Лабораторная работа № 3 по курсу дискретного анализа: Исследование качества программ

Выполнил студент группы 08-207 МАИ *Хренов Геннадий.*

Условие

1. Для реализации словаря из предыдущей лабораторной работы необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить. Результатом лабораторной работы является отчёт, состоящий из:
   Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
   Выводов о найденных недочётов.
   Сравнение работы исправленной программы с предыдущей версии.
   Общих выводов о выполнении лабораторной работы, полученном опыте.

Описание

Во время написания программы, особенно в период обучения необходимо всегда следить за такими аспектами как: полное освобождение выделеной памяти, эффективность выполнения, расход памяти. Для этого нужно знать методы инструментирования и профилирования. Ниже представлены средства, которые я использовал для отладки своей работы.

Valgrind

Утечки памяти одни из самых трудных для обнаружения ошибок, поэтому на места выделения и освобождения памяти стоит обращать особое внимание. Valgrind хорошо известен как мощное средство поиска ошибок работы с памятью. Он состоит из ядра и утилит на основе этого ядра. Я работал с memcheck - основным модулем, обеспечивающим обнаружение утечек памяти, и прочих ошибок, связанных с неправильной работой с областями памяти. Работать с valgrind просто — его поведение полностью управляется опциями командной строки, а также не требует специальной подготовки программы. Если программа запускается командой "программа аргументы то для ее запуска под управлением valgrind, необходимо в начало этой командной строки добавить слово valgrind, и указать опции, необходимые для его работы. По умолчанию, valgrind запускает модуль memcheck, однако пользователь может указать какой модуль должен выполняться с помощью опции –tool, передав в качестве аргумента имя нужного модуля. Пример работы:

```
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ valgrind ./solution
==13917== Memcheck, a memory error detector
==13917== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13917== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13917== Command: ./solution
==13917==
+ a 12
OK
A
OK: 12
- a
OK
a
NoSuchWord
==13917==
==13917== HEAP SUMMARY:
==13917==     in use at exit: 0 bytes in 0 blocks
==13917==   total heap usage: 6 allocs, 6 frees, 75,105 bytes allocated
==13917==
==13917== All heap blocks were freed -- no leaks are possible
==13917==
==13917== For counts of detected and suppressed errors, rerun with: -v
==13917== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
gennadii@lenovo-b560:~/workdir/da/laba2/solution$
```

Как видим, утечек памяти нет. Некоторые полезные ключи:
–log-fd - позволяет указать дескриптор файла, в который будет выводиться отчет о работе (по умолчанию это число 2 — стандартный вывод сообщений об ошибках).
–time-stamp - (yes или no, по умолчанию no) приводит к выдаче временных меток в отчет о работе (время отсчитывается от начала работы программы).
Но неверно думать, что valgrind умеет работать только с утечками. У него есть несколько других полезных функций, которые пригодились мне в работе. Например, valgrind указывает на неправильное освобождение памяти. Сообщение выглядит примерно так:

```
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ make
g++ -std=c++11 -O2 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result -pedantic  -o solution lab21.cpp rbtree.h
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ valgrind ./solution
==13957== Memcheck, a memory error detector
==13957== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13957== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13957== Command: ./solution
==13957==
+ a 12
OK
+ A 2
Exist
==13957== Mismatched free() / delete / delete []
==13957==    at 0x4C3123B: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13957==    by 0x109076: main (in /home/gennadii/workdir/da/laba2/solution/solution)
==13957==  Address 0x5b7e730 is 0 bytes inside a block of size 257 alloc'd
==13957==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13957==    by 0x108FF9: main (in /home/gennadii/workdir/da/laba2/solution/solution)
==13957==
```

