Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе**

**Дисциплина**: Проектирование ОС и компонентов

**Тема**: Анализ и профилирование прикладных приложений в ОС Windows и Linux

Выполнил студент гр. 13541/4 Степанов Д.С.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт - Петербург

2017

**Выполнение работы**

1. **Windows**

Всем процессам в операционной системе Windows предоставляется важнейший ресурс – виртуальная память (virtual memory). Все данные, с которыми процессы непосредственно работают, хранятся именно в виртуальной памяти.

Название "виртуальная" произошло из-за того что процессу неизвестно реальное (физическое) расположение памяти – она может находиться как в оперативной памяти (ОЗУ), так и на диске. Операционная система предоставляет процессу виртуальное адресное пространство (ВАП, virtual address space) определенного размера и процесс может работать с ячейками памяти по любым виртуальным адресам этого пространства, не "задумываясь" о том, где реально хранятся данные.

Размер виртуальной памяти теоретически ограничивается разрядностью операционной системы. На практике в конкретной реализации операционной системы устанавливаются ограничения ниже теоретического предела. Например, в 64 разрядных системах (x64) теоретический предел равен 16 экзабайт (264 байт = 16 777 216 ТБ = 16 ЭБ). При этом процессам выделяется 8 ТБ, ещё столько же отдается системе, остальное адресное пространство в нынешних версиях Windows не используется.

Введение виртуальной памяти, во-первых, позволяет прикладным программистам не заниматься сложными вопросами реального размещения данных в памяти, во-вторых, дает возможность операционной системе запускать несколько процессов одновременно, поскольку вместо дорогого ограниченного ресурса – оперативной памяти, используется дешевая и большая по емкости внешняя память. Схема реализации виртуальной памяти в 32-разрядной операционной системе Windows представлена на рисунке 1.

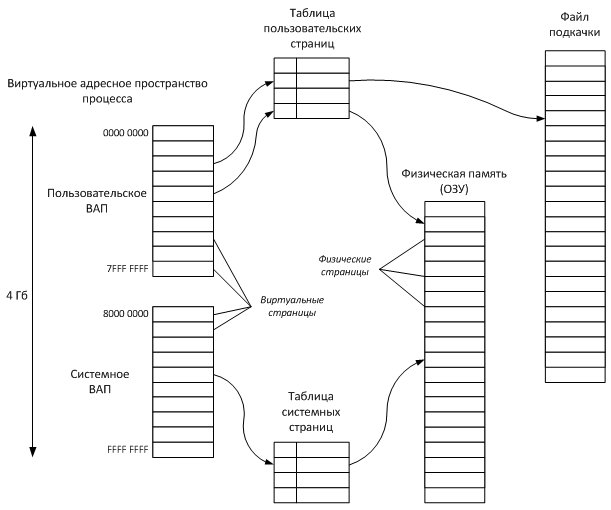


Рис. 1. Реализация виртуальной памяти в 32-разрядных Windows

Существует несколько способов выделения виртуальной памяти процессам при помощи Windows API. Рассмотрим два основных способа – с помощью функции VirtualAlloc и с использованием кучи.

1. WinAPI функция VirtualAlloc позволяет резервировать и передавать виртуальную память процессу. При резервировании запрошенный диапазон виртуального адресного пространства закрепляется за процессом (при условии наличия достаточного количества свободных страниц в пользовательском ВАП), соответствующие виртуальные страницы становятся зарезервированными (reserved), но доступа к этой памяти у процесса нет – при попытке чтения или записи возникнет исключение. Чтобы получить доступ, процесс должен передать память зарезервированным страницам, которые в этом случае становятся переданными (commit).

2. Для более гибкого распределения памяти существует куча процесса, которая управляется диспетчером кучи (heap manager). Кучу используют WinAPI функция HeapAlloc, а также оператор языка C malloc и оператор C++ new. Диспетчер кучи предоставляет возможность процессу выделять память с гранулярностью 8 байтов (в 32-разрядных системах), а для обслуживания этих запросов использует те же функции ядра, что и VirtualAlloc.

С помощью встроенных средств отладки Visual Studio 2013 рассмотрим состояние памяти программы. Во вкладке «дизассемблированный код» можно посмотреть представление кода в ассемблерном виде, а также адреса памяти, в которых хранятся инструкции (рисунок 2)

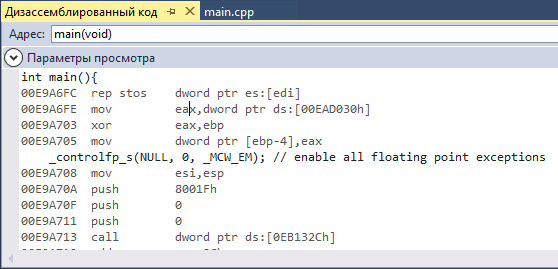


Рис.2. Вкладка «дизассемблированный код»

Узнав адрес начальной инструкции функции main, во вкладке «память» можно посмотреть состояние памяти в точке останова (рисунок 3).

