Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: И. А. Мариничев
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 21.11.20
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

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.

Результатом лабораторной работы является отчёт, состоящий из:

- 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.
- Сравнение работы исправленной программы с предыдущей версией.
- Общих выводов о выполнении лабораторной работы, полученном опыте. Минимальный набор используемых средств должен содержать утилиту gprof и библиотеку dmalloc, однако их можно заменять на любые другие аналогичные или более развитые утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov).

Используемы утилиты: valgrind (callgrind, kcachegrind), gcov (lcov, genhtml).

1 Описание

В данной лабораторной работе мне необходимо проверить, не содержит ли моя программа из предыдущей лабораторной каких-либо утечек памяти или недостатков, связанных с потреблением оперативной памяти. Даже несмотря на то, что моя программа работает правильно, вполне возможно, что она работает неоптимально. Именно для этого мы воспользуемся утилитами **valgrind (callgrind, kcachegrind)** и **gcov (lcov, genhtml)**.
Хотя в задании к данной работе рекомендовалось использовать библиотеку **dmallloc**, в процессе изучения её возможносте я неоднократно сталкивался с мнением, что её использование не сильно оправдвно, поскольку имеются более мощные альтернативы, в том числе и утилита **valgrind**, поэтому выбор пал именно на неё. [1]
В качетве альтернативы утилите **gprof** я воспользуюсь инструментом **callgrind** для проверки вызовов и затратности тех или иных функций. Также она имеет простой формат графа вызовов, который поддерживается отличными средствами визуализации, например, **kcachegrind**.
В качестве дополнения посмотрим на покрытость моего кода при помощи утилиты **gcov**.
Тест, на котором производилось исследование, состоит из 1000000 строк и включает вставку, удаление, поиск, сохранение и загрузку из файла.

2 Дневник отладки

Для начала протестируем программу с помощью **valgrind**-а, используя при этом ключи –leak-check=full –show-leak-kinds=all –log-file=valgrind-out.txt, которые включают функцию обнаружения утечки, показывают все типы утечек и выводят результат в выходной файл соответственно.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ valgind
--leak-check=full --show-leak-kinds=all --log-file=valgrind-out.txt
./solution <../tests/01.t
```

Посмотрим на результаты:

```
==34== HEAP SUMMARY:
==34==     in use at exit: 122,880 bytes in 6 blocks
==34==   total heap usage: 28 allocs,22 frees,195,724 bytes allocated
==34==
==34== 8,192 bytes in 1 blocks are still reachable in loss record 1 of 6
==34==    at 0x4C3089F: operator new[](unsigned long)
==34==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocate_i
==34==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFB80: std::ios_base::sync_with_stdio(bool)
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== 8,192 bytes in 1 blocks are still reachable in loss record 2 of 6
==34==    at 0x4C3089F: operator new[](unsigned long)
==34==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocate_i
==34==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFBA1: std::ios_base::sync_with_stdio(bool)
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== 8,192 bytes in 1 blocks are still reachable in loss record 3 of 6
==34==    at 0x4C3089F: operator new[](unsigned long)
==34==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocate_i
==34==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFBC2: std::ios_base::sync_with_stdio(bool)
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== 32,768 bytes in 1 blocks are still reachable in loss record 4 of 6
==34==    at 0x4C3089F: operator new[](unsigned long)
==34==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_allo
```

```
==34==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFC37: std::ios_base::sync_with_stdio(bool)
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== 32,768 bytes in 1 blocks are still reachable in loss record 5 of 6
==34==    at 0x4C3089F: operator new[](unsigned long)
==34==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_allo
==34==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFC51: std::ios_base::sync_with_stdio(bool) )
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== 32,768 bytes in 1 blocks are still reachable in loss record 6 of 6
==34==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_
==34==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_allo
==34==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==34==    by 0x4EDFC6B: std::ios_base::sync_with_stdio(bool)
==34==    by 0x109AEA: main (2-1.cpp:9)
==34==
==34== LEAK SUMMARY:
==34==    definitely lost: 0 bytes in 0 blocks
==34==    indirectly lost: 0 bytes in 0 blocks
==34==      possibly lost: 0 bytes in 0 blocks
==34==    still reachable: 122,880 bytes in 6 blocks
==34==         suppressed: 0 bytes in 0 blocks
==34==
==34== For counts of detected and suppressed errors,rerun with: -v
==34== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