Другой тип операции, которую обнаруживает Valgrind, это использование неинициализированного значения в условном операторе. Встретилась такая ошибка при записи нетерминированной строки в бинарный файл:

```
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ make
g++ -std=c++11 -O2 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result -pedantic  -o solution lab21.cpp rbtree.h
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ valgrind ./solution
==13979== Memcheck, a memory error detector
==13979== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13979== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13979== Command: ./solution
==13979==
+ a 12
OK
! Save t.bin
OK
==13979== Syscall param write(buf) points to uninitialised byte(s)
==13979==    at 0x54ED154: write (write.c:27)
==13979==    by 0x546B1BC: _IO_file_write@@GLIBC_2.2.5 (fileops.c:1203)
==13979==    by 0x5469F50: new_do_write (fileops.c:457)
==13979==    by 0x5469F50: _IO_do_write@@GLIBC_2.2.5 (fileops.c:433)
==13979==    by 0x546930F: _IO_file_close_it@@GLIBC_2.2.5 (fileops.c:136)
==13979==    by 0x545B386: fclose@@GLIBC_2.2.5 (iofclose.c:53)
==13979==    by 0x1090F4: main (in /home/gennadii/workdir/da/laba2/solution/solution)
==13979==  Address 0x5b7e9a2 is 2 bytes inside a block of size 4,096 alloc'd
==13979==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13979==    by 0x545B18B: _IO_file_doallocate (filedoalloc.c:101)
==13979==    by 0x546B378: _IO_doallocbuf (genops.c:365)
==13979==    by 0x546A497: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:759)
==13979==    by 0x54689EC: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1266)
==13979==    by 0x545C976: fwrite (iofwrite.c:39)
==13979==    by 0x1091DF: main (in /home/gennadii/workdir/da/laba2/solution/solution)
==13979==
```

В общем, Valgrind позволяет программисту запустить исполняемый файл внутри своей собственной среды, в которой он проверяет непарные вызовы malloc и другие виды недопустимого использования памяти или неверных операций с памятью.

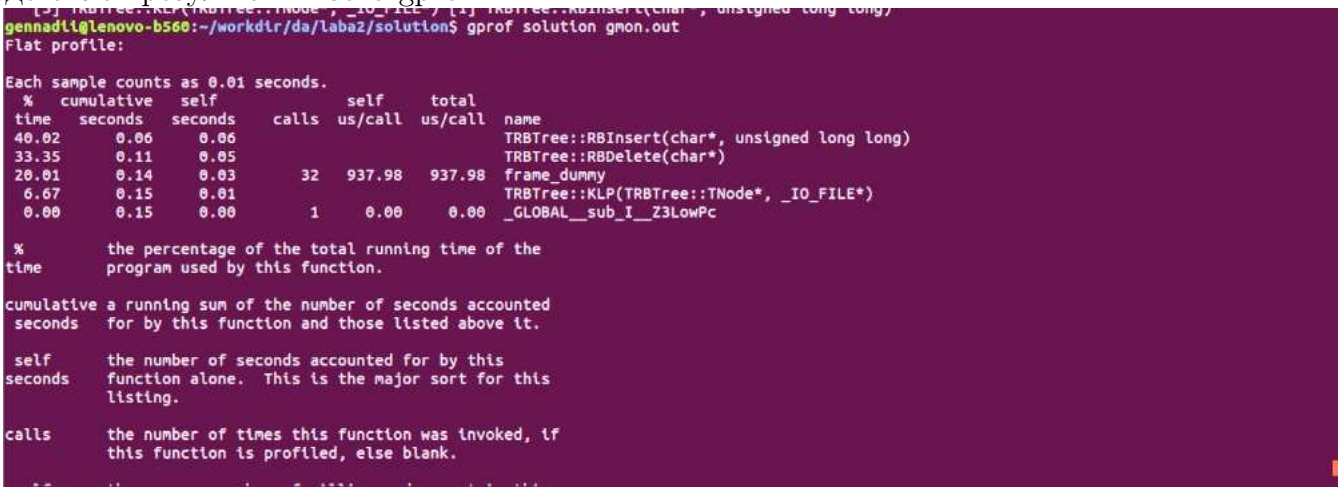
massif

Massif используется для измерения количества памяти, которую использует программа. Также massif учитывает и дополнительные байты для служебной информации или выравнивания. Massif это модуль valgrind, поэтому запускается так: valgrind –tool=massif... Подробные данные пишутся в файл massif.out.id процесса. Вот часть вывода:

