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

Выполнил студент группы 08-207 МАИ *Днепров Иван.*

Условие

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:

1. Дневника выплонения 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 недочётах.
2. Сравнение работы исправленной программы с предыдущей версией.
3. Общих выводов о выполнении лабораторной работы, полученном опыте.

Вариант задания: 3. PATRICIA.

Методы профилирования и используемые утилиты

Для исследования качества кодя я использовал три утилиты: gprof, perf и valgrind.

Gprof – программа для гарантированного профилировнаия. Она модифицирует код при компиляции, внедряя в него специальные библиотеки, собирающие информацию о выполнении программы, которые затем передаются профилировщику. Плюсом гарантированного профилирования является максимальная точность полученных данных (ни один вызов функции не будет пропущен). Минусом же такого подхода является необходимость перекомпилирования и невозможность профилования программ, исходники которых не доступны.

Для статического профилирования я использовал perf. Идея статического профилирования заключается в том, что проверяемая программа запускается и после чего от 100 до 1000 раз в секунду останавливается, а профилирующая утилита анализирует текущий контекст исполнения. После гего, используя отладочную информацию, утилита определяет вызываемую в текущий момент функцию и собирает статистику исполнения той или иной функции. Идея заключается в том, что если программа при выполнении обной функции была отсановленна в два раза больше раз, чем при выполнении другой, то он провела в первой функции в два раза больше времени, а, как следствие, эта функция потратила больше процессорного времени. Плюсы статического профилирования заключаются в отсутствии необходимости перекомпилировния программы для ее проверки, но точность такой проверки сильно зависить от сложности теста (для коротких тестов данные будут получены с большой погрешностью).

А для профилировнаия на эмуляторе я использовал valgring. Профилировние на эмуляторе заключается в том, что при выполнении программа эмулирует целевой процессор и во время выполнения профилируемого кода перхватывает все процессорные команды и анализирует их. Такое профилирование получается максимально точным без необходимости перекомпляции, но благодаря накладным расходам на эмуляцию, и очень медленным. Вообще, valgrind – очень мощьный инструмент, но я при его помощи проверил только утечки памяти, которых, кстати, неоказалось.

Процесс тестирования

Тут я прикладываю литинг работы с терминалом.

В начале я перекомпилирую программу и работаю с ней при помощи gprof. Как видно ниже, для моего теста (вставка или удаления различных элементов 69000 раз) больше всего времени занимает работа функции преобразования ключа к нижнему регистру, что означает, что функции добавления или удаления элемента дерева работают достаточно быстро.

Далее я работал с утилитой pref, которая показывает примерно то же самое, но мы можем посмотреть на работу программы и окружающие ее системные функции. В ванном случае самой прожорливой функцией оказалась фукнкция добавления элемента, которая вызывает функцию приводящую ключ к нижнему регистру, видимо все дело в прогрешности gprof.

И вконце я проверил свой код на утечки при помощи valgrind, который показал, что таковых не имеется.

```
vanyadneprov@ubuntu:~/Desktop/da2$ ./da2 < text.txt > output.txt
vanyadneprov@ubuntu:~/Desktop/da2$ gprof ./da2 < text.txt > output.txt
vanyadneprov@ubuntu:~/Desktop/da2$ gprof da2 gmon.out > profile-data.txt
vanyadneprov@ubuntu:~/Desktop/da2$ cat profile-data.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 33.35      0.03     0.03                             StrLower
 16.68      0.05     0.02                             WriteSubKey
 11.12      0.06     0.01                             CreateNewTPNode
 11.12      0.07     0.01                             TPKeyInsert
 11.12      0.08     0.01                             TPNodeDelete
 11.12      0.09     0.01                             _fini
  5.56      0.09     0.01                             ReadSubKey

 %         the percentage of the total running time of the
time       program used by this function.
```

```
cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.


Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
vanyadneprov@ubuntu:~/Desktop/da2$ sudo perf stat ./da2 <text.txt > output.txt

 Performance counter stats for './da2':

        141.888002      task-clock (msec)         #    0.970 CPUs utilized
                 4      context-switches          #    0.028 K/sec
                 1      cpu-migrations            #    0.007 K/sec
               795      page-faults               #    0.006 M/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
```

```
   <not supported>      branch-misses

       0.146303014 seconds time elapsed
vanyadneprov@ubuntu:~/Desktop/da2$ sudo perf record -e cycles -c 1000000 ./da2 < text
vanyadneprov@ubuntu:~/Desktop/da2$ sudo perf report
  30.50%  da2      [kernel.kallsyms]  [k] queue_work_on
  24.50%  da2      [kernel.kallsyms]  [k] do_syscall_64
  24.45%  da2      libc-2.27.so       [.] __GI___libc_write
   2.32%  da2      da2                [.] TPKeyInsert
   1.30%  da2      [kernel.kallsyms]  [k] finish_task_switch
   1.16%  da2      libc-2.27.so       [.] _int_malloc
   0.84%  da2      da2                [.] CommonSubstrLen
   0.70%  da2      libc-2.27.so       [.] _IO_vfscanf
   0.65%  da2      [kernel.kallsyms]  [k] native_set_pte_at
   0.56%  da2      [kernel.kallsyms]  [k] fsnotify
   0.51%  da2      [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
   0.51%  da2      da2                [.] ExtendedTPNodeSearch
   0.46%  da2      [kernel.kallsyms]  [k] n_tty_write
   0.42%  da2      [kernel.kallsyms]  [k] copy_user_generic_unrolled
   0.42%  da2      [kernel.kallsyms]  [k] exit_to_usermode_loop
   0.42%  da2      [kernel.kallsyms]  [k] tty_write
   0.42%  da2      [kernel.kallsyms]  [k] vfs_write
   0.37%  da2      [kernel.kallsyms]  [k] __vfs_write
   0.37%  da2      [kernel.kallsyms]  [k] clear_page_orig
   0.33%  da2      [kernel.kallsyms]  [k] __check_heap_object
   0.33%  da2      libc-2.27.so       [.] malloc
vanyadneprov@ubuntu:~/Desktop/da2$ valgrind --leak-check=full --leak-resolution=med .
==24888== Memcheck, a memory error detector
==24888== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==24888== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==24888== Command: ./da2
==24888==
==24888==
==24888== HEAP SUMMARY:
==24888==     in use at exit: 0 bytes in 0 blocks
==24888==   total heap usage: 195,236 allocs, 195,236 frees, 2,977,899 bytes allocate
==24888==
==24888== All heap blocks were freed -- no leaks are possible
==24888==
==24888== For counts of detected and suppressed errors, rerun with: -v
==24888== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Выводы

Каждый метод и утилита для профилирования хороша по-своему, но я считаю наиболее универсальной perf, благодаря тому, что для проверки приложения его не нужно перекомпилоровать, что не всегда представляется возможным. Perf имеет некую погрешность потму что она учитывает не всё время работы конкретной функции, а количество остановок в ней, и хотя их число не меньше 100 за одну секунду, некая погрешность всё-таки остается, и, как видно, при сравнении данных полученных perf и gprof, в список может не попасть функция, потребляющая наибольшее число процессорного времени, данные, полученные при таком профилировании тоже полезны. А предпочтение этой утилите я отдал за ее универсальность и высокуцу скорость работы. Но всё это условно, для каждого конкретного сценария использования нужно подбирать конкретную утилиту.