

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

Выполнил студент группы 08-208 МАИ *Куликов Алексей*.

Условие

Кратко описывается задача: 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.

Метод решения

Изучение утилит для исследования качества программ таких как gcov, gprof, perf, valgrind, heaptrack и их использование для оптимизации программы.

Valgrind

Valgrind – набор инструментов для обнаружения утечек памяти, отладки ее использования, а так же профилирования программы.

В ходе работы над данной лабораторной работой valgrind будет использован только в качестве инструмента для обнаружения утечек памяти т.к. для остальных двух целей существуют более совершенные инструменты.

Использование

```
==3759== HEAP SUMMARY:
==3759==      in use at exit: 9,515,695 bytes in 62,398 blocks
==3759==    total heap usage: 640,825 allocs, 578,427 frees, 152,212,587 bytes allocated
==3759==
==3759== 48 bytes in 1 blocks are possibly lost in loss record 1 of 11
==3759==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3759==    by 0x10D262: TAVL<TString, unsigned long long>::Deserialize(std::basic_ifstream<char, std::char_traits<char>>, std::ios_base::openmode) (in /home/alex/temp/DA/lab2/lab2)
==3759==    by 0x109518: main (in /home/alex/temp/DA/lab2/lab2)
==3759==
==3759== 514 bytes in 2 blocks are possibly lost in loss record 2 of 11
==3759==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3759==    by 0x10D27B: TAVL<TString, unsigned long long>::Deserialize(std::basic_ifstream<char, std::char_traits<char>>, std::ios_base::openmode) (in /home/alex/temp/DA/lab2/lab2)
==3759==    by 0x109518: main (in /home/alex/temp/DA/lab2/lab2)
==3759==
==3759== 4,470,831 (456,768 direct, 4,014,063 indirect) bytes in 9,516 blocks are definitely lost in loss record 3 of 11
==3759==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3759==    by 0x10C609: TAVL<TString, unsigned long long>::Insert(TString const&, unsigned long long) (in /home/alex/temp/DA/lab2/lab2)
==3759==    by 0x1092FD: main (in /home/alex/temp/DA/lab2/lab2)
==3759==
==3759== 5,044,302 (605,184 direct, 4,439,118 indirect) bytes in 12,608 blocks are definitely lost in loss record 4 of 11
==3759==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
```

```

==3759==    by 0x10D262: TAVL<TString, unsigned long long>::Deserialize(std::basic_ifstream<char, std::
==3759==    by 0x109518: main (in /home/alex/temp/DA/lab2/lab2)
==3759==
==3759== LEAK SUMMARY:
==3759==    definitely lost: 1,061,952 bytes in 22,124 blocks
==3759==    indirectly lost: 8,453,181 bytes in 40,271 blocks
==3759==    possibly lost: 562 bytes in 3 blocks
==3759==    still reachable: 0 bytes in 0 blocks
==3759==    suppressed: 0 bytes in 0 blocks
==3759==
==3759== For counts of detected and suppressed errors, rerun with: -v
==3759== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)

```

Valgrind'ом была найдена глупейшая ошибка при удалении из дерева в случае пустого правого/левого поддерева.

На всеобъемлющем (в моем представлении) и довольно объемном тесте при запуске была обнаружена крупная утечка памяти. Valgrind при запуске бинарника, откомпилированного с ключом `-g3` показал место выделения памяти, которую никто не возвращал назад. Методом исключения было обнаружено, что не возвращает память именно процедура удаления т.к. , если принудительно явно не удалять элемент в конце работы отработывал деструктор и отработывал корректно, возвращая всю выделенную память системе. Далее было проверено что будет, если удалить единственный элемент из дерева. Это как раз случай свободных левого-правого поддереьев. Все равно утекала память. Далее методом пристального взгляда в процедуру удаления была найдена проблема: возврат из функции происходил до возвращения памяти. Наши победили ихних! Память не течет.

gprof

gprof – утилита для профилирования программы.

