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

**Цель работы**

Ранее для ОС Windows и Linux было реализовано приложение, осуществляющее логирование запуска новых приложений в системе (выводит в файл информацию о времени запуска приложения, пользователе, приоритете процесса и т.д.). Необходимо:

* Провести анализ загрузки приложения в ОЗУ
* Профилирование функционирования приложения
* Трассировку

**Выполнение работы**

**Windows**

**Анализ адресного пространства**

У каждого процесса есть свое виртуальное адресное пространство, в котором расположены все необходимые данные и исходный код программ. Посмотреть состояние виртуальной памяти процесса можно с помощью отладчика. В данной работе использовалась среда разработки Visual Studio 2015 и встроенный отладчик этой среды.

Для просмотра адресного пространства процесса необходимо установить точку останова в любой достижимой строчке программы запустить отладку. Далее через меню «Отладка»-«Окна»-«Память» можно открыть 4 окна просмотра памяти программы, каждое из которых работает независимо. Данные окна позволяют просматривать актуальное состояние памяти программы во время ее выполнения, редактировать какие-либо ячейки памяти и искать ячейки памяти, в которых расположены различные объекты программы (переменные, функции и т.д.) по их имени.

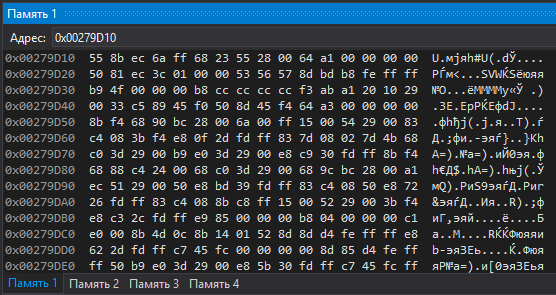


Рис. 1. Адрес функции main в памяти

На рисунке 1 приведен пример поиска адреса функции main в памяти процесса. Как видно, функция расположена по адресу 0x00279D10. При этом, если осуществить пересборку проекта, то адрес функции main изменится (рис. 2).

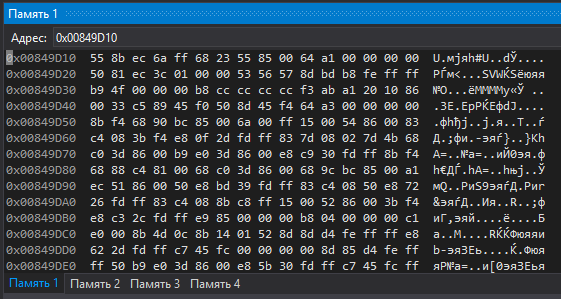


Рис. 2. Адрес функции main после перекомпиляции проекта

Так же, посмотреть расположение исходного кода программы в памяти можно с помощью инструмента дизассемблирования отладчика («Отладка»-«Окна»-«Дизассемблированный код»).

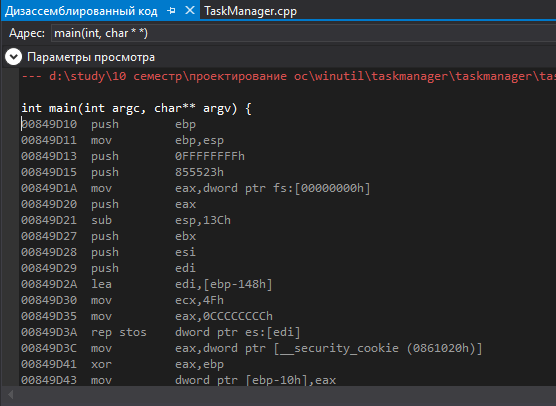


Рис. 3. Дизассемблированный код программы в памяти

Так же, с помощью программы PE Explorer можно определить адрес точки входа программы. Как показано на рисунке 4, точка входа анализируемой программы расположена по адресу 0x0041C37A. При перекомпиляции программы этот адрес не меняется. Данный адрес разный для различных исполняемых файлов.