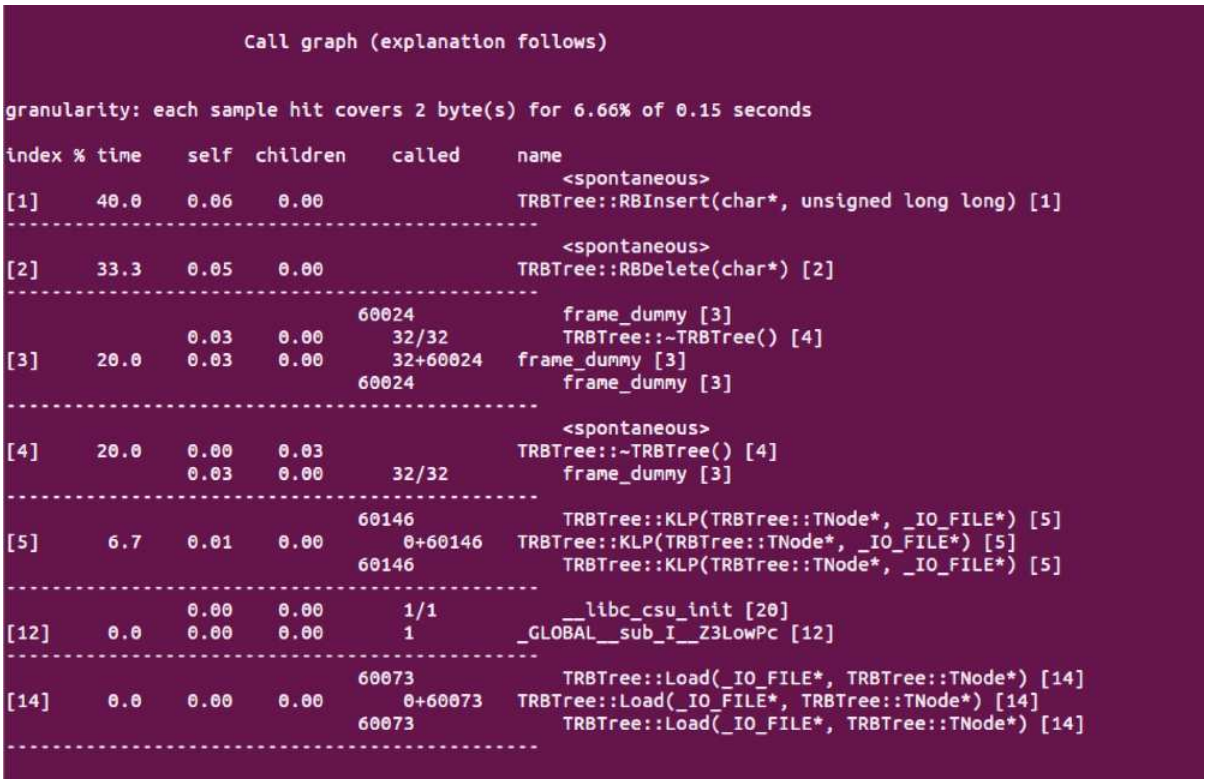
```
gennadii@lenovo-b560:~/workdir/da/laba2/solution$ ms_print massif.out.11434
--------------------------------------------------------------------------------
Command:            ./solution
Massif arguments:   (none)
ms_print arguments: massif.out.11434
--------------------------------------------------------------------------------


    MB
192.5^                                                                       #
     |                                                                     :@:#
     |                                                                 :::::@:#
     |                                                              :::@:::::@:#
     |                                                           @:::::@:::::@:#
     |                                                       ::::@:::::@:::::@:#
     |                                                    :@::::::@:::::@:::::@:#
     |                                               ::::@::::::@:::::@:::::@:#
     |                                          :::::: :@::::::@:::::@:::::@:#
     |                                     ::::: ::: :@::::::@:::::@:::::@:#
     |                                 :::::::::: ::: :@::::::@:::::@:::::@:#
     |                             :::::: :::::: ::: :@::::::@:::::@:::::@:#
     |                          :::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |                       @@:::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |                    :::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |                 ::::::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |            @:::::: :::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |         :@:@::: :: :::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |      @:::@:@::: :: :::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
     |   ::@:@:::@:@::: :: :::@ :::::@:::: :::::: ::: :@::::::@:::::@:::::@:#
   0 +--------------------------------------------------------------------->Gi
     0                                                                   3.693

Number of snapshots: 84
 Detailed snapshots: [5, 7, 11, 13, 23, 29, 49, 59, 69, 79, 82 (peak)]

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
```

