Зображення, що містить будівля, ескіз, малюнок, замок

Автоматично згенерований опис  
Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Кібербезпека та захист інформації:

Архітектура комп’ютерних систем

Студент: Бичок Вадим Вячеславович

Робота: 5

Назва: Системні виклики в архітектурі х86 на асемблері для операційної системи Linux.

Використані програми: NASM, LD

Суть завдання полягає у створенні, відкритті та маніпуляції з файлом згідно варіанту(10, 15, 40, 25, 5, 6):

1. Створити масив відповідно до вашого варіанту.  
   Зображення, що містить текст, Шрифт, знімок екрана, Графіка

   Автоматично згенерований опис
2. Вивести на екран “My array sum is :” (не використовуючи функцію printf) з результатом операції відповідно до вашого варіанту (Множення кожного елемента на 2 та сума елементів).   
   Зображення, що містить текст, Шрифт, знімок екрана

   Автоматично згенерований опис
3. Створити файл lab5\_<номер вашого варіанту> з правами доступу відповідно до вашого варіанту у результаті.  
   Зображення, що містить знімок екрана, текст, Мультимедійне програмне забезпечення, програмне забезпечення

   Автоматично згенерований опис
4. Виконати системне переривання відповідного до вашого варіанту(Дозапис результата операції у файл).  
   Зображення, що містить текст, Мультимедійне програмне забезпечення, програмне забезпечення, Шрифт

   Автоматично згенерований опис

Детальніше по коду:

global \_start

Ініціалізуємо список, змінні-лічильники, назву файлу, рядка що потрібно вивести, і їхні довжини.  
section .data

a dq 10, 15, 40, 25, 5, 6

lan equ $ - a

lena equ ($ - a)/8

filename db "lab5\_1", 0

lenf equ $ - filename

mya db "My array sum is: ", 0

lenm equ $ - mya

sum dq 0

lens dq 0

section .text

\_start:

Ініціалізуємо лічильник циклу  
 mov rax, 0

mulsum:

Порівнюємо лічильник і дожину циклу, якщо рівні то виходимо з циклу.  
 cmp rax, lena

je writemyar

Множемо лічильник на 8 і записуємо в rcx для зсуву на наступний елемент.  
 imul rcx, rax, 8

Беремо наступний елемент списку.  
 mov rbx, [a+rcx]

Множимо елемент на 2  
 imul rbx, 2

Заносимо назад за адресою значення числа помножене на 2.  
 mov [a+rcx], rbx

Додаємо до суми елемент списку помножений на 2.  
 add [sum], rbx

Збільшуємо лічильник на 1 і переходимо на початок циклу.  
 inc rax

jmp mulsum

Мітка запису суми в командний рядок.  
writemyar:

Заносимо в rax 1, це номер системного виклику sys\_write.  
 mov rax, 1

Записуємо в дескриптор 1, це вивід в командний рядок.  
 mov rdi, 1

Заносимо в rsi адресу рядка "My array sum is: ", для її виведення.  
 mov rsi, mya

Заносимо в rdx довжину рядка, який ми хочемо вивести.  
 mov rdx, lenm

Викликаємо системну функцію, яка лежить регістрі rax.  
 syscall

Записуємо в rax значення, чию довжину хочемо найти.  
 mov rax, [sum]

Ініціалізуємо лічильник який і буде нашою довжиною.  
 mov r9, 0

Заносимо в rcx 10, щоб рахувати кількість цифр в числі.  
 mov rcx, 10

Цикл знаходження довжини числа, яке потрібно перевести в рядок.  
lensum:

Порівнюємо регістр rax і 0, так як після ділення в rax записується результат.  
 cmp rax, 0

Виходимо з циклу, якщо результат ділення 0, тобто число закінчилося.  
 je l

Додаємо по 1 кожного разу, якщо число ще не 0.  
 inc r9

Анульовуємо rdx, в який записується остача, щоб не виникало проблем.  
 xor rdx, rdx