В результате поиска ошибки оказалось, что данная строчка:

```
1 || std::ios::sync_with_stdio(false);
```

работающая с синхронизицией потоков ввода и вывода C/C++ оставляет все еще достижимую область памяти. И это действительно так, ведь стандартные потоки "никогда"не уничтожаются. "Утечка"появляется просто потому, что синхронизированный буффер по умолчанию не выделяет динамическую память. [2]
Теперь посмотрим на количесво вызовов, затратность тех или иных функций. Для этого воспользуемся **callgrind**-ом. Скомпилируем нашу программу с ключом -g3 для того, чтобы наблюдать исходный код в нашем средстве визуализации **kcachegrind**-е. **Kcachegrind** покажет нам граф вызовов функций и их связь с другими элементами нашей программы. Также можно будет увидеть наиболее вызываемые функции и многое другое.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ valgrind
--tool=callgrind ./solution <../tests/01.t
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ kcachegrind
callgrind.out.591
```

Посмотрим на визуализацию нашего отчета:

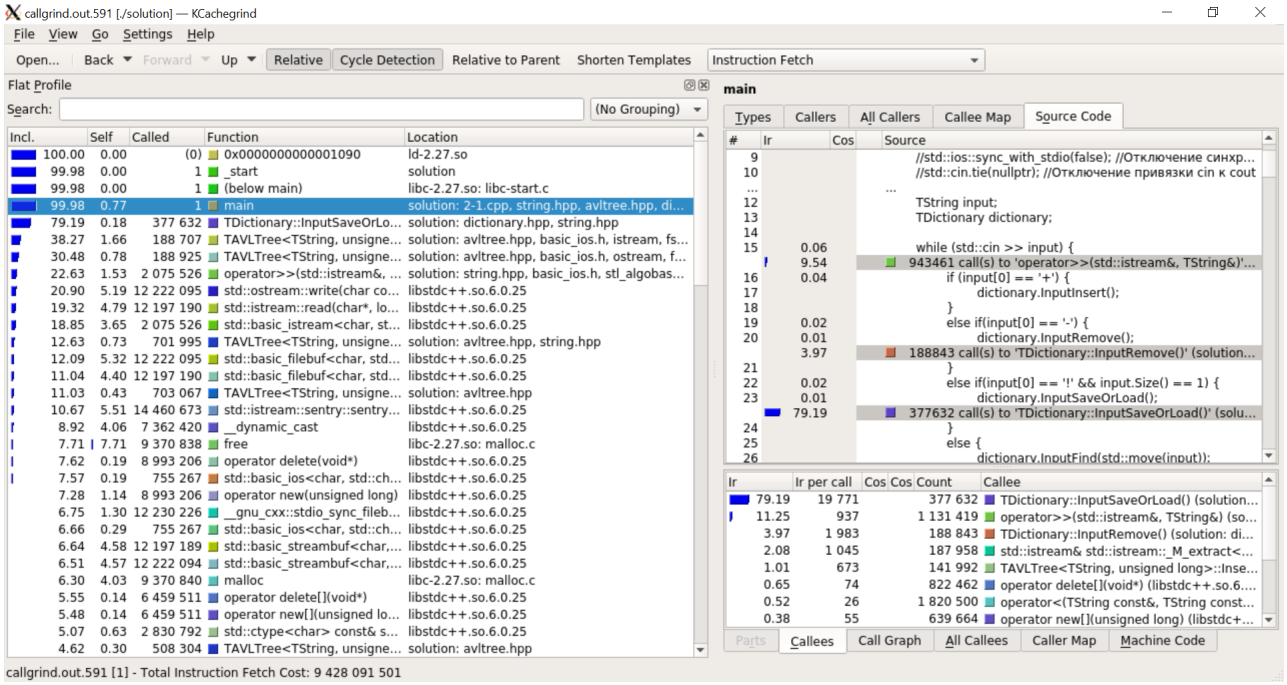

Рис.1

Слева стоит обращать внимание на те строчки, в которых функции находятся в наших исходных файлах, остальное же нас особо не интересует. Справа сверху мы можем наблюдать исходный код с вызовами, а чуть ниже вызывающие элемнты, в данном случае для функции main.
Теперь посмотрим на наглядное изображение графа вызовов, например, для функции InputRemove, которая отвечает за удаление элементов из словаря.

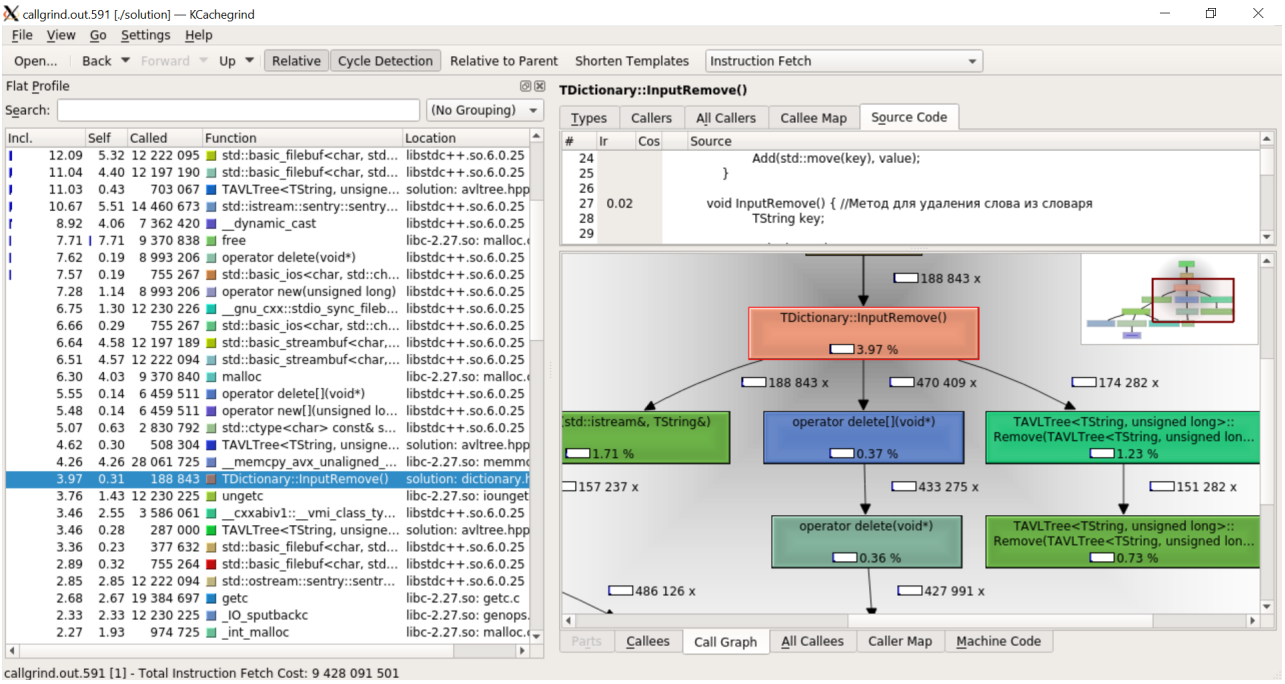

Рис.2

И напоследок посмотрим на покрытость моего кода, то есть процент задействованности строчек кода. Для этого воспользуемся утилитой **gcov**. Скомпилируем наш файл 2-1.cpp специальным образом и посмотрим на результат.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ g++ --coverage
2-1.cpp -o main
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ ./main
<../tests/01.t
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ lcov -t
"mainrep" -o mainrep.info -c -d .
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab2/profiling$ genhtml
-o report mainrep.info
```