**Работа загрузчика программ**

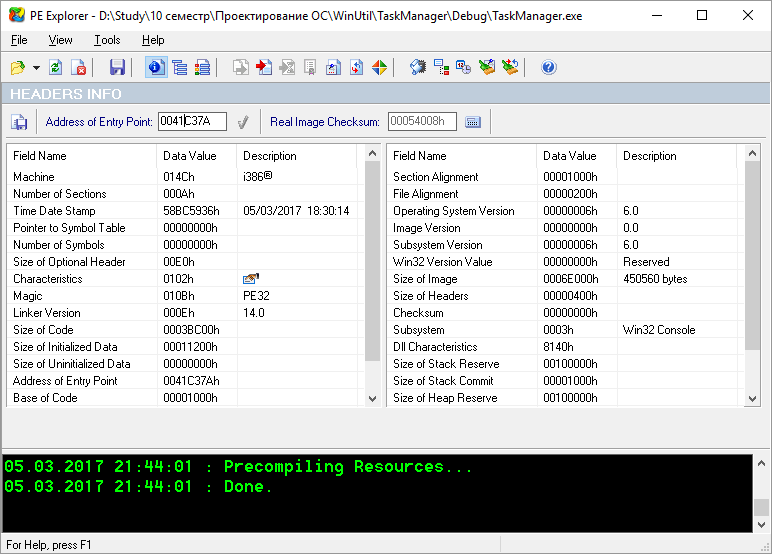


Рис. 4. PE Explorer.

Далее рассмотрим процесс профилирования приложений в Windows. Воспользуемся стандартными возможностями среды Visual Studio. Для запуска теста производительности необходимо выбрать в меню «Отладка»-«Профилировщик»-«Тест производительности». При этом необходимо отметить, что желательно собрать проект в режиме «Release», потому что при сборке в режиме отладки код программы инструментируется дополнительными инструкциями и меньше оптимизируется. Пример отчета приведен на рисунке 5.