Ділимо значення в регістрі rax на rcx, частка запишеться rax, о остача в rdx, регістр rcx не змінюється, що ми і використаємо пізніше.  
 div rcx

jmp lensum

Мітка l створена для виходу з попереднього циклу.  
l:

Записуємо результат (r9) в змінну lens.  
 mov [lens], r9

Заносимо в регістр rdx адресу числа для перевірки.  
 mov rdx, sum

Заносимо в регістр rdi вказівник на кінець буфера, тобто на першу цифру в числі.  
 lea rdi, [sum+r9]

Ініціалізуємо rbx як 0, тобто лічильник довжини рядка.  
 mov rbx, 0

Заносимо в rax значення яке потрібно перевести в рядок.  
 mov rax, [sum]

.loop:

Очищуємо rdx для запису остачі з ділення.  
 mov rdx, 0

Ділимо rax на rcx(10) і отримуємо результат(частку) в rax, і остачу в rdx(яка нам і потрібна для запису поточного числа як символа).  
 div rcx

До остачі додаємо ascii символ “0” і отримуємо ascii символ нашої цифри.  
 add rdx, '0'

Показник на символ числа, яке записуємо, зменшуємо для того щоб в наступному проході циклу записати вже наступну цифру у рядок.  
 dec rdi

Після ділення div залишок автоматично розміщується в rdx, але лише молодший байт (dl) містить значення останньої цифри (від 0 до 9). Це тому, що при діленні на 10 залишок завжди менше 10 і може бути записаний у 8-бітному регістрі. rdx — 64-бітний регістр. Щоб додати до нього ASCII-значення '0', потрібно б було спочатку подбати про те, щоб верхні біти rdx були обнулені. Використання лише dl дозволяє уникнути цих додаткових кроків, бо ми можемо бути впевнені, що значення залишку повністю розміщується у восьми молодших бітах.  
Тому записуємо цифру в буфер саме так.  
 mov [rdi], dl

Збільшуємо довжину рядка.  
 inc rbx

Якщо rax == 0, тобто більше немає чого ділити, усі цифри оброблено і записано в рядок.  
 cmp rax, 0

Якщо rax != 0, повторюємо цикл.  
 jne .loop

Мітка де виводиться в командний рядок результат операції, тут все аналогічно до попереднього системного виклику тільки значення і його довжина різні.  
writeint:

mov rax, 1

mov rdi, 1

mov rsi, sum

mov rdx, [lens]

syscall

Мітка в якій створюється файл, надаються на нього права, і те що з ним ми будемо робити.  
crtf:

Номер 2 це номер системного виклику sys\_open яким і будемо створювати, або відкривати файл, якщо він існує, заносимо його в регістр rax.  
 mov rax, 2

Заносимо в rdi ім’я файлу, який будемо створювати, чи змінювати, це перший аргумент який передається системному виклику.  
 mov rdi, filename

rsi це другий аргумент який передається системному виклику, в нього записуємо що і як робити з файлом.  
O\_APPEND (1024) — Відкриття файлу в режимі дозапису.

O\_CREAT (64) — Створює файл, якщо він не існує.

O\_WRONLY (1) — Відкриває файл для запису.  
 mov rsi, 1+64+1024

rdx це третій аргумент, і в нього записуємо права, які надаватимемо на файл, якщо ми його створюємо.  
Префікс 0o (або іноді просто 0) вказує на **восьмеричну систему** числення, що дозволяє системі відрізнити ці значення від десяткових.  
 mov rdx, 0o777

syscall

Мітка в якій знову використовуємо sys\_write, тільки тепер дескриптор не 0, а значення, яке після створення файлу записується в регістр rax, ми його зберігаємо в регістрі r12, і використовуємо для запису того ж що і в минулих системних викликах sys\_write.

