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: А. К. Киреев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19

Дата:
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 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.

Минимальный набор используемых средст должен содержать утилиту gprof и библиотеку dmalloc, однако их можно заменять на любые другие аналогичные или более известные утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov).

1 Описание

Для выполнения данной лабораторной работы я буду использовать свою рабочую реализацию красно-чёрного дерева. Из инструментов для анализа программ я выбрал **Valgrind** и **gprof**.

Valgrind — инструментальное программное обеспечение, предназначенное для динамического анализа и отладки использования памяти, обнаружения утечек памяти. Просто необходимая вещь во время работы с памятью процесса. Язык C++ не имеет своего сборщика мусора, с памятью в нём приходится работать вручную, что зачастую порождает множество ошибок из-за невнимательности программиста связанных с памятью. Основным типом ошибок являются утечки памяти, то есть когда память была выделена и не может быть впоследствии освобождена, потому что программа больше не имеет указателей на выделенный блок памяти, что может привести к тому, что процессу не хватит выделенной ему памяти, произойдёт истощение кучи. **Valgrind** помогает обнаружить обращения и работу с неинициализированными участками памяти, помогает найти ошибки связанные с неправильным освобождением памяти, а также показывает расход памяти программы. Стоит отметить, что при запуске **Valgrind** сильно падает производительность программы.

gprof — инструмент для профилирования программ. Профилирование позволяет вам изучить, где ваша программа расходует свое время и какие функции вызывали другие функции, пока программа исполнялась. Эта информация может указать вам на ту часть программы, которая исполняется медленнее, чем вы ожидали, а также в этой информации содержится то, сколько раз вызывались те или иные функции в коде. Вся эта информация поможет вам написать более оптимизированный и качественный код. При запуске **gprof** вам будет показана таблица, в которой можно будет найти всю полезную информацию: общий итог времени, затраченного на исполнение каждой функции вашей программы; процент от общего времени исполнения вашей программы, затраченный на выполнение функций; количество вызовов функции и т.д.

2 Исходный код

Полный исходный код для красно-чёрного дерева был представлен в отчете к ЛР2. Давайте проверим наш код в **Valgrind** на наличие ошибок при работе с памятью.

```
kak@MacBookAir-K:~/Рабочий стол/DA/lab2/solution valgrind ./solution <../tests/03.t
>/dev/null
==10371== Memcheck,a memory error detector
==10371== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==10371== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==10371== Command: ./solution
==10371==
==10371==
==10371== HEAP SUMMARY:
==10371==     in use at exit: 122,880 bytes in 6 blocks
==10371==   total heap usage: 28,237 allocs,28,231 frees,8,777,208 bytes allocated
==10371==
==10371== LEAK SUMMARY:
==10371==    definitely lost: 0 bytes in 0 blocks
==10371==    indirectly lost: 0 bytes in 0 blocks
==10371==      possibly lost: 0 bytes in 0 blocks
==10371==    still reachable: 122,880 bytes in 6 blocks
==10371==         suppressed: 0 bytes in 0 blocks
==10371== Rerun with --leak-check=full to see details of leaked memory
==10371==
==10371== For lists of detected and suppressed errors,rerun with: -s
==10371== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как мы видим, никаких ошибок выявлено не было. Давайте намеренно сделаем одну ошибку: закомментируем *delete* вершины в методе *Remove*, что приведет к утечке памяти; и попытаемся найти и исправить её с помощью **Valgrind**.

Листинг 1: rb_tree.cpp

```
1   void TRBTree::Remove(TRBTreeNode* node) {
2       TRBTreeNode* toDelete = node;
3       TColor toDeleteColor = toDelete->Color;
4       TRBTreeNode* toReplace;
5       TRBTreeNode* toReplaceParent;
6       if (node->Left == NULL) {
7           //...
8       } else {
9           //...
10      }
```

```
11          if (toDeleteColor == TColor::Black) {
12              RemoveFixUp(toReplace, toReplaceParent);
13          }
14          // FORGET TO FREE MEMORY HERE -> delete node;
15      }
```

Сообщение от **Valgrind** на небольшом тесте, где мы удаляем лишь одну вершину.

```
kak@MacBookAir-K:~/Рабочий стол/DA/lab2/solution valgrind --leak-check=full
./solution
==10549== Memcheck,a memory error detector
==10549== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==10549== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==10549== Command: ./solution
==10549==
+ a 1
+ b 2
a
-a
a
OK
OK
OK: 1
OK
NoSuchWord
==10549==
==10549== HEAP SUMMARY:
==10549==     in use at exit: 123,184 bytes in 7 blocks
==10549==   total heap usage: 10 allocs,3 frees,196,200 bytes allocated
==10549==
==10549== 304 bytes in 1 blocks are definitely lost in loss record 1 of 7
==10549==    at 0x483CE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/v
==10549==    by 0x10AEF1: NRBTree::TRBTree::Insert(NPair::TPair const&) (in
/home/kak/Рабочий стол/DA/lab2/solution/solution)
==10549==    by 0x10CA70: RequestHandler() (in /home/kak/Рабочий стол/DA/lab2/solution
==10549==    by 0x10A7EE: main (in /home/kak/Рабочий стол/DA/lab2/solution/solution)
==10549==
==10549== LEAK SUMMARY:
==10549==    definitely lost: 304 bytes in 1 blocks
==10549==    indirectly lost: 0 bytes in 0 blocks
==10549==      possibly lost: 0 bytes in 0 blocks
==10549==    still reachable: 122,880 bytes in 6 blocks
==10549==         suppressed: 0 bytes in 0 blocks
```

