УДК 681.3.07

О.Л. Левошко, ст. викл., Л.В. Константинова, ас.
Кіровоградський національний технічний університет

Мова програмування асемблер в задачах системного програмування операційної системи Windows

В статті описано особливості можливостей мови асемблера по оптимізації деяких задач системного програмування.
файлові операції, операції керування пам'яттю, відображення файлу в пам'яті, вмонтований асемблер

Системне програмування досить складна область, яка вимагає від розробника глибоких знань функціонування операційної системи. В більшості випадків до програм висуваються доволі жорсткі вимоги до об'єму пам'яті, яка використовується і до швидкодії самих програм [2]. Все це потребує високого ступеню оптимізації цих програм, як за швидкістю так і керуванням пам'яттю. Покращення показників працюючих програм можливе лише у тому випадку, якщо чітко уявляти можливості операційної системи по оптимізації програмного забезпечення.

Метою розробки є визначення позитивних моментів оптимізації задач системного програмування в ОС Windows. Оптимізація в застосуванні до процесу розробки і налагодження програм означає покращення характеристик роботи програмного продукту. Це є комплексом заходів по підвищенню продуктивності програм.

В умовах жорсткої конкуренції питання продуктивності є важливою умовою успіху чи невдачі програми на ринку програмних продуктів. Без серйозної роботи над покращенням продуктивності програмного коду не можна забезпечити конкурентоздатність додатку [1].

Процес оптимізації програмного забезпечення ускладнюється тим, що важко піддається алгоритмізації. Не існує єдиного критерію оптимізації.

У системному програмуванні вирішуються задачі з керування файловою системою, пам'яттю і процесами, міжпроцесорними комунікаціями, мережними з'єднаннями та інші з використанням інтерфейсу прикладного програмного забезпечення, а саме за допомогою функцій WIN API 32-розрядних операційних систем Windows [3]. Розглянемо деякі аспекти оптимізації подібних задач за допомогою мови програмування низького рівня асемблера.

Файлові операції та операції керування пам'яттю значною мірою впливають на продуктивність багатьох програмних продуктів, тому доцільно працювати над підвищенням ефективності їх виконання.

Файлові операції копіювання, переміщення, пошуку та видалення файлів та каталогів можна виконувати як з використанням бібліотечних функцій C++.NET, так і за допомогою функцій інтерфейса WIN API операційної системи [4]. Продуктивність виконання переважно залежить від алгоритма реалізації файлових операцій. Наприклад, на продуктивність операції копіювання файлів суттєвий вплив має розмір буфера пам'яті для зчитування/запису, спосіб організації даних, що зберігаються в пам'яті, кількість байтів, що пересилаються або копіюються.

Дуже часто під час копіювання одного файла в інший потрібне перетворення даних. Програмна реалізація такого перетворення суттєво впливає на продуктивність програми. Вмонтований асемблер дозволяє виконати ефективне перетворення даних з мінімальною втратою продуктивності. Крім того, асемблер дозволяє написати специфічні алгоритми обробки та перетворення, які, наприклад, в C++ реалізувати складно, використовуючи тільки бібліотечні функції.

Засоби розробки на асемблері дозволяють створювати швидкі утіліти командного рядку (консольні додатки). Використання в таких утилітах системних викликів Windows дозволяє виконати багато складних функцій (копіювання файлів, функції пошуку й сортування, обробка та аналіз математичних виразів і т.і.) з дуже високою швидкодією.

Покажемо на простих прикладах деякі методи оптимізації задач системного програмування.

Розглянемо метод оптимізації додатків для реалізації можливостей зручного маніпулювання даними. В наведеному фрагменті показано, як можна оптимізувати за швидкістю обробку великих масивів даних.