wrtf:

mov r12, rax

mov rax, 1

mov rdi, r12

mov rsi, mya

lea rdx, [lenm]

dec rdx

syscall

mov rax, 1

mov rdi, r12

mov rsi, sum

mov rdx, [lens]

syscall

Мітка в якій закриваємо файл, просто вносячи дескриптор файлу як перший аргумент.  
clsf:

mov rax, 3

mov rdi, filename

syscall

Завершуємо програму.  
endpr:

mov rax, 60

xor rdi, rdi

syscall

Контрольні запитання:

1. Що таке системний виклик у асемблері ?  
   Це виклик системної функції, яка приймає аргументи записані в регістри, за своєю послідовністю, у них свій calling convention який відрізняється від Windows.
2. В чому різниця організації системного виклику для Linux та Windows в коді асемблера?  
   Різниця полягає в тому що, по перше, сам виклик виглядає по різному, тобто в Windows ми викликаємо за допомогою call (funct) якийсь конкретний системний виклик(функцію), а у Linux спочатку заносимо в регістр rax номер системного виклику, а потім, після задання всіх потрібних аргументів, викликаємо syscall.

1. В чому особливість додавання з нового рядка(у рядок) у файл у мові асемблер в системі Linux?  
   В Unix-подібних системах, таких як Linux, новий рядок представлений символом \n (ASCII 10 або 0x0A). У текстовому файлі для переходу на новий рядок необхідно додати цей символ вручну.  
     
   При виклику sys\_open в регістр rsi потрібно дописати O\_APPEND (1024) — Відкриття файлу в режимі дозапису, що і буде заставляти саме додавати записи у кінець файлу, що відкрили, або створили.
2. В чому особливість запису у файл в Linux?  
   У Linux запис у файл здійснюється через системні виклики:

sys\_open (номер виклику 2) відкриває файл і повертає дескриптор файлу, який використовується для інших операцій.

Потрібно при відкритті, або створенні файлу вказати як саме будемо записувати.  
rsi це другий аргумент який передається системному виклику, в нього записуємо що і як робити з файлом.

O\_APPEND (1024) — Відкриття файлу в режимі дозапису.  
O\_TRUNC (512) — скоротити файл до нуля, якщо він уже існує.

O\_CREAT (64) — Створює файл, якщо він не існує.

O\_WRONLY (1) — Відкриває файл для запису.  
  
sys\_write (номер виклику 1) записує дані в файл, використовуючи дескриптор файлу, адресу даних та їхню довжину.

sys\_close (номер виклику 3) закриває файл, звільняючи ресурси.

### **1. Системні виклики sys\_open, sys\_write, і sys\_close**

У Linux запис у файл здійснюється через системні виклики:

* **sys\_open** (номер виклику 2) відкриває файл і повертає дескриптор файлу, який використовується для інших операцій.
* **sys\_write** (номер виклику 1) записує дані в файл, використовуючи дескриптор файлу, адресу даних та їхню довжину.
* **sys\_close** (номер виклику 3) закриває файл, звільняючи ресурси.

**2. Використання дескрипторів файлів**

Після відкриття файлу системний виклик sys\_open повертає дескриптор файлу — ціле число, яке є унікальним ідентифікатором відкритого файлу для поточного процесу. Цей дескриптор необхідно використовувати в інших системних викликах, як-от sys\_write та sys\_close, для ідентифікації файлу.

**3. Права доступу до файлів**

При відкритті або створенні файлу задаються права доступу, як-от:

* **O\_APPEND (1024)** — Відкриття файлу в режимі дозапису.
* **O\_TRUNC (512)** — скоротити файл до нуля, якщо він уже існує.
* **O\_CREAT (64)** — Створює файл, якщо він не існує.
* **O\_WRONLY (1)** — Відкриває файл для запису.

**5. Передача адреси та довжини даних**

У системному виклику sys\_write потрібно явно передавати:

* Адресу даних у пам'яті (rsi).
* Кількість байтів для запису (rdx).

