

Лабораторна робота №9

Використання функцій API Win32 у програмах на асемблері

Мета: Навчитися використовувати у програмах на асемблері функції Windows динамічного виділення пам'яті та запису файлів.

Завдання:

1. Створити у середовищі MS Visual Studio проект з ім'ям **Lab9**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути головний файл **main9.asm** та модулі **module** (за необхідності) та модуль **longop** попередніх робіт.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуємий файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – файл числових значень згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати, вихідний текст та дизасембльований машинний код програми.

Теоретичні відомості

Для забезпечення можливостей створювати програми, які працюватимуть у середовищі Windows, програмістам наданий інтерфейс API (Application Program Interface) у вигляді прототипів системних функцій та інших бібліотечних файлів. При розробці власних програм програмісти можуть писати у вихідних текстах програмного коду виклики функцій API Win32. У такий спосіб можна створювати програми на різноманітних мовах програмування, у тому числі на асемблері.

Написання на асемблері програм з викликами системних функцій було відоме ще давно, наприклад, у MS-DOS – там виклики системних функцій були оформлені як переривання INTnn.

Виклик функцій API Win32 у програмах на асемблері зручно програмувати за допомогою директиви INVOKE. Формат виклику

invoke ім'яфункції, параметри

Програмування запису файлів

Для того, щоб записати щось у файл, потрібно відкрити цей файл, а якщо цього файлу ще немає – то створити. Так чи інакше, файл відкривається і з ним зв'язується деякий ідентифікатор, так званий хендл (handle). Використовуючи цей хендл, можна викликати функції для запису у файл порцій інформації. Для відкриття або для створення файлу можна скористатися функцією CreateFile. Ця функція відкриває старий або створює новий файл (відповідно до вказаних їй параметрів) і записує хендл файлу у регістр EAX. Якщо значення хендлу не дорівнює INVALID_HANDLE_VALUE, то програмі дозволений доступ до файлу. Щоб записати щось у файл, можна використати функцію WriteFile. Після роботи з файлом необхідно його закрити, викликавши функцію CloseHandle. Нижче наведений приклад запису у файл рядку тексту

```
.data
    hFile dd 0
    pRes dd 0
    szFileName db "tmp.txt",0
    szTextBuf db "Рядок тексту, записаний у файл",0

.code
    . . .
    invoke CreateFile, ADDR szFileName,
                                GENERIC_WRITE,
                                FILE_SHARE_WRITE,
                                0, CREATE_ALWAYS,
                                FILE_ATTRIBUTE_NORMAL,
                                0

    cmp eax, INVALID_HANDLE_VALUE
    je @exit                      ;доступ до файлу неможливий
    mov hFile, eax
    invoke strlen, ADDR szTextBuf
    invoke WriteFile, hFile, ADDR szTextBuf, eax, ADDR pRes, 0
    invoke CloseHandle, hFile

@exit:
    . . .
```

Величини GENERIC_WRITE, FILE_SHARE_WRITE, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL є символічними константами, оголошеними у файлі windows.inc. У разі помилки відкриття файлу (наприклад, через вказування неправильного імені файлу) функція CreateFile повертає через регістр EAX значення INVALID_HANDLE_VALUE. Функція WriteFile має такі параметри: хендл файлу, адреса блоку даних, далі кількість байтів даних, які потрібно записати, далі адреса перемінної, у яку буде записуватися інформація про коректність операції. Останній параметр у нашому випадку нульовий.

Для того, щоб записати у текстовий файл декілька рядків тексту, кожний рядок повинен завершуватися символами 13, 10. Ці коди можна дописувати окремим викликом функції WriteFile.

Вказування імені файлу

Для цього зручно скористатися стандартним діалоговим вікном Windows. Декілька часто уживаних діалогових вікон оформлені у бібліотеку діалогових вікон загального користування. Для доступу до функцій цієї бібліотеки потрібно підключити файли **comdlg32.inc, lib**.