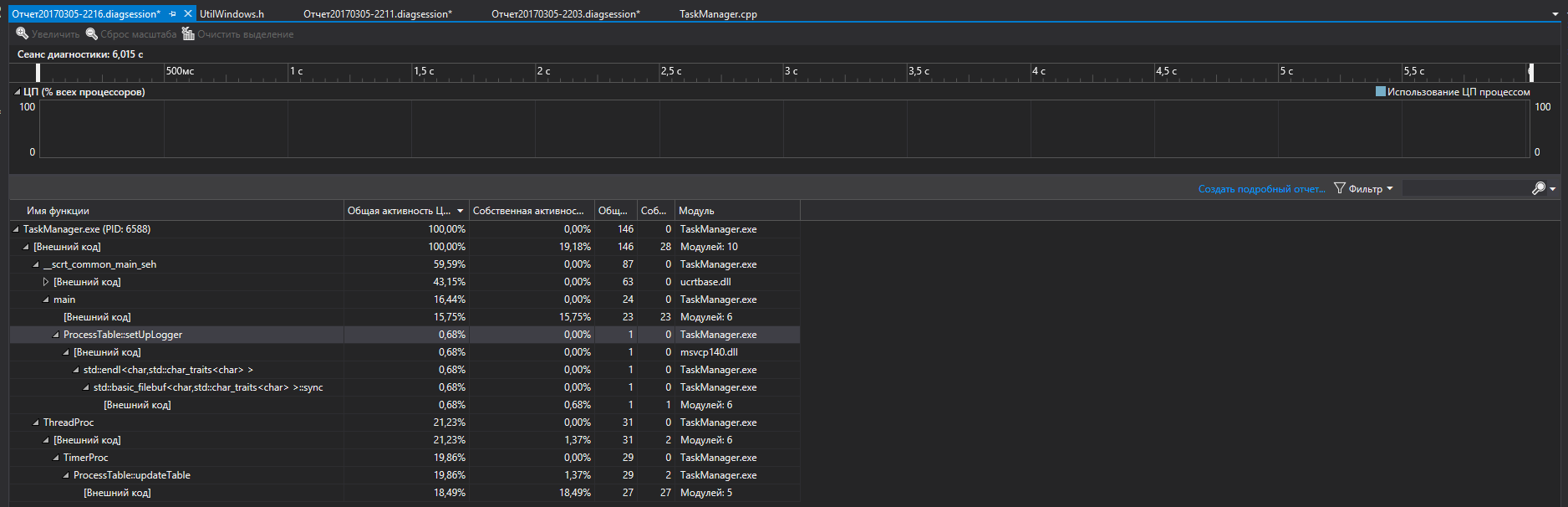


Рис. 5. Пример отчета о производительности

Отчет о производительности утилиты приведен на рисунке 6.

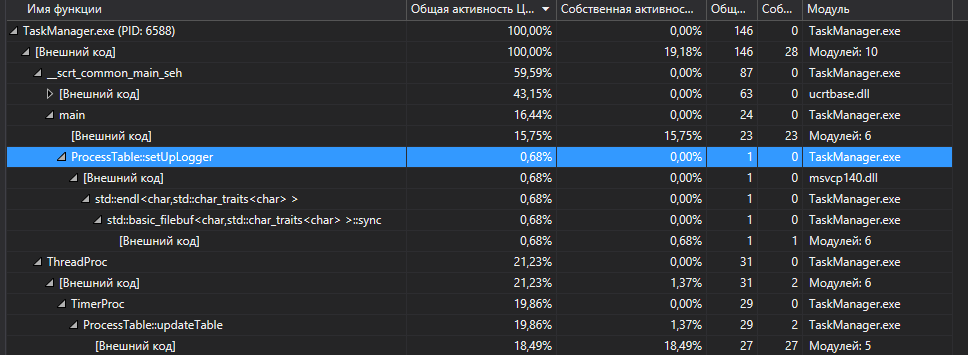


Рис. 6. Производительность утилиты

Как видно на рисунке, программа запускается в 2 потока. Мы видим, в основном потоке программы большинство ресурсов тратится на вызовы внешних библиотек, например, на внешний код из ucrtbase.dll (динамическая библиотека стандартной библиотеки С). Второй поток занимает 21% всех ресурсов процессора. Мы видим, что основные ресурсы при этом так же тратятся на вызовы внешних библиотек. Действительно, в функции updateTable() происходят вызовы функций из написанной нами библиотеки, которая производит достаточно ресурсоемкие системные вызовы (снимок таблицы процессов и т.д.).

Для анализа используемой памяти можно так же использовать отладчик Visual Studio. Он позволяет отслеживать использование памяти программой в реальном времени, делать снимки памяти а так же следить за состоянием кучи. Результаты анализа программы приведены на рисунке 7.