Linux не має автоматичного розпізнавання кінця рядка або буферизації, тому програмісту потрібно контролювати точну кількість байтів.

### В чому особливість створення анонімного файлу в Linux? **Ключові особливості анонімного файлу**

1. **Відсутність імені у файловій системі**:
   * Анонімний файл не має імені та не з'являється в файловій системі. Доступ до нього можливий тільки через файловий дескриптор, отриманий під час його створення.
2. **Створення за допомогою memfd\_create**:
   * В Linux є системний виклик memfd\_create, який створює анонімний файл, що існує тільки в пам'яті. Цей файл зникає після закриття дескриптора або завершення програми.
   * Синтаксис виклику: int memfd\_create(const char \*name, unsigned int flags);
   * Параметр name — це умовне ім'я для ідентифікації файлу в пам'яті, але воно не прив'язується до файлової системи.
   * Прапори, як-от MFD\_CLOEXEC, визначають поведінку файлу при успадкуванні процесами.
3. **Тимчасовий файл з O\_TMPFILE**:
   * Файли без імені можна створити за допомогою прапора O\_TMPFILE в sys\_open. Це дозволяє створити файл, який не має імені в файловій системі, і відразу доступний лише через дескриптор.
   * Приклад: open("/tmp", O\_TMPFILE | O\_RDWR, 0600); створить тимчасовий файл без імені, доступний тільки для поточного процесу.
4. **Видалення після закриття**:
   * Після закриття дескриптора, анонімний файл видаляється з пам'яті. Це зручно для тимчасових даних, які не потрібно зберігати постійно.
5. В чому особливість перейменування файлу в Linux?  
   Команда mv у Linux використовує системний виклик rename для перейменування, або переміщення файлів або каталогів, але лише тоді коли операція відбувається в межах одного і того ж файлового розділу(жорсткого диску), тобто без зміни фізичного місця файлу.   
   В цьому випадку операція дуже швидка, оскільки лише оновлюється запис у каталозі, що містить посилання на inode (унікальний ідентифікатор файлу в файловій системі).   
   Якщо виконується переміщення з одного розділу в інший то rename не використовується.

### Яка структура використання регістрів при системному виклику в Linux? **Архітектура x86\_64**

У 64-бітній архітектурі **x86\_64**, процес виклику системної функції (системний виклик) виглядає наступним чином:

1. **Реєстрація системного виклику**: Для 64-бітних систем використовується **номер системного виклику** (системний виклик має свій унікальний ідентифікатор), який передається через **регістр** rax. Цей номер визначає, яку саме операцію необхідно виконати в ядрі.
2. **Передача параметрів**: Параметри для системного виклику передаються через спеціальні регістри, в залежності від того, скільки параметрів потрібно для виклику:
   * **rax** — номер системного виклику.
   * **rdi** — перший параметр.
   * **rsi** — другий параметр.
   * **rdx** — третій параметр.
   * **r10** — четвертий параметр.
   * **r8** — п’ятий параметр.
   * **r9** — шостий параметр.

Якщо системний виклик вимагає більше ніж шість параметрів, то додаткові параметри передаються через стек.

1. **Виконання системного виклику**: Для виконання системного виклику в Linux на архітектурі x86\_64 використовується інструкція **syscall**. Це призводить до переривання, в результаті якого програма переходить у режим ядра, і ядро обробляє запит.
2. **Результат системного виклику**: Після виконання системного виклику результат (як правило, код повернення) повертається в **rax**. Якщо системний виклик успішний, у цьому регістрі буде міститися результат операції (наприклад, ідентифікатор файлу або кількість успішно виконаних операцій), а якщо виникне помилка, то в rax буде міститися код помилки (як правило, значення в rax буде від'ємним).

### Архітектура x86 (32-біт) та системні виклики

У 32-бітних системах (x86) для виконання системних викликів використовується інструкція int 0x80, яка викликає переривання і дозволяє здійснювати перехід від користувацького простору до режиму ядра.