Для появи стандартного діалогового вікна відкриття файлу для запису треба викликати функцію **GetSaveFileName**. Перед викликом цієї функції необхідно створити структуру типу OPENFILENAME, заповнити поля цієї структури потрібними значеннями, а потім викликати функцію GetSaveFileName, передавши у якості параметру адресу структури. Необхідно відзначити, що подібний спосіб виклику функцій використовується для багатьох функцій API Win32 – замість передачі великої кількості параметрів через стек створюється блок даних, структура, адреса якої потім вказується при виклику.

Оформимо вибір імені файлу у вигляді окремої процедури MySaveFileName. Оскільки структура типу OPENFILENAME містить тимчасову інформацію, то можна її запрограмувати, наприклад, як локальну структуру.

```
include \masm32\include\comdlg32.inc
includelib \masm32\lib\comdlg32.lib

.data
    szFileName db 256 dup(0)                ; буфер для імені файлу

.code
MySaveFileName proc
    LOCAL ofn : OPENFILENAME

    invoke RtlZeroMemory, ADDR ofn, SIZEOF ofn    ; спочатку усі поля обнулюємо
    mov ofn.lStructSize, SIZEOF ofn
    mov ofn.lpstrFile, OFFSET szFileName
    mov ofn.nMaxFile, SIZEOF szFileName
    invoke GetSaveFileName, ADDR ofn              ; виклик вікна File Save As
    ret
MySaveFileName endp

main:
    . . .
    call MySaveFileName
    cmp eax, 0                                ; перевірка: якщо у вікні було натиснуто кнопку Cancel, то EAX=0
    je @exit
```

Для ініціалізації деяких полів використана директива SIZEOF, яка обчислює на етапі компіляції розмір деякого об'єкту у байтах.

Запис масиву значень факторіалу. Перший варіант циклу

Нехай маємо процедуру обчислення факторіалу, яка зветься, наприклад, Factorial. Для запису масиву значень $n!$ (n від 1 до n_{max}) потрібно у циклі n_{max} разів викликати цю процедуру. На кожному кроці такого циклу результат – двійковий код підвищеної розрядності перетворюється у рядок десяткових цифр і записується у файл. Цей алгоритм можна відобразити так:

```
n = 1
while (n <= nmax)
{
  Factorial(n, fact)           //процедура обчислює n! Результат записує у fact
  StrDec(fact, textbuf)       //перетворення двійкового коду у десятковий
  WriteFile(textbuf)         //запис рядка десяткових цифр
  n = n+1
}
```

Оформлення обчислення факторіалу у вигляді окремої самодостатньої процедури сприяє структурованості програмного коду, проте має суттєвий недолік – відбувається виконання багатьох зайвих операцій множення підвищеної розрядності, оскільки кожний виклик процедури Factorial означає послідовне множення чисел від 1 до n .

Другий варіант циклу

```
fact = 1
n = 1
while (n <= nmax)
{
  fact = fact * n              //множення числа підвищеної розрядності (fact) на 32-бітове (n)
  StrDec(fact, textbuf)       //перетворення двійкового коду у десятковий рядок
  WriteFile(textbuf)         //запис десяткового рядка цифр
  n = n+1
}
```

Використання динамічної пам'яті

Для організації масивів даних особливо тих, які створюються тимчасово, зручно використовувати динамічну (глобальну) пам'ять. Об'єм динамічної пам'яті для кожної програми Win32 обмежується двома гігабайтами.

Для створення динамічного масиву можна використати функцію **GlobalAlloc**, яка належить до складу API Win32. Ця функція у залежності від параметрів виклику може повертати вказівник – адресу блоку пам'яті, що виділяється. Потім цей вказівник використовується для запису або читання потрібних даних. Коли динамічний масив стає непотрібним, його треба знищити за допомогою функції **GlobalFree**.

```
.data
    pD dd ? ; це буде 32-бітовий вказівник

.code
    . . .
    invoke GlobalAlloc, GPTR, 1024 ; створюємо динамічний масив з 1024 байт
    mov pD, eax ; беремо вказівник з EAX – результат GlobalAlloc
    . . . ; працюємо з динамічним масивом
    invoke GlobalFree, pD ; знищуємо динамічний масив - звільняємо пам'ять
```

Вказування параметру GPTR означає, що GlobalAlloc повертає адресу виділеного блоку пам'яті, а також цей блок від початку заповнюється нулями.

