Лабораторная работа № 3 по курсу дискретного анализа: инструментирование и профилирование.

Выполнил студент группы 08-208 МАИ *Коростелев Дмитрий Васильевич.*

Введение

Для отладки различных приложений программисту крайне часто приходится пользоваться дополнительными программами или утилитами, которые позволяют находить логические ошибки или баги. Но, помимо этого, при разработке программ, написанных на языках, где программист может собственноручно оперировать памятью вычислительной машины требуется следить, чтобы при отработке определённых блоков программы выполнялось освобождение ранее использованной памяти. Кроме того, есть потребность в получении данных о производительности программы.

Задание

Реализовать Абстрактный тип данных под названием б-дерево, которое поддерживает следующие операции – вставка, удаление, поиск, сохранение и загрузка в файл.

Метод решения

Для реализации данного типа данных для начала требуется ознакомится с самой структурой. Все реализации методов на данным деревом были взяты из книги «Алгоритмы и структуры данных» под авторством Кормена, Лейзерсона, Ривеста и Штайна. Последовательно реализуем алгоритмы поиска ключа, объединения двух дочерних узлов, вставки в дерево, далее удаление с присущими ему операциями. Далее требуется провести тестирование программы, проверить линейность основных операции а также наличие утечек памяти. Для этого будем использовать утилиты valgrind, address sanitizer, gprof.

Отладка и проверка программы.

В ходе выполнения всей лабораторной работы я столкнулся с рядом проблем, вызванных выходом за пределы массива или утечек памяти, однако большинство из них решались довольно просто и не требовали использования дополнительных утилит. Но не обошлось и без довольно серьезных проблем, из-за которых мне пришлось использовать средства отладки и поиска ошибок. Первая проблема появилась на самом раннем этапе моего написания данной программы.

```
valgrind ./a.out
==2268== Memcheck, a memory error detector
```

```
==2268== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
et al.
==2268== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
copyright info
==2268== Command: ./a.out
==2268==
+ abababa 123
OK
==2268==
==2268== HEAP SUMMARY:
==2268==     in use at exit: 122,880 bytes in 6 blocks
==2268==   total heap usage: 10 allocs, 4 frees, 222,552 bytes
allocated
==2268==
==2268== LEAK SUMMARY:
==2268==    definitely lost: 0 bytes in 0 blocks
==2268==    indirectly lost: 0 bytes in 0 blocks
==2268==      possibly lost: 0 bytes in 0 blocks
==2268==    still reachable: 122,880 bytes in 6 blocks
==2268==         suppressed: 0 bytes in 0 blocks
==2268== Rerun with --leak-check=full to see details of leaked memory
==2268==
==2268== For counts of detected and suppressed errors, rerun with: -v
==2268== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Как мы видим по итогу выполнения программы всегда остается доступная для использования память. Для устранения этой проблемы попросим valgrind вывести более подробную информацию с помощью ключей –leak-check=full –show-leak-kinds=all. Стало очевидным, что данная утечка возникает при отключении синхронизации между двумя потоками.

Следующая утечка оказалась более критичной. При тестировании своего дерева, при добавлении в него нескольких ключей и после выхода без последующего удаления происходила огромная утечка памяти.

```
valgrind ./a.out
==2479== Memcheck, a memory error detector
==2479== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
et al.
==2479== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
copyright info
==2479== Command: ./a.out
==2479==
```

```
==2479==
==2479== Process terminating with default action of signal 27
(SIGPROF)
==2479==    at 0x54ECDAF: __open_nocancel (open64.c:69)
==2479==    by 0x550091F: write_gmon (gmon.c:370)
==2479==    by 0x55010DA: _mcleanup (gmon.c:444)
==2479==    by 0x5420614: __cxa_finalize (cxa_finalize.c:83)
==2479==    by 0x108D92: ??? (in /home/dmitry/ADTaA/Abstract_data_
types_and_Algorithms/DataTypes/B-Tree/a.out)
==2479==    by 0x4010B72: _dl_fini (dl-fini.c:138)
==2479==    by 0x5420040: __run_exit_handlers (exit.c:108)
==2479==    by 0x5420139: exit (exit.c:139)
==2479==    by 0x53FEB9D: (below main) (libc-start.c:344)
==2479==
==2479== HEAP SUMMARY:
==2479==     in use at exit: 332,526 bytes in 2,504 blocks
==2479==   total heap usage: 7,499 allocs, 4,995 frees, 575,026
bytes allocated
==2479==
==2479== LEAK SUMMARY:
==2479==    definitely lost: 199,760 bytes in 2,497 blocks
==2479==    indirectly lost: 0 bytes in 0 blocks
==2479==      possibly lost: 0 bytes in 0 blocks
==2479==    still reachable: 132,766 bytes in 7 blocks
==2479==         suppressed: 0 bytes in 0 blocks
==2479== Rerun with --leak-check=full to see details of leaked memory
==2479==
==2479== For counts of detected and suppressed errors, rerun with: -v
==2479== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
Profiling timer expired
```

При помощи средств valgrind удалось обнаружить данную утечку. Однако на этапе тестирования стало понятно, что сам фреймворк valgrind сильно тормозит работу моей программы из-за чего стало неудобным проводить дальнейшие исправления.

Для устранения данной утечки я использовал утилиту ASAN, которая использует немного иной при сборе информации во время работы программы. Address Sanitizer довольно прост в использовании и подключается при помощи ключа -fsanitize=address.

Сама ошибка заключалась в том, что при создании нового ключа выделяется память