```
==10549== Reachable blocks (those to which a pointer was found) are not shown.
==10549== To see them,rerun with: --leak-check=full --show-leak-kinds=all
==10549==
==10549== For lists of detected and suppressed errors,rerun with: -s
==10549== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Ключ –leak-check=full выводит нам подробную информацию о каждой утечке. Из сообщения мы видим, что та память, что выделялась в *Insert* при помощи *new* не удаляется. Именно это приводит к утечке. Значит где-то мы забываем освободить эту память или теряем на неё указатель и физически не можем получить уже доступ. Первый вариант невозможет, так как дерево корректо очищается рекурсивно в самом конце программы. Значит при удалении мы лишь отвязываем узел и теряем его насовсем. Находим место, где мы должны писать *delete*, добавляем эту команду и проверяем снова.

```
kak@MacBookAir-K:~/Рабочий стол/DA/lab2/solutionvalgrind ./solution
==10801== Memcheck,a memory error detector
==10801== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==10801== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==10801== Command: ./solution
==10801==
+ a 1
+ b 2
a
-a
a
OK
OK
OK: 1
OK
NoSuchWord
==10801==
==10801== HEAP SUMMARY:
==10801==     in use at exit: 122,880 bytes in 6 blocks
==10801==   total heap usage: 10 allocs,4 frees,196,200 bytes allocated
==10801==
==10801== LEAK SUMMARY:
==10801==    definitely lost: 0 bytes in 0 blocks
==10801==    indirectly lost: 0 bytes in 0 blocks
==10801==      possibly lost: 0 bytes in 0 blocks
==10801==    still reachable: 122,880 bytes in 6 blocks
==10801==         suppressed: 0 bytes in 0 blocks
```

```
==10801== Rerun with --leak-check=full to see details of leaked memory
==10801==
==10801== For lists of detected and suppressed errors,rerun with: -s
==10801== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Мы починили ошибку, наш код корректно работает.
Теперь изучим наш код на оптимизированность и быстродействие, используя профилировщик **gprof**.

```
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
33.37      0.04     0.04    49973     0.00     0.00  NPair::TPair::TPair(char*,unsigne
long long)
16.69      0.06     0.02    28229     0.00     0.00  NPair::TPair::TPair(NPair::TPair
const&)
16.69      0.08     0.02    25064     0.00     0.00  NRBTree::TRBTree::Search(char*,NI
8.34      0.09     0.01    49972     0.00     0.00  NRBTree::TRBTree::Insert(NPair::TI
const&,NRBTree::TRBTreeNode*)
8.34      0.10     0.01    18043     0.00     0.00  NPair::TPair::operator=(NPair::TPa
const&)
8.34      0.11     0.01    17933     0.00     0.00  NRBTree::TRBTree::Remove(NRBTree:
8.34      0.12     0.01     9971     0.00     0.00  NRBTree::TRBTree::RemoveFixUp(NRBT
0.00      0.12     0.00   100000     0.00     0.00  StrToLower(char*)
0.00      0.12     0.00    49973     0.00     0.00  NRBTree::TRBTree::Insert(NPair::TI
const&)
0.00      0.12     0.00    28229     0.00     0.00  NRBTree::TRBTreeNode::TRBTreeNode
const&)
0.00      0.12     0.00    25064     0.00     0.00  NRBTree::TRBTree::Search(char*,NPa
0.00      0.12     0.00    24963     0.00     0.00  NRBTree::TRBTree::Remove(char
const*)
0.00      0.12     0.00    14068     0.00     0.00  NRBTree::TRBTree::Recolor(NRBTree
0.00      0.12     0.00    10677     0.00     0.00  NRBTree::TRBTree::LeftRotate(NRBTr
0.00      0.12     0.00     9371     0.00     0.00  NRBTree::TRBTree::RightRotate(NRBT
0.00      0.12     0.00        1     0.00     0.00  RequestHandler()
0.00      0.12     0.00        1     0.00     0.00  NRBTree::TRBTree::DeleteTree(NRBTr
0.00      0.12     0.00        1     0.00     0.00  NRBTree::TRBTree::TRBTree()
0.00      0.12     0.00        1     0.00     0.00  NRBTree::TRBTree::~TRBTree()
```

Как мы видим, большинство времени работы занимают конструкторы пар. Давайте поймём почему. Посмотрим на таблицу ниже, которая говорит о вызовах функций и о том, сколько работала каждая из них вместе с функциями, которые она вызывала.

