помилку, і програма виводить повідомлення про помилку та виходить з циклу.

Якщо кількість прочитаних байтів не відповідає розміру **input\_event**, це свідчить про помилку, і програма завершується з кодом помилки **EIO**.

1. **Обробка подій**:

При успішному читанні події, якщо тип події є **EV\_KEY** (подія клавіатури), перевіряється значення **ev.value**, яке вказує на тип події: 0 для відпущеної, 1 для натиснутої, 2 для утримуваної клавіші.

Якщо виявлена подія клавіатури, програма виводить тип події з масиву **evval** та шістнадцяткове та десяткове значення коду клавіші **ev.code**.

1. **Закриття дескриптора**:

Після завершення роботи програми, файловий дескриптор закривається за допомогою функції **close(fd)**.

1. **Статус завершення**:

Якщо програма виходить з циклу читання через помилку, вона повертає **EXIT\_FAILURE**.

**Висновок:**

Я навчився користуватись системними викликами Linux для взаємодії з процесами та пристроями вводу на низькому рівні. Виконання цієї лабораторної роботи дало мені практичні навички та глибше розуміння роботи операційної системи з пам'яттю та пристроями вводу.

У першій частині я зміг успішно прочитати пам'ять іншого процесу, використовуючи адресу пам'яті та PID. Це досвід підкреслив важливість розуміння внутрішньої структури процесів в ОС та показав, як захист пам'яті може впливати на взаємодію між процесами.

У другій частині я зосередився на перехопленні клавіатурних подій. Цей досвід ознайомив мене з тим, як працювати з файлами пристроїв **/dev/input**, і я зрозумів, що перехоплення вводу з клавіатури може бути корисним для створення програм, які реагують на користувацькі дії навіть тоді, коли вони не активні.

Робота над цією лабораторною роботою відкрила для мене можливості безпосередньої взаємодії з ядром Linux через системні виклики. Я також вивчив розміщення і застосування системних бібліотек та інструментів для налагодження, що є критично важливими для розробки низькорівневого програмного забезпечення.

Завдяки цій роботі я отримав цінний досвід, який збільшив мої технічні знання і зрозуміння системного програмування. Також це підсилило мою обережність і відповідальність при роботі з можливостями, які можуть впливати на приватність користувачів та безпеку системи.