Видно где сколько времени проводила программа и, соответственно, части кода которые следует оптимизировать.

```

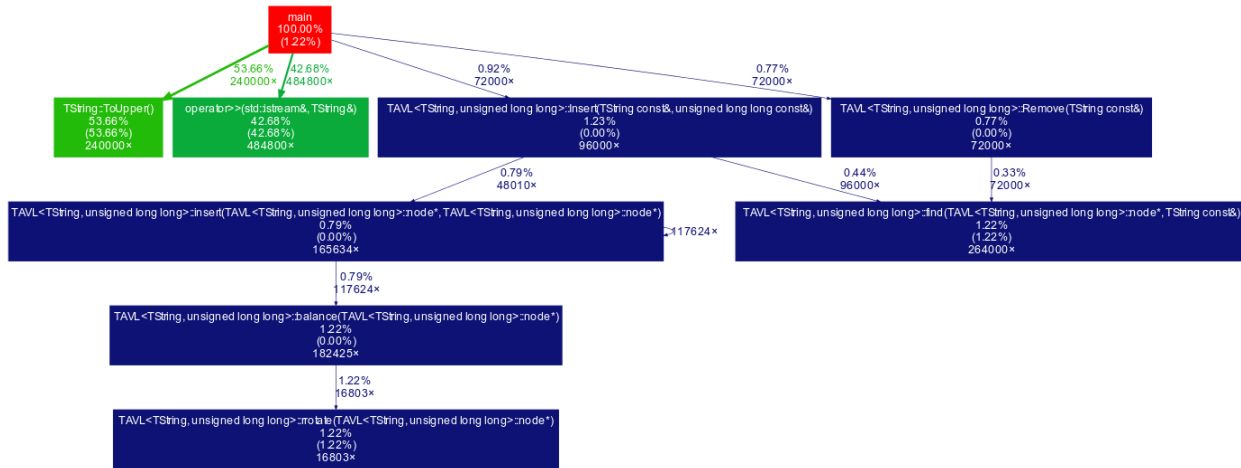
Each sample counts as 0.01 seconds.
%   cumulative    self           self         total
time  seconds    seconds   calls   us/call   us/call   name
53.68    0.44    0.44    240000    1.83     1.83  TString::ToUpper()
42.70    0.79    0.35    484800    0.72     0.72  operator>>(std::istream&, TString&)
 1.22    0.80    0.01    264000    0.04     0.04  TAVL<TString, unsigned long
long>::find(TAVL<TString, unsigned long long>::node*, TString const&)
 1.22    0.81    0.01    16803    0.60     0.60  TAVL<TString, unsigned long
long>::rrotate(TAVL<TString, unsigned long long>::node*)
 1.22    0.82    0.01                main
 0.00    0.82    0.00   1334574    0.00     0.00  TAVL<TString, unsigned long
long>::height(TAVL<TString, unsigned long long>::node*)
 0.00    0.82    0.00    643203    0.00     0.00  operator>(TString const&, TString const&)
 0.00    0.82    0.00    549600    0.00     0.00  operator==(TString const&, char const*)
 0.00    0.82    0.00    489621    0.00     0.00  operator<(TString const&, TString const&)

```

Использование:

```
g++ -pg -Wall main.cpp -o lab2
./lab2 < input > /dev/null
gprof ./lab2 < input > profile
```

Также можно строить графы вызовов для более наглядного представления при помощи утилиты gprof2dot.



Как можно видеть в таблице и на графе «бутылочное горлышко» программы – это две основные функции: ToUpper класса String и оператор чтения из потока того же класса String.

Обе эти функции вызываются на каждую команду пользователя и, по всей видимости, довольно затратны. Хотя для ввода строки из потока это и не удивительно, он был сделан довольно коряво, а вот тот факт, что преобразование строки к верхнему регистру будет потреблять какие-то значительные ресурсы меня удивил. Эта функция представляет из себя всего лишь цикл по длине строки, который если текущий символ буква в нижнем регистре просто меняет на ту же букву в верхнем.

Тем не менее ее удалось оптимизировать исключив лишнюю проверку на принадлежность буквам. Результат.

% time	cumulative seconds	self seconds	self calls	self us/call	total us/call	name
69.26	0.54	0.54	484800	1.11	1.11	operator>>(std::istream&, TString&)
29.50	0.77	0.23	240000	0.96	0.96	TString::ToUpper()
1.28	0.78	0.01	64811	0.15	0.15	TString::operator=(TString const&)

К сожалению, для оптимизации ввода строки из потоков на данный момент не достаточно знаний.

Литература: <https://eax.me/c-cpp-profiling/>

gcov





Утилита gcov предназначена для исследования покрытия кода. Исследуются как строки кода, так и ветви исполнения. Утилита позволяет узнать точное количество исполнения каждой строки кода во время тестирования и принять решение о важности оптимизации наиболее часто исполняемых частей кода либо сокращении объема кода за его ненадобностью.

При использовании gcov в связке с lcov можно получить отчет в более наглядной html форме (См. рисунки ниже).



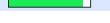
Использование:

```
make
./lab2 < input
gcov -b main.cpp
lcov -t "lab2" -o lab2.info -c -d .
genhtml -o report lab2.info
```

make компилирует программу с дополнительными флагами компиляции `-fprofile-arcs -ftest-coverage` и линковки `-lgcov`.