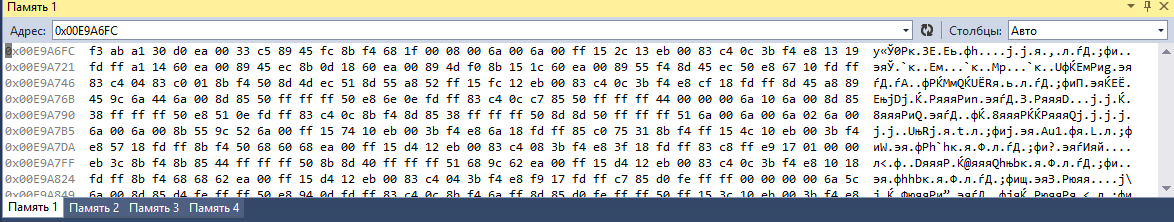


Рис.3. Окно просмотра состояния памяти

С помощью вкладки «контрольные значения» можно посмотреть текущие значения и адреса хранения всех переменных, с которыми работает программа. На рисунке 4 приведен пример для переменной wtext, которая хранит путь к анализируемой программе:

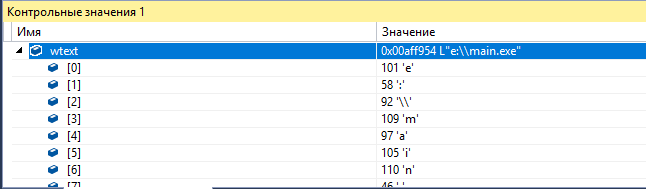


Рис. 4. Вкладка «контрольные значения»

Также стандартные средства VisualStudio предоставляют все необходимые средства для профилирования программ. Например можно посмотреть сколько процентов от общего времени работы программы занимает время работы какой-либо функции (рисунок 5)

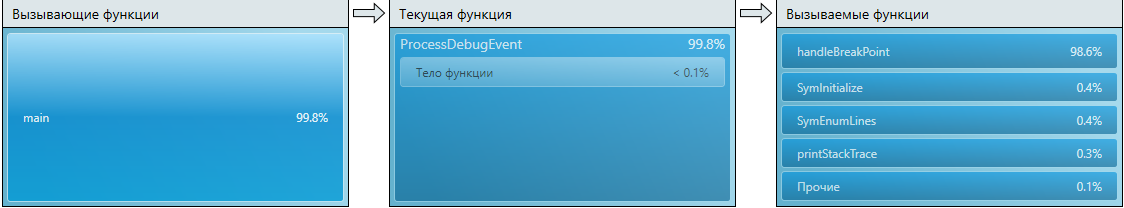


Рис. 5. Время работы отдельной функции

С помощью утилиты Exeinfo возможно посмотреть различную информацию о exe файле, включая точку входа программы и количество резервируемой памяти. Ниже приведен вывод программы для exe файла исследуемой программы:

Portable Executable Header Information

===================================

Machine : Intel 386

Number Of Sections : 7

Number Of Symbols : 0

Optional Header Size : 224 Bytes

Time Stamp : 0x58ce7d45

Characteristics :

Optional Header Information

===================================

Linker Version : 12.0

Size Of Code : 232,960 Bytes

Size of initialized data : 69,120 Bytes

Size of uninitialized data : 0 Bytes

Section Alignment : 4,096 Bytes

File Alignment : 512 Bytes

OS Version : 6.0

Image Version : 0.0

Subsystem Version : 6.0

Size Of Image : 421,888 Bytes

Size Of Headers : 1,024 Bytes

Checksum : 0x00000000

Subsystem : Windows character subsystem

Reserve Stack Size : 0x00100000

Commit Stack Size : 0x00001000

Reserve Heap Size : 0x00100000

Commit Heap Size : 0x00001000

Base Address : 0x00400000

Entry Point : 0x0001be38

Base Of Code : 0x00001000

Base Of Data : 0x00001000

Чтобы отследить подключение динамических и статических библиотек в проекте, необходимо воспользоваться средством просмотра памяти в текущей точке программы.

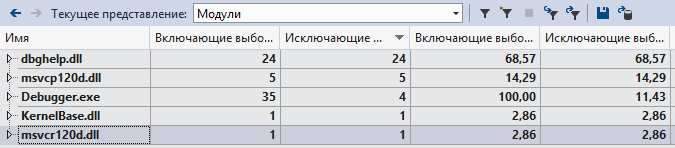


Рис. 6. Средство просмотра памяти

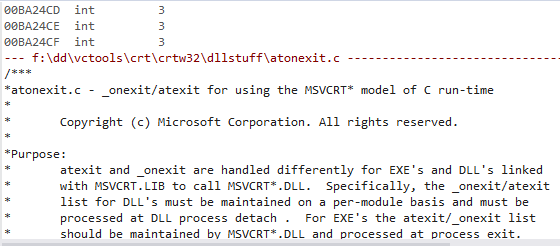


Рис. 7. Средство просмотра памяти

1. **Linux**

Для анализа работы программ для ОС Linux в большинстве случаев используется отладчик GDB[].GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Пользователь может изменять внутренние переменные программ и даже вызывать функции независимо от обычного поведения программы.