Из отчета сгенерированного при помощи утилиты **lcov** можно увидеть, что процент покрытости моего кода достаточно высок и лишь моя реализация строкового типа имеет незадействованные строки, что было вполне ожидаемо.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - profiling	Lines:	274	296	92.6 %
Test:	mainrep.info	Functions:	52	57	91.2 %
Date:	2020-11-20 15:57:27				

Filename	Line Coverage		Functions	
2-1.cpp	100.0 %	13 / 13	100.0 %	3 / 3
avltree.hpp	96.4 %	161 / 167	100.0 %	26 / 26
dictionary.hpp	97.0 %	32 / 33	100.0 %	7 / 7
string.hpp	81.9 %	68 / 83	76.2 %	16 / 21

Generated by: LCOV version 1.13

Рис.3

```
  108              :         }
  109              :
  110            0 :         constIterator begin() const {
  111            0 :                 return storage;
  112              :         }
  113              :
  114              :         iterator end() { //Завершение итерирования по строке
  115              :                 if (storage != nullptr) {
  116              :                         return storage + alreadyUsed;
  117              :                 }
  118              :                 return nullptr;
  119              :         }
  120              :
  121            0 :         constIterator end() const {
  122            0 :                 if (storage != nullptr) {
  123            0 :                         return storage + alreadyUsed;
  124              :                 }
  125            0 :                 return nullptr;
  126              :         }
  127              :
  128      1345616 :         const valueType *Cstr() const { //Метод для нахождения размера строки
  129      1345616 :                 return storage;
  130              :         }
  131              :
  132      5234591 :         int Size() const {
  133      5234591 :                 return alreadyUsed;
  134              :         }
  135              :
  136              :         TString &operator+=(const valueType &ch) { //Перегрузка оператора +=
  137              :                 PushBack(ch);
  138              :                 return *this;
  139              :         }
  140              :
  141      7417122 :         const valueType &At(int index) const { //Получение символа в строке
  142      7417122 :                 if (index < 0 || index > alreadyUsed) {
  143            0 :                         throw std::out_of_range("You are doing this wrong!");
  144              :                 }
  145              :
  146      7417122 :                 return storage[index];
```

Рис.4

3 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я научился работать с утилитами **valgrind (callgrind, kcachegrind)** и **gcov (lcov, genhtml)**.
Valgrind - очень простая и мощная утилита, позволяющая существенно ускорить отладку и профилирование прграмм. Она позволяет находить различные утечки памяти, что очень полезно для языков, работающих с указателями. Я обязательно буду пользоваться ей и в дальнейшем при профилировании и отладке программ.
Сегодня требования к производительности ПО очень разные, но вероятно не секрет, что многие приложения имеют очень жесткие требования по скорости выполнения. В общем случае для всех приложений, кроме простейших, работает правило: чем выше призводительность, тем полезнее и популярнее будет приложение. По этой причине анализ производительности является (или должен являться) важнейшей задачей для многих прикладных разработчиков. Анализ производительности - очень трудоемкий процесс, поэтому были созданы инструменты, которые способны упростить его.

Список литературы

[1] *Работа с dmalloc*
URL: `http://alexott.net/ru/linux/valgrind/DMalloc.html` (дата обращения: 17.11.2020).

[2] *Valgrind Frequently Asked Questions*
URL: `https://www.valgrind.org/docs/manual/faq.html#faq.deflost` (дата обращения: 15.11.2020).