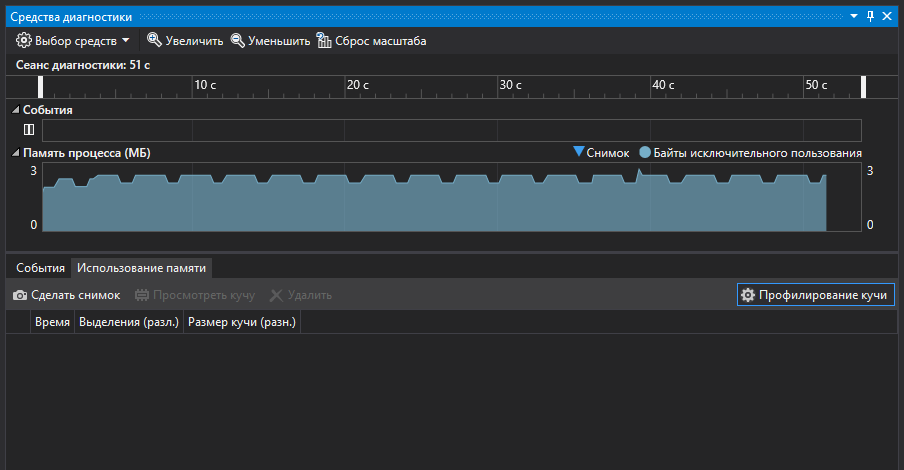


Рис. 7. Профилирование памяти программы

На рисунке видно, что утилита требует около 3 Мб памяти. При этом, занимаемая память циклически изменяется с периодом в 1 секунду. Это происходит из-за того, что раз в секунду происходит обновление таблицы процессов. При этом мы видим, что куча пустая, т.к. написанная утилита не выделяет память на куче.

**Анализ использования библиотек**

Утилита реализована в 2 вариантах: с использование статической библиотеки и динамической. Статическая библиотека связывается с программой на этапе компиляции, поэтому она сразу включена в адресное пространство программы. Проверить это можно установив точку останова на первую же строчку программы и проанализировав адресное пространство.

Рассмотрим адресное пространство процесса при работе со статической библиотекой. Для анализа воспользуемся инструментом дизассемблирования кода и монитором памяти. На рисунках 8 и 9 мы можем видеть, что исходный библиотеки изначально расположен в памяти программы.

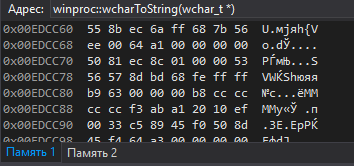


Рис. 8. Функция статической библиотеки в адресном пространстве процесса

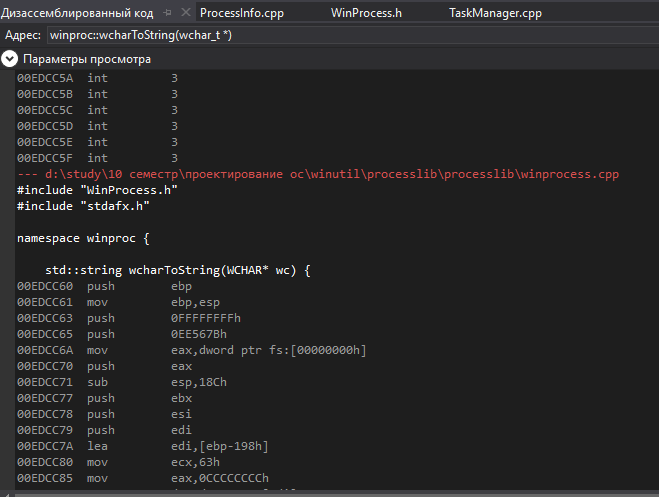


Рис. 9. Исходный код статической библиотеки в дизассемблированном коде программы

**Linux**

**Анализ адресного пространства**

Для анализа адресного пространства процесса можно воспользоваться средствами отладчика gdb. Так же можно проанализировать дизассемблированный код утилиты с помощью objdump. Рассмотрим оба варианта.

Определим адрес точки входа с помощью дизассемблирования:

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ objdump -d ./task\_manager > disassemble.out

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ head -n 20 disassemble.out

./task\_manager: file format elf64-x86-64

Disassembly of section .init:

000000000040ea58 <\_init>:

40ea58: 48 83 ec 08 sub $0x8,%rsp

40ea5c: 48 8b 05 95 c5 20 00 mov 0x20c595(%rip),%rax # 61aff8 <\_DYNAMIC+0x210>

40ea63: 48 85 c0 test %rax,%rax

40ea66: 74 05 je 40ea6d <\_init+0x15>

40ea68: e8 43 05 00 00 callq 40efb0 <\_ZNSt8ios\_base4InitD1Ev@plt+0x10>

40ea6d: 48 83 c4 08 add $0x8,%rsp

40ea71: c3 retq

Как мы видим, точка входа программа расположена по адресу 0x40ea58. С помощью gdb можно получить ту же информацию. Так же модно посмотреть адреса всех остальных функций, например, функции main:

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ gdb ./task\_manager

Reading symbols from ./task\_manager...done.