Нехай у вихідному текстовому файлі необхідно замінити символи прогалини символами плюс та зберегти копію файла під іншим ім'ям. Припустимо, що в якості файла джерела використовується текстовий файл з ім'ям readfile, а в якості файла приймача – writefile. Заміну символів виконаємо за допомогою блоку асемблерних команд. Вихідний текст консольного додатку наведемо у наступному лістингу.

```
#include "stdafx.h"
#include <stdio.h>
int main(int argc, _tchar* argv[])
{       FILE *fin, *fout;
        char buf[256];
        int bRead, bWritten;
        if ( (fin=fopen( "d:\\readfile", "r"))==NULL )
        {       printf( "The file 'readfile' was not opened\n" );
                exit(1);
        }
        if ( (fout=fopen( "d:\\writefile","w+"))==NULL )
        {       printf( "The file 'writefile' was not opened\n" );
                exit(1); }
        while ((bRead=fread(buf, sizeof(char), sizeof(buf), fin))>0)
        {
                _asm {
                        mov     ECX, bRead
                        lea     ESI, buf
                        mov     AL, ' '
                        next_ch:
                        cmp     BYTE PTR [ESI], AL
                        je      repl
                        inc     ESI
                        dec     ECX
                        jnz     next_ch
                        jmp     ex
                        repl:
                        mov     [ESI], '+'
                        inc     ESI
                        dec     ECX
                        jnz     next_ch
                        ex:
                        };
bWritten=fwrite(buf, sizeof(char), bRead, fout);
};
fclose(fin);
fclose(fout);
return 0;
}
```

Асемблерний блок виконує перетворення наступним чином: в регістр ESI завантажується адреса буферу пам'яті, де знаходяться зчитані дані. Регістр ECX вміщує

кількість байтів, які необхідно обробити. Символ, який замінюємо, тобто проміжок, знаходиться у регістрі AL. У кожній ітерації виконується порівняння символів у пам'яті і в регістрі AL. Якщо символи однакові, то на місце проміжку в буфер пам'яті записується символ плюс, і виконується перехід до наступної ітерації. Адреса елемента в буфері пам'яті інкрементується, а лічильник символів декрементується:

```
cmp     BYTE PTR [ESI], AL
je      repl
inc     ESI
dec     ECX
jnz     next_ch
```

У випадку нерівності символів виконується перехід до наступної ітерації одночасно з інкрементом адреси у регістрі ESI.

Вміст буферу пам'яті після перетворення зберігається у новому файлі з дескриптором fout.

```
bWritten= fwrite(buf, sizeof(char), bRead, fout)
```

За допомогою асемблера можливо створювати досить складні алгоритми обробки даних з файлів.

Жоден з додатків не обходиться без маніпуляцій з пам'яттю. Мова C++ має функцію malloc і оператор new для роботи з пам'яттю. У більшості випадків програмістам достатньо цих засобів. Але деякі задачі потребують більш гнучкого контролю над використанням пам'яті. У цьому випадку дуже зручною виявляється функція прикладного інтерфейсу WIN API VirtualAlloc. Ця функція дуже широко застосовується і порівняно з бібліотечною функцією malloc має цілий ряд переваг.

На відміну від malloc, функція VirtualAlloc дозволяє виділити частину пам'яті, яка вирівняна за межею сторінки і якій можна привласнити атрибути доступу (тільки читання, читання/запис, дозвіл виконання програмного коду і т.і.). Це дозволяє додатку виконувати обробку даних з максимальною швидкістю. Крім цього, функція VirtualAlloc може резервувати пам'ять без її фізичного виділення, що знижує навантаження на операційну систему в цілому.

Інший приклад пов'язаний з використанням функції розподілу пам'яті VirtualAlloc в операціях копіювання. Копіювання виконується у блоці асемблерних команд, причому використовуються команди рядкових примітивів з префіксом повторення rep. Це дозволяє виконати операцію максимально швидко. Вихідний текст програми, в якій використовуються переваги як функції VirtualAlloc, так і асемблерних команд рядкових примітивів, наведено в наступному лістингу.

```
#include "stdafx.h"
#include <windows.h>
#include <time.h>
int main(int argc, _tchar* argv[])
{
        int* src=NULL;
        int* dst=NULL;
        printf(" VirtualAlloc copying with ASM EXAMPLE\n\n");
        srand((unsigned)time(NULL));
        src=(int*)VirtualAlloc(NULL, 10, MEM_COMMIT, PAGE_READWRITE);
        int* bsrc=src;
        dst=(int*)VirtualAlloc(NULL, 10, MEM_COMMIT, PAGE_READWRITE);
        printf("\nsrc:");
        for(int cnt=0; cnt<10; cnt++)
        {
                *src=rand();
                printf("%d", *src);
                src++;
        }
```