Gprof

Gprof используется для профилирования.При вызове компилятора нужно указать параметр -pg. Gprof позволит узнать какие и сколько раз функции вызывались в программе, а также узнать время их работы. При компиляции на коцах функции расставляются контрольные точки. Разница между этими точками и есть время исполнения. Ниже представлен результат вызова gprof:



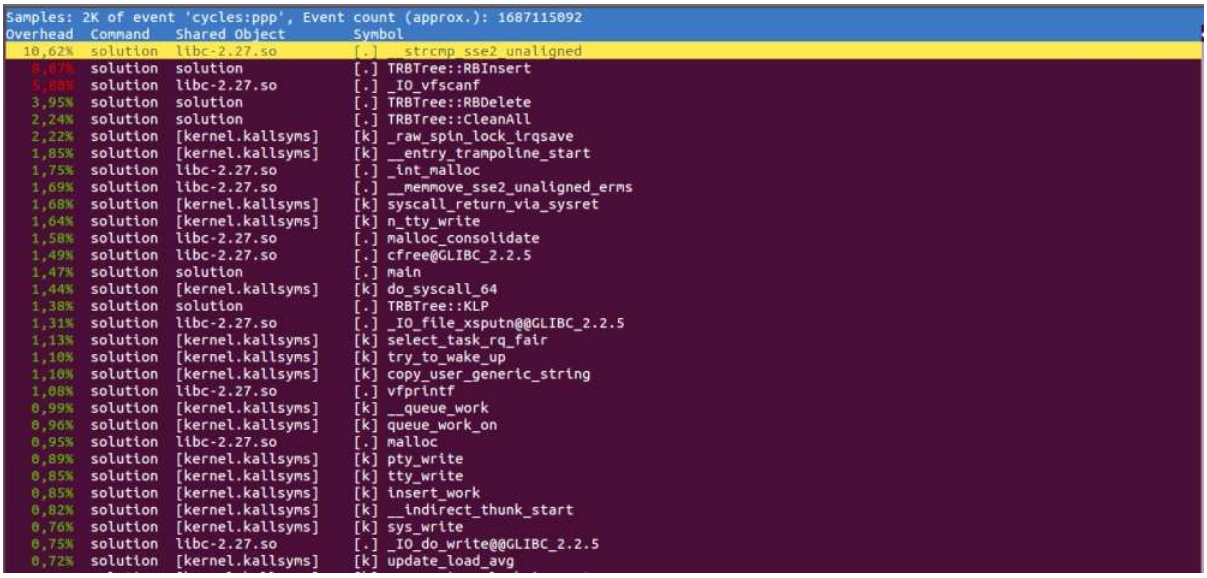
Это лишь первая часть работы утилиты. Вторая - это граф вызовов. Он показывает информацию о количестве вызовов, списки вызываемых и вызывающих функций. Благодаря этой информации можно выделить фрагменты кода, использующие значительное время работы процессора или использующие большое количество вызовов функций.



```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 6.66% of 0.15 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]     40.0    0.06    0.00                 TRBTree::RBInsert(char*, unsigned long long) [1]
-----------------------------------------------
                                               <spontaneous>
[2]     33.3    0.05    0.00                 TRBTree::RBDelete(char*) [2]
-----------------------------------------------
                                60024             frame_dummy [3]
                0.03    0.00      32/32           TRBTree::~TRBTree() [4]
[3]     20.0    0.03    0.00      32+60024    frame_dummy [3]
                                60024             frame_dummy [3]
-----------------------------------------------
                                               <spontaneous>
[4]     20.0    0.00    0.03                 TRBTree::~TRBTree() [4]
                0.03    0.00      32/32           frame_dummy [3]
-----------------------------------------------
                                60146             TRBTree::KLP(TRBTree::TNode*, _IO_FILE*) [5]
[5]      6.7    0.01    0.00       0+60146    TRBTree::KLP(TRBTree::TNode*, _IO_FILE*) [5]
                                60146             TRBTree::KLP(TRBTree::TNode*, _IO_FILE*) [5]
-----------------------------------------------
                0.00    0.00       1/1            __libc_csu_init [20]
[12]     0.0    0.00    0.00       1          _GLOBAL__sub_I__Z3LowPc [12]
-----------------------------------------------
                                60073             TRBTree::Load(_IO_FILE*, TRBTree::TNode*) [14]
[14]     0.0    0.00    0.00       0+60073    TRBTree::Load(_IO_FILE*, TRBTree::TNode*) [14]
                                60073             TRBTree::Load(_IO_FILE*, TRBTree::TNode*) [14]
-----------------------------------------------
```