(gdb) print \_init

$1 = {<text variable, no debug info>} 0x40ea58 <\_init>

(gdb) print main

$2 = {int (int, char \*\*)} 0x40f0d8 <main(int, char\*\*)>

(gdb)

Мы видим, что адрес точки входа совпадает с тем что показал дизассемблированный код. Так же можно используя gdb убедиться в том, что статические библиотеки сразу связываются с исполняемым файлом:

(gdb) print util::

getOwnerByPID[abi:cxx11](int)

getProcessList()

getSystemStartTime()

getSystemStartTime()::starttime

getTimeFromClock(std::\_\_cxx11::basic\_string<char, std::char\_traits<char>, std::allocator<char> > const&)

is\_number(std::\_\_cxx11::basic\_string<char, std::char\_traits<char>, std::allocator<char> > const&)

(gdb) print util::getProcessList

$7 = {std::vector<int, std::allocator<int> > (void)} 0x41592c <util::getProcessList()>

(gdb) print util::getSystemStartTime

$8 = {time\_t (void)} 0x415a99 <util::getSystemStartTime()>

(gdb)

Мы получили адреса функций из статической библиотеки. Теперь проанализируем программу, подключающуюся к динамической библиотеке.

(gdb) print util::get

getOwnerByPID[abi:cxx11](int)

getOwnerByPID[abi:cxx11](int)@plt

getProcessList()

getProcessList()@plt

getTimeFromClock(std::\_\_cxx11::basic\_string<char, std::char\_traits<char>, std::allocator<char> > const&)

getTimeFromClock(std::\_\_cxx11::basic\_string<char, std::char\_traits<char>, std::allocator<char> > const&)@plt

(gdb) print util::getProcessList()

No symbol "util" in current context.

(gdb) b ProcessTable.cpp:22

Breakpoint 1 at 0x40f72d: file /home/kivi/sp\_labs/task\_manager\_static/ProcessTable.cpp, line 22.

(gdb) r

Starting program: /home/kivi/sp\_labs/task\_manager\_static/task\_manager

[Thread debugging using libthread\_db enabled]

Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".

Printing results into: ./task\_manager.out

[New Thread 0x7ffff6505700 (LWP 5024)]

[Switching to Thread 0x7ffff6505700 (LWP 5024)]

Thread 2 "task\_manager" hit Breakpoint 1, ProcessTable::updateTable (this=0x6194a0 <table>) at /home/kivi/sp\_labs/task\_manager\_static/ProcessTable.cpp:22

22 auto&& pids = util::getProcessList();

(gdb) print util::getProcessList

$1 = {std::vector<int, std::allocator<int> > (void)} 0x7ffff79b545f <util::getProcessList()>

(gdb)

Как видим, при запуске отладчика он знает о наличии функций библиотеки, так как автоподсказка выводит их имена. Однако, эти функции не находятся в адресном пространстве процесса, т.к. при попытке получить их адреса отладчик выводит ошибку. Далее была установлена точка останова на строчку программы, в которой происходит вызов функции библиотеки. После запуска программы и дойдя то точки останова, мы видим, что можем получить адрес функции динамической библиотеки (в примере мы получили адрес 0x7ffff79b545f).

**Профилирование**

Для профилирования программы воспользуемся утилитой valgrind. Данная утилита включает в себя набор анализаторов, в том числе анализатор производительности и анализатор использования памяти.

Запустим анализ производительности:

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ valgrind --tool=callgrind ./task\_manager

==5201== Callgrind, a call-graph generating cache profiler

==5201== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.