Пример дизассемблирования программы:

(gdb) disassemble main

Dump of assembler code for function main:

0x0804950c <+0>: lea 0x4(%esp),%ecx

0x08049510 <+4>: and $0xfffffff0,%esp

0x08049513 <+7>: pushl -0x4(%ecx)

0x08049516 <+10>: push %ebp

0x08049517 <+11>: mov %esp,%ebp

0x08049519 <+13>: push %ebx

0x0804951a <+14>: push %ecx

0x0804951b <+15>: sub $0x10,%esp

0x0804951e <+18>: mov %ecx,%ebx

0x08049520 <+20>: call 0x80486b0 <fork@plt>

0x08049525 <+25>: mov %eax,-0xc(%ebp)

0x08049528 <+28>: cmpl $0x0,-0xc(%ebp)

0x0804952c <+32>: jne 0x8049544 <main+56>

0x0804952e <+34>: mov 0x4(%ebx),%eax

Просмотр входной точки программы:

(gdb) info files

Symbols from "/home/user/SimpleDebugger/SimpleDebugger/Debug/SimpleDebugger".

Unix child process:

Using the running image of child process 1536.

While running this, GDB does not access memory from...

Local exec file:

`/home/user/SimpleDebugger/SimpleDebugger/Debug/SimpleDebugger',

file type elf32-i386.

Entry point: 0x80486d0

Вывод подключенных библиотек:

(gdb) info sharedlibrary

From To Syms Read Shared Object Library

0xb7fdf860 0xb7ff713c Yes /lib/ld-linux.so.2

No linux-gate.so.1

0xb7f129f0 0xb7f80cdd Yes (\*) /usr/lib/i386-linux-gnu/libstdc++.so.6

0xb7e8a600 0xb7eb7ed5 Yes /lib/i386-linux-gnu/i686/cmov/libm.so.6

0xb7e6a0a0 0xb7e7ff45 Yes (\*) /lib/i386-linux-gnu/libgcc\_s.so.1

0xb7cd5420 0xb7e0333e Yes /lib/i386-linux-gnu/i686/cmov/libc.so.6

(\*): Shared library is missing debugging information.

Вывод файлов, которые использует программа:

(gdb) info sources

Source files for which symbols have been read in:

/home/user/SimpleDebugger/SimpleDebugger/tbd.c,

/usr/include/i386-linux-gnu/sys/ptrace.h, /usr/include/stdio.h,

/home/user/SimpleDebugger/SimpleDebugger/<built-in>,

/home/user/SimpleDebugger/SimpleDebugger/tbd.h,

/usr/include/i386-linux-gnu/sys/user.h,

/usr/include/i386-linux-gnu/sys/types.h, /usr/include/libio.h,

/usr/include/i386-linux-gnu/bits/types.h,

/usr/lib/gcc/i586-linux-gnu/4.9/include/stddef.h,

/build/glibc-Tw0rNg/glibc-2.19/time/../sysdeps/unix/syscall-template.S,

/build/glibc-Tw0rNg/glibc-2.19/string/../sysdeps/i386/rawmemchr.S,

/build/glibc-Tw0rNg/glibc-2.19/string/../sysdeps/i386/i586/memcpy.S,

Детально память программы возможно изучить с помощью утилиты valgrind, если запустить её с флагом –tool=callgrind:

user@debian:~/SimpleDebugger/SimpleDebugger/Debug$ valgrind --tool=callgrind ./SimpleDebugger

==1638== Callgrind, a call-graph generating cache profiler

==1638== Copyright (C) 2002-2013, and GNU GPL'd, by Josef Weidendorfer et al.

==1638== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info

==1638== Command: ./SimpleDebugger

==1638==

==1638== For interactive control, run 'callgrind\_control -h'.

In debugger process 1638

==1639==

==1639== Events : Ir

==1639== Collected : 1382610

==1639==

==1639== I refs: 1,382,610

Exit from debugging cycle

Program exited with status (0)

==1638==

==1638== Events : Ir

==1638== Collected : 1409464

==1638==

==1638== I refs: 1,409,464

**Вывод**

При выполнении данной работы были проведены анализ и профилирование проекта, разработанного в прошлом семестре – Debugger. Проект был исследован и в ОС Windows, и в ОС Linux.

В ОС Windows все необходимые средства содержатся в средстве разработки Visual Studio, кроме просмотра заголовков исполняемых файлов, для этого была использована утилита exeinfo. В ОС Linux также все необходимые средства содержатся в одном пакете – gdb. Данная утилита является очень удобным и многофункциональным средством отладки и профилирования программ, но для детального изучения памяти программы лучше воспользоваться утилитой valgrind.

**Список использованных источников**

1. Интернет-источник. Microsoft Developer Network. <https://msdn.microsoft.com/ru-ru/dn308572.aspx> Дата обращения: 15.03.2017
2. Valgrind documentation. <http://valgrind.org/docs/manual/index.html>
3. GDB documentation. <https://www.gnu.org/software/gdb/documentation/>