Perf

Perf - ещё один профилировщик. Он работает на счётчиках производительности. Это регистры аппаратного обеспечения процессора, которые подсчитывают события, инструкции и т.д.. Perf выводит довольно общирные данные о выполнении программы. Я рассмотрел его подкоманды record и report. Record записывает данные, собранные о программе, report выводит их на экран.



```
Samples: 2K of event 'cycles:ppp', Event count (approx.): 1687115092
Overhead  Command   Shared Object        Symbol
  10,62%  solution  libc-2.27.so        [.] __strcmp_sse2_unaligned
   8,87%  solution  solution            [.] TRBTree::RBInsert
   5,80%  solution  libc-2.27.so        [.] _IO_vfscanf
   3,95%  solution  solution            [.] TRBTree::RBDelete
   2,24%  solution  solution            [.] TRBTree::CleanAll
   2,22%  solution  [kernel.kallsyms]   [k] _raw_spin_lock_irqsave
   1,85%  solution  [kernel.kallsyms]   [k] __entry_trampoline_start
   1,75%  solution  libc-2.27.so        [.] _int_malloc
   1,69%  solution  libc-2.27.so        [.] __memmove_sse2_unaligned_erms
   1,68%  solution  [kernel.kallsyms]   [k] syscall_return_via_sysret
   1,64%  solution  [kernel.kallsyms]   [k] n_tty_write
   1,58%  solution  libc-2.27.so        [.] malloc_consolidate
   1,49%  solution  libc-2.27.so        [.] cfree@GLIBC_2.2.5
   1,47%  solution  solution            [.] main
   1,44%  solution  [kernel.kallsyms]   [k] do_syscall_64
   1,38%  solution  solution            [.] TRBTree::KLP
   1,31%  solution  libc-2.27.so        [.] _IO_file_xsputn@@GLIBC_2.2.5
   1,13%  solution  [kernel.kallsyms]   [k] select_task_rq_fair
   1,10%  solution  [kernel.kallsyms]   [k] try_to_wake_up
   1,10%  solution  [kernel.kallsyms]   [k] copy_user_generic_string
   1,08%  solution  libc-2.27.so        [.] vfprintf
   0,99%  solution  [kernel.kallsyms]   [k] __queue_work
   0,96%  solution  [kernel.kallsyms]   [k] queue_work_on
   0,95%  solution  libc-2.27.so        [.] malloc
   0,89%  solution  [kernel.kallsyms]   [k] pty_write
   0,85%  solution  [kernel.kallsyms]   [k] tty_write
   0,85%  solution  [kernel.kallsyms]   [k] insert_work
   0,82%  solution  [kernel.kallsyms]   [k] __indirect_thunk_start
   0,76%  solution  [kernel.kallsyms]   [k] sys_write
   0,75%  solution  libc-2.27.so        [.] _IO_do_write@@GLIBC_2.2.5
   0,72%  solution  [kernel.kallsyms]   [k] update_load_avg
```

В выводе мы видим структурированные данные о функциях программы и времени, которое на них затрачено. Обращаю внимание на интересный интерфейс. Perf мне понравился больше gprofa, так как он выводит больше информации о системных вызовах и имеет более приятный интерфейс.

Выводы

Оказалось, что инструментирование и профилирование такие же трудоемкие процессы как и написание программы. Однако это дает огромное количество полезной информации о твоей программе и её слабых местах. Можно узнать, где течет память, где программа долго выполняется и почему, как программа вызывает свои функции и т.д.. Valgrind произвёл впечатление своей большой функциональностью и простотой пользования. Также отмечу работу gprof и её предоставление данных о времени, которые раньше я находил в ручную, с помощью таймеров и т.д.. Лабораторная работа оказалась достаточно интересной, я познакомился с некоторыми новыми средствами для повышения качества программы, которые не раз ещё буду исспользовать.