==5201== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==5201== Command: ./task\_manager

==5201==

==5201== For interactive control, run 'callgrind\_control -h'.

Printing results into: ./task\_manager.out

exit

==5201==

==5201== Events : Ir

==5201== Collected : 73281674

==5201==

==5201== I refs: 73,281,674

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ ls -l

total 1616

-rwxrwxr-x 1 kivi kivi 120184 дек 9 10:05 a.out

-rw------- 1 kivi kivi 261917 мар 6 21:29 callgrind.out.5201

……

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$

Данная утилита генерирует выходной файл с названием callgrind.out. Просмотреть его можно и в текстовом виде, однако это очень неудобно. Для визуализации результатов можно воспользоваться утилитой KCacheGrind (рисунок 10). Данный инструмент имеет много режимов отображения. Выберем режим профилирования ELF-объектов.

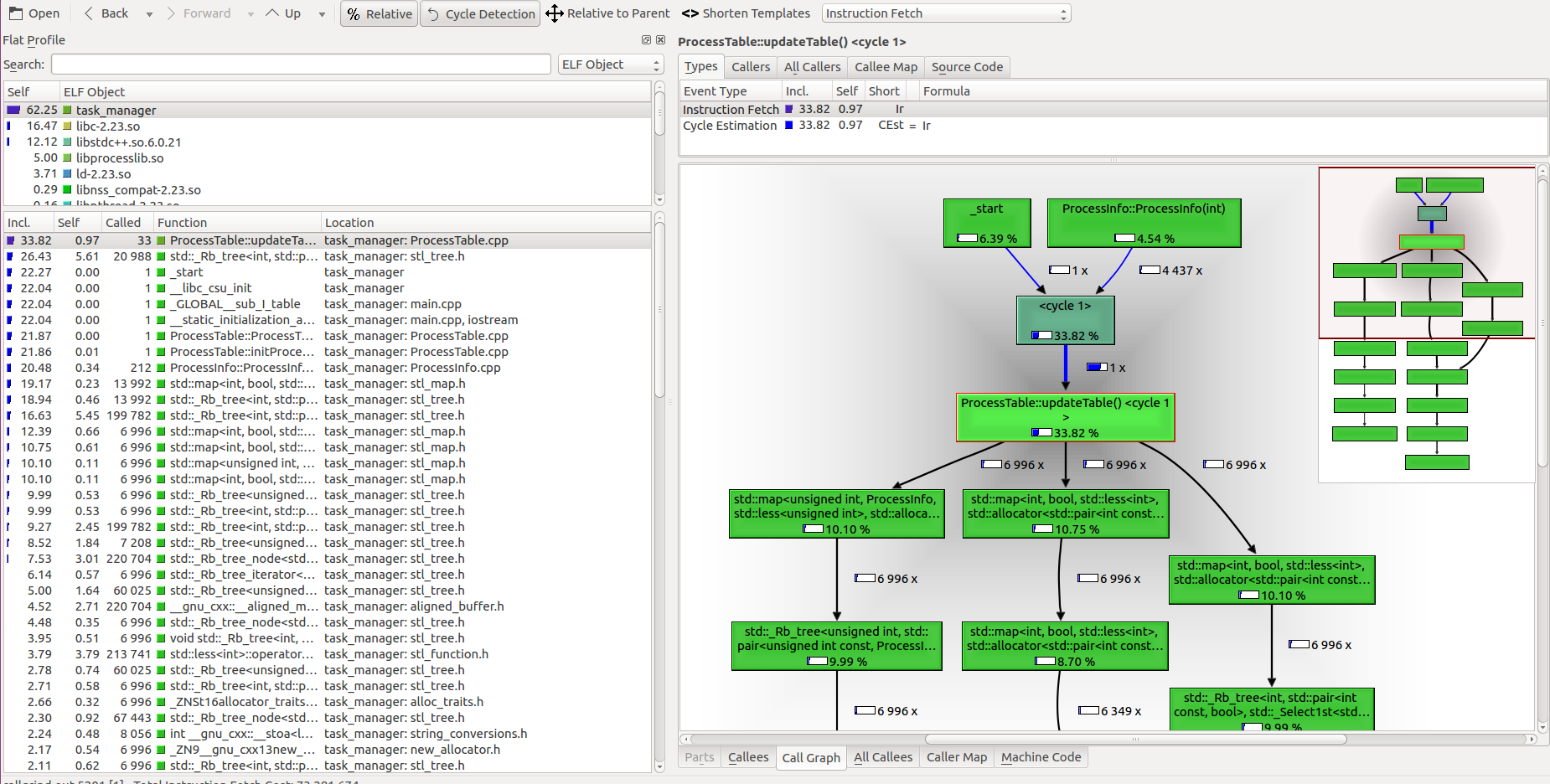


Рис. 10. Анализ производительности в KCacheGrind

В верхней левой части окна можно выбрать анализируемый ELF-объект. Слева снизу мы можем увидеть список функций в данном объекте и посмотреть их ресурсоемкость (левый столбец). Выбрав определенную функцию, мы справа можем увидеть граф вызовов функций с отображением их ресурсоемкости. Как видно самая ресурсоемкая функция в анализируемом модуле – ProcessTable::updateTable(), что не удивительно, т.к. в ней выполняется обновление таблицы процессов (наиболее сложная операция). Этот результат так же совпадает с результатами в Windows.