//Копіювання виконується з великою швидкістю наступними асемблерними командами

```
        _asm {
                mov ESI, bsrc
                mov EDI, dst
                mov ECX, 10
                cld
                rep movsd
                }
        printf("\n\ndst :");
        for (int cnt=0; cnt<10; cnt++)
        {printf("%d", *dst);
        dst++;
        }
        VirtualFree(bsrc, 0, MEM_RELEASE);
        VirtualFree(dst, 0, MEM_RELEASE);
        getchar();
        return 0;
}
```

Операційні системи Windows підтримують ще одну досить корисну технологію роботи з файлами. Для операцій використовуються файли, які відображаються в пам'яті. Ця технологія дуже зручна для одночасної обробки файлів декількома процесами. Менеджер віртуальної пам'яті операційної системи дозволяє програмі працювати з файлом так, наче він завантажений в оперативну пам'ять комп'ютера. Для роботи з файлом, відображеному в пам'яті, необхідно виконати наступні кроки:

1. Відкрити файл за допомогою виклику CreateFile.
2. Передати дескриптор файлу функції WIN API CreateFileMopping.
3. Отримати вказівник на буфер пам'яті, де знаходиться файл, за допомогою функції MapViewOfFile.
4. По завершенні роботи з файлом необхідно викликати функцію UnmapViewOfFile.
5. Видалити дескриптор об'єкта відображення файлу та закрити дескриптор файлу за допомогою функції CloseHandle.

Застосування відображення файлу в пам'ять забезпечує високу продуктивність, якщо скористатись вмонтованим асемблером.

Отже, розглянуто одночасне використання вмонтованого асемблера і мови високого рівня (C++.NET) для написання класичних Windows-додатків процедурно-орієнтованого типу. Розробка таких програм зручна завдяки швидкості налагодження. Оскільки процедура розробляється в тілі основної програми, то не потрібно спеціальних засобів для її компоновки з головною програмою. Також не потрібно піклуватись про порядок передачі параметрів та про відновлення стеку. До недоліків цього методу оптимізації можна віднести деякі обмеження, котрі накладає середовище програмування на роботу асемблерних модулів, а також те, що процедури, розроблені на вмонтованому асемблері, не можна перетворити на зовнішні окремі модулі.

На основі всього вищевикладеного можна зробити висновок, що мова асемблера найкраще підходить для оптимізації деяких частин великого програмного проекту, який написано на мові високого рівня. Застосування асемблера – це один з найбільш дійових методів оптимізації програм і традиційні його переваги - компактність та швидкість виконання програмного коду можуть бути корисними при розв'язанні задач системного програмування.

В перспективі подальших розробок планується розглянути використання асемблера в задачах об'єктно-орієнтованого програмування.

Список літератури

1. Юрий Магда Ассемблер. Разработка и оптимизации Windows-приложений.- СПб : БХВ-Петербург, 2003.
2. Кип Р. Ирвин Язык ассемблера для процессоров Intel. 4-е издание.: Пер. С англ. – М.: Издательский дом «Вильямс», 2005.
3. Вильямс А. Системное программирование в Windows 2000 для профессионалов.-СПб.:Питер, 2001.
4. Юрий Магда Использование ассемблера для оптимизации программ на C++.- СПб : БХВ-Петербург, 2004.

В статье описаны особенности возможностей языка ассемблера по оптимизации некоторых задач системного программирования.

Peculiarities of the assembler language features concerning optimizations of some tasks in system programming are described in the article.