#### Структура виклику системного виклику в **elf32**:

1. **Номер системного виклику**: Номер системного виклику (ідентифікатор функції, яку потрібно викликати) передається в регістр **eax**. Кожен системний виклик має свій унікальний номер (наприклад, для read номер системного виклику — це 3, для write — 4 тощо).
2. **Параметри системного виклику**: Параметри передаються через інші регістри:
   * **ebx** — перший параметр.
   * **ecx** — другий параметр.
   * **edx** — третій параметр.
   * **esi** — четвертий параметр (для деяких системних викликів).
   * **edi** — п’ятий параметр (для деяких системних викликів).
   * **ebp** — шостий параметр (для деяких системних викликів).

Параметри передаються у регістри відповідно до кількості аргументів, необхідних для системного виклику.

1. **Виконання системного виклику**: Виконання системного виклику здійснюється за допомогою інструкції **int 0x80**, яка є перериванням для виклику функцій ядра. Це призводить до того, що програма переходить до режиму ядра і виконується обробка запиту.
2. **Результат системного виклику**: Після виконання системного виклику результат або код помилки повертається в регістр **eax**. Якщо виклик був успішним, **eax** містить результат (наприклад, кількість прочитаних байт або дескриптор файлу). Якщо сталося помилку, **eax** містить негативний код помилки.
3. Що таке права доступу до файлу і яка їх структура?  
     
   У Linux, **права доступу до файлів** — це механізм, який визначає, хто і яким чином може взаємодіяти з файлами та директоріями. Цей механізм є основою системи безпеки і дозволяє адміністраторам і користувачам налаштовувати рівень доступу до ресурсів системи, щоб контролювати доступ до файлів і каталогів.

**Структура прав доступу до файлів**

Права доступу до файлів в Linux зазвичай описуються за допомогою трьох основних компонентів:

1. **Тип доступу**: Що можна зробити з файлом (читати, записувати або виконувати).
2. **Категорії користувачів**: Хто має ці права.
3. **Представлення прав доступу**: Це спосіб, яким права записуються або відображаються у файловій системі.

**1. Типи доступу**

У Linux існують три основних типи доступу:

* **r (read)** — право на читання файлу або вмісту каталогу.
* **w (write)** — право на запис у файл або зміну вмісту каталогу.
* **x (execute)** — право на виконання файлу (для програм) або право на доступ до каталогу (щоб в ньому переглядати чи змінювати файли).

**2. Категорії користувачів**

У Linux права доступу до файлів визначаються для трьох категорій користувачів:

* **Власник файлу (owner)**: Це користувач, який створив файл або той, хто має право змінювати його права доступу.
* **Група (group)**: Група користувачів, до якої належить файл. Якщо декілька користувачів належать до однієї групи, вони можуть мати спільні права доступу до файлів цієї групи.
* **Інші (others)**: Всі інші користувачі, які не є власниками файлу і не належать до його групи.

**3. Представлення прав доступу**

Права доступу до файлів в Linux зазвичай представляються у вигляді трьох символів для кожної категорії користувачів:

* Перші три символи визначають права власника файлу.
* Другі три — права групи.
* Треті три — права для всіх інших користувачів.

**Числове представлення прав доступу**

Права доступу також можна вказувати за допомогою числового формату, де кожен тип доступу має своє значення:

* **r (read)** = 4
* **w (write)** = 2
* **x (execute)** = 1

Числові значення для кожної категорії користувачів додаються, щоб отримати загальне значення для прав доступу. Наприклад:

* rwx = 4 + 2 + 1 = 7
* r-x = 4 + 0 + 1 = 5
* r-- = 4 + 0 + 0 = 4

Отже, права доступу для файлу, що мають значення rwxr-xr--, у числовому вигляді будуть представлені як **754**:

* Власник має права rwx (7).
* Група має права r-x (5).
* Інші користувачі мають права r-- (4).