Для анализа используемой памяти используется инструмент massif. Он запускается так же как и callgrind и так же выводит результаты в текстовом формате в файл.

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ valgrind --tool=massif ./task\_manager

==6027== Massif, a heap profiler

==6027== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas Nethercote

==6027== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==6027== Command: ./task\_manager

==6027==

Printing results into: ./task\_manager.out

exit

==6027==

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$ massif-visualizer massif.out.6027

massif-visualizer(6040)/kdeui (kdelibs): Attempt to use QAction "toggleDataTree" with KXMLGUIFactory!

massif-visualizer(6040)/kdeui (kdelibs): Attempt to use QAction "toggleAllocators" with KXMLGUIFactory!

description: "(none)"

command: "./task\_manager"

time unit: "i"

snapshots: 95

peak: snapshot # 5 after "1.79905e+07i"

peak cost: "169.8 KiB" heap "7.9 KiB" heap extra "0 B" stacks

kivi@kivi-VirtualBox:~/sp\_labs/task\_manager\_static$

Для просмотра отчета massif можно использовать инструмент massif-vizualizer (рис. 11). На графике слева видно, что программа использует около 144 Кб памяти. График при этом цветом выделяет какой объем в памяти занимает каждый объект. Справа можно выбрать один из снимков памяти (который делает massif во время анализа) и более подробно посмотреть какие объекты и где занимают память.