Directory	Line Coverage ↕	Functions ↕
/home/alex/temp/OA/lab2	 75.7 % 187 / 247	50.0 % 21 / 42
/usr/include/x86_64-linux-gnu/bits	 100.0 % 1 / 1	- 0 / 0
z	 100.0 % 6 / 6	- 0 / 0
z/bits	 41.2 % 7 / 17	33.3 % 1 / 3

Как можно видеть, было задействовано более 75% кода. Это не слишком впечатляющий результат. Заглянем внутрь.

Filename	Line Coverage ↕	Functions ↕
avl.hpp	 100.0 % 109 / 109	100.0 % 14 / 14
custom_string.hpp	 46.2 % 49 / 106	19.2 % 5 / 26
main.cpp	 90.6 % 29 / 32	100.0 % 2 / 2

В модулях avl.hpp и main.cpp задействовано более 90% кода. Программа использовала всего 46% модуля custom_string.hpp потому что он был написан не только для использования во второй лабораторной, но и, возможно, в будущих. Из-за этого многие его функции по просту не используются. Поэтому дальнейшее сокращение кода не целесообразно (в моем понимании).

perf

Perf – утилита для профилирования программ.

К сожалению, не удалось справиться с проблемой деманглинга имен в программе. Как частичное решение можно использовать утилиту `c++filt`, которая может расшифровать имена, но только в режиме отчета, выводящего в стандартный выходной поток. Это не слишком удобно т.к. не получится интерактивно взаимодействовать с `perf`’ом, а следовательно смотреть дизасемблированный код и т.д. и т.п.

С помощью `c++filt` удалось добиться такого результата.

```

# Total Lost Samples: 0
#
# Samples: 13K of event 'cycles'
# Event count (approx.): 13209000000
#
# Overhead  Command  Shared Object      Symbol
# .....  .....  .....
#
13.11% lab2_dbg libstdc++.so.6.0.25 [.] std::basic_istream<char, std::char_traits<char> >::sentry
11.47% lab2_dbg libc-2.27.so [.] _IO_fflush
9.48% lab2_dbg libstdc++.so.6.0.25 [.] std::basic_ostream<char, std::char_traits<char> >::flush
6.23% lab2_dbg libstdc++.so.6.0.25 [.] std::basic_istream<char, std::char_traits<char> >::get()
3.68% lab2_dbg lab2_dbg [.] operator>>(std::basic_istream<char, std::char_traits<char> >
3.57% lab2_dbg libc-2.27.so [.] toupper
2.96% lab2_dbg libc-2.27.so [.] _IO_file_sync@@GLIBC_2.2.5
2.57% lab2_dbg libc-2.27.so [.] _IO_getc
2.09% lab2_dbg lab2_dbg [.] main

```

```

perf record -e cycles -c 1000000 ./lab2_dbg < input
perf report | c++filt | head -n 20

```

Отсюда видно, что показания других профилировщиков частично подтверждаются и тонкие места в работе программы выявлены ровно те же самые.

Heaptrack

Heaptrack – утилита, позволяющая получить информацию о работе программы с памятью. В ее возможности входит профилирование программы по памяти, обнаружение частей кода «транжирящих» память, пиковые значения потребления памяти, обнаружение утечек памяти и многое другое.

Использование:

```

heaptrack ./lab2 < input
heaptrack --analyze "/home/alex/temp/DA/lab2/heaptrack.lab2.15475.gz"

```

либо с графическим интерфейсом

```

heaptrack_gui heaptrack.lab2.15475.gz

```

Данная утилита обладает довольно широким спектром возможностей, и поэтому все данные исследования работы программы с памятью сюда включать не стоит. Это бы слишком загромодило отчет о проделанной работе.

При необходимости возможна демонстрация работы вне отчета.

Как можно будет увидеть основным потребителем памяти является опять же оператор ввода строки из потока. За все время работы программы над тестом им было выделено и освобождено порядка 60 МБ памяти. Всего же за время работы алгоритма было выделено и освобождено около 150 МБ памяти. При этом общее пиковое потребление не превышало 90 КБ. Это говорит о малом потреблении памяти, и о не самой удачной

стратегии ее распределения. Слишком часто работают системные вызовы выделения/возвращения памяти, что негативно сказывается на производительности программы. Но тем не менее удалось добиться полного отсутствия утечек памяти.

Недочёты

Некоторые недочёты остались нерешенными. Неудалось оптимизировать ввод строк из потока из-за недостатка знаний и опыта работы с ними.

Выводы

Описать область применения реализованного алгоритма. Указать типовые задачи, решаемые им. Оценить сложность программирования, кратко описать возникшие проблемы при решении задачи.