Розглянемо приклад обчислення факторіалу зі статичними даними

```
.data
    Result dd 32 dup(0) ; статичний масив Result 1024 байтів
    val dd 32 dup(0) ; статичний масив val
    n dd 1

.code
main:
    mov dword ptr[val], 1 ; val = 1
@cycle:
    inc dword ptr [n] ; n = n+1
    mov eax, dword ptr [n]
    cmp eax, 50 ; 50!
    jg @endfactorial

    push offset val
    push eax ; n
    push offset Result
    push 256
    call MulTo32_LONGOP ; Result = val * n

    push offset Result
    push offset val
    push 256
    call Copy_LONGOP ; val <- Result
    jmp @cycle
```

А тепер замість статичного 1024-байтового масиву val створимо тимчасовий динамічний масив, адресу якого буде зберігати вказівник pVal. Після обчислення факторіалу цей масив, який зберігає тимчасові значення підвищеної розрядності, стає непотрібним і він знищується.

```
.data
    Result dd 32 dup(0)          ; статичний масив
    pVal dd ?                    ; це буде вказівник на тимчасовий динамічний масив val
    n dd 1

.code
main:
    invoke GlobalAlloc, GPTR, 1024
    mov pVal, eax
    mov dword ptr[eax], 1        ; val = 1 (запис у масив по вказівнику робиться через регістр)
@cycle:
    inc dword ptr [n]            ; n = n+1
    mov eax, dword ptr [n]
    cmp eax, 50                  ; 50!
    jg @endfactorial

    push pVal
    push eax                     ; n
    push offset Result
    push 256
    call MulTo32_LONGOP          ; Result = val * n

    push offset Result
    push pVal
    push 256
    call Copy_LONGOP             ; val <- Result

    jmp @cycle

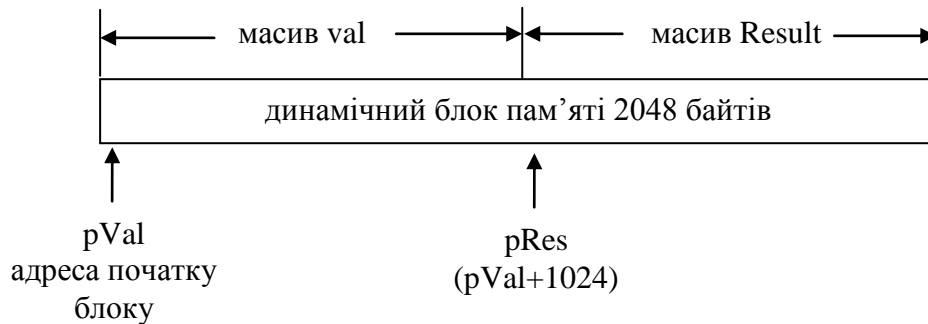
@endfactorial:
    invoke GlobalFree, pVal      ; знищуємо динамічний масив - звільняємо пам'ять
    . . .
```

Червоним виділено інструкції коду, які стосуються тимчасового динамічного масиву val.

Зверніть увагу на те, що вказівник pVal містить адресу потрібного об'єкту, тому процедурам MulTo32_LONGOP та Copy_LONGOP у відповідних параметрах передаються не "offset ім'я об'єкту", а значення вказівника.