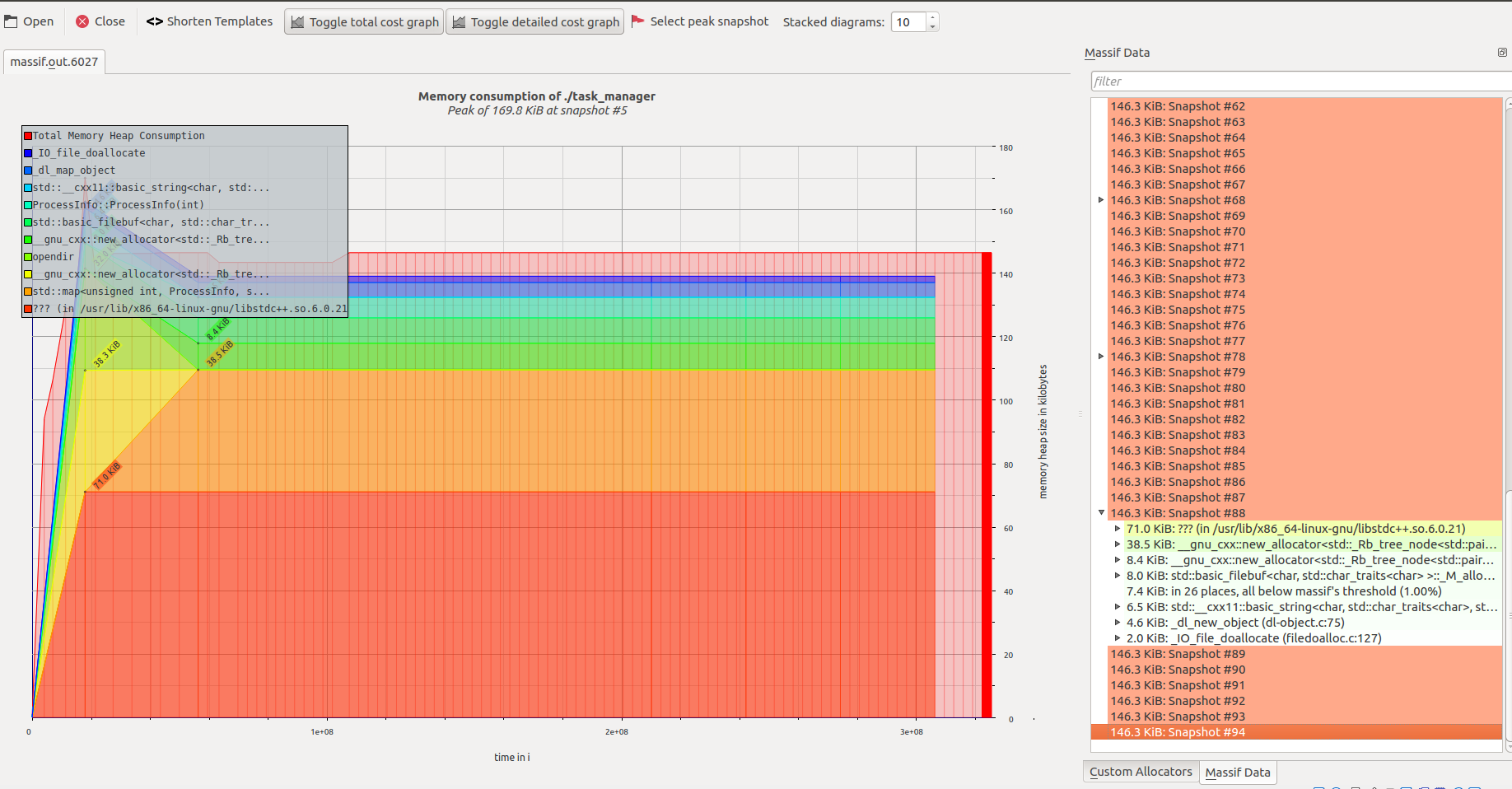


Рис. 11. Анализ памяти с использованием massif

**Вывод**

В данной работе были изучены основы анализа прикладных утилит в ОС Windows и Linux. Сначала было проанализировано адресное пространство программы. С помощью инструментов отладки и дизассемблирования были определены адреса точек входа программ, функции main и других объектов программы. Так же было проанализировано распределение статических и динамических библиотек в адресном пространстве процесса и определено, что статические библиотеки изначально располагаются в памяти процесса, а динамические подгружаются в память во время выполнения. Результаты в обеих ОС были схожи.

Далее было произведено профилирование программ. Профилирование показало, что наиболее ресурсоемкими функциями в программах являются те, которые производят вызовы функций WinAPI или системных вызовов Linux.

**Список использованных источников**

1. Документация WinAPI и Visual Studio: <https://msdn.microsoft.com/library>
2. Сайт gdb: <https://www.gnu.org/software/gdb/>
3. Сайт valgrind: http://valgrind.org/