```
index % time    self  children    called     name
<spontaneous>
[1]    100.0    0.00    0.14                 main [1]
0.00    0.14       1/1           RequestHandler() [2]
-----------------------------------------------
0.00    0.14       1/1           main [1]
[2]    100.0    0.00    0.14       1         RequestHandler() [2]
0.06    0.00   49932/49932       NPair::TPair::TPair(char*,unsigned long long)
[3]
0.00    0.05   49932/49932       NRBTree::TRBTree::Insert(NPair::TPair
......
```

Заметим, что дольше всего (помимо функции $main$ и обработчика, которые работают всегда) работает функция вставки $Insert$. Увидим, что мы туда передаём пару типа $NPair :: TPair$, которая создается при вызове функции из обработчика. Это приводит к лишнему копированию: при передачи объекта в функцию вызывается конструктор, который копирует строку и значение, а также в самой $Insert$ при создании вершины вызывается конструктор копирования, который так же копирует строку.
Теперь немного оптимизируем код, передавая в функцию вставки не сам объект типа $NPair :: TPair$, а лишь указатель на строку и значение, чтобы избежать чрезмерного копирования строк при создании временной пары, которая передается в $Insert$ из $RequestHandler$. Снова запустим **gprof** и убедимся, что он стал показывать прирост производительности.

```
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
25.03      0.02     0.02    49972     0.00     0.00  NRBTree::TRBTree::Insert(char*,u
long long,NRBTree::TRBTreeNode*)
25.03      0.04     0.02    28229     0.00     0.00  NPair::TPair::TPair(char*,unsigne
long long)
12.52      0.05     0.01    49973     0.00     0.00  NRBTree::TRBTree::Insert(char*,u
long long)
12.52      0.06     0.01    25064     0.00     0.00  NRBTree::TRBTree::Search(char*,N
12.52      0.07     0.01    24963     0.00     0.00  NRBTree::TRBTree::Remove(char
const*)
12.52      0.08     0.01    18043     0.00     0.00  NPair::TPair::operator=(NPair::T
const&)
0.00      0.08     0.00   100000     0.00     0.00  StrToLower(char*)
0.00      0.08     0.00    28229     0.00     0.00  NRBTree::TRBTreeNode::TRBTreeNode
long long)
```

```
0.00      0.08      0.00     25064      0.00      0.00  NRBTree::TRBTree::Search(char*,NPa
0.00      0.08      0.00     17933      0.00      0.00  NRBTree::TRBTree::Remove(NRBTree:
0.00      0.08      0.00     14068      0.00      0.00  NRBTree::TRBTree::Recolor(NRBTree
0.00      0.08      0.00     10677      0.00      0.00  NRBTree::TRBTree::LeftRotate(NRBTr
0.00      0.08      0.00      9971      0.00      0.00  NRBTree::TRBTree::RemoveFixUp(NRBT
0.00      0.08      0.00      9371      0.00      0.00  NRBTree::TRBTree::RightRotate(NRBT
0.00      0.08      0.00         1      0.00      0.00  RequestHandler()
0.00      0.08      0.00         1      0.00      0.00  NRBTree::TRBTree::DeleteTree(NRBTr
0.00      0.08      0.00         1      0.00      0.00  NRBTree::TRBTree::TRBTree()
0.00      0.08      0.00         1      0.00      0.00  NRBTree::TRBTree::~TRBTree()
```

3 Тест производительности

Сверим теперь две реализации при помощи бенчмарка.

```
MacBook-Air-K:support_copy AK$ ./b1 <../tests/03.t
==============START============
INSERT rb tree time: 150624ms
================================
DELETE rb tree time: 18540ms
================================
SEARCH rb tree time: 30643ms
==============END==============
MacBook-Air-K:support_copy AK$ ./b2 <../tests/03.t
==============START============
FIXED INSERT rb tree time: 78788ms
================================
FIXED DELETE rb tree time: 17422ms
================================
FIXED SEARCH rb tree time: 30006ms
==============END==============
```

Мы видим, что на большом тесте время работы вставки в дерево уменьшилось почти вдвое из-за отсутствия лишнего копирования строк. Это очень хороший результат.

4 Выводы

Выполнив данную лабораторную работу, я лучше познакомился с такими инструментами, как **Valgrind** и **gprof**. Они незаменимы и очень полезны при разработке больших проектов, при поиске ошибок, а в особенности при оптимизации программ по времени работы и по памяти. Также в ходе данной лабораторной работы я стал лучше разбираться в том, какие ошибки при работе с памятью в C++ мне могут встретиться, понял, как их избегать.

Список литературы

[1] *Википедия — Valgrind*
URL: `https://ru.wikipedia.org/wiki/Valgrind` (дата обращения: 05.11.2020).

[2] *OpenNET — Профилятор gprof.*
URL: `https://ru.wikipedia.org/wiki/Valgrind` (дата обращения: 05.11.2020).

[3] *Valgrind.*
URL: `https://www.valgrind.org` (дата обращения: 05.11.2020).