Звісно, так само замість статичного масиву Result можна запрограмувати роботу ще з одним відповідним динамічним масивом. Здавалося, замість двох статичних масивів val та Result можна оголосити два вказівника, наприклад, pVal та pRes та двічі викликати функцію GlobalAlloc, і відповідно, для знищення двох масивів потрібно буде двічі викликати функцію GlobalFree. Проте, такий підхід не є досконалим. Замість багатьох маленьких масивів

краще виділити динамічний блок пам'яті, у якому будуть розташовані потрібні структури даних. Замість двох масивів по 1024 байтів створимо один блок пам'яті 2048 байтів, у якому виділимо адреси потрібних масивів. Адреси окремих масивів можна зберігати, наприклад, у окремих вказівниках.



Приклад реалізації

```
.data
    pRes dd ?           ; це буде вказівник на тимчасовий динамічний масив Result
    pVal dd ?           ; це буде вказівник на тимчасовий динамічний масив val
    n dd 1

.code
main:
    invoke GlobalAlloc, GPTR, 2048
    mov pVal, eax
    mov dword ptr[eax], 1    ; val = 1 (запис у масив по вказівнику робиться через регістр)
    add eax, 1024            ; адреса масиву Result
    mov pRes, eax

@cycle:
    inc dword ptr [n]        ; n = n+1
    mov eax, dword ptr [n]
    cmp eax, 50              ; 50!
    jg @endfactorial

    push pVal
    push eax                  ; n
    push pRes
    push 256
    call MulTo32_LONGOP      ; Result = val * n

    push pRes
    push pVal
    push 256
    call Copy_LONGOP         ; val <- Result

    jmp @cycle

@endfactorial:
    invoke GlobalFree, pVal   ; знищуємо динамічний блок пам'яті
    . . .
```

Варіанти завдання

Запрограмувати на асемблері запис у файл масиву значень факторіалу $n!$ (n від 1 до n_{max}). Вказування імені файлу у стандартному діалоговому вікні (функція GetSaveFileName). Для кожного студента своє значення n_{max}

$$n_{max} = 30 + 2 \times H,$$

де H – це номер студента у журналі.

№ вар	Варіант циклу факторіалу	Код результату	Масиви для даних підвищеної розрядності	Масив-буфер для імені файлу
1	1	десятковий	динамічні	динамічний
2	2	десятковий	динамічні	динамічний
3	1	шістнадцятковий	динамічні	динамічний
4	2	шістнадцятковий	динамічні	динамічний
5	1	десятковий	статичні	динамічний
6	2	десятковий	статичні	динамічний
7	1	шістнадцятковий	статичні	динамічний
8	2	шістнадцятковий	статичні	динамічний
9	1	десятковий	динамічні	статичний
10	2	десятковий	динамічні	статичний
11	1	шістнадцятковий	динамічні	статичний
12	2	шістнадцятковий	динамічні	статичний
13	1	десятковий	статичні	динамічний
14	2	десятковий	статичні	динамічний
15	1	шістнадцятковий	статичні	динамічний
16	2	шістнадцятковий	статичні	динамічний
17	1	десятковий	динамічні	динамічний
18	2	десятковий	динамічні	динамічний
19	1	шістнадцятковий	динамічні	динамічний
20	2	шістнадцятковий	динамічні	динамічний
21	1	десятковий	статичні	динамічний
22	2	десятковий	статичні	динамічний
23	1	шістнадцятковий	статичні	динамічний
24	2	шістнадцятковий	статичні	динамічний
25	1	десятковий	динамічні	статичний
26	2	десятковий	динамічні	статичний
27	1	шістнадцятковий	динамічні	статичний
28	2	шістнадцятковий	динамічні	статичний
29	1	десятковий	динамічні	статичний
30	2	десятковий	динамічні	статичний

Зміст звіту:

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасембльованого машинного коду
6. Висновки

Контрольні питання:

1. Що таке API Win32?
2. Як викликати системну функцію ОС Windows?
3. Що таке хендл?
4. Як відкрити файл для запису даних?
5. Яка функція записує у файл?
6. Як створити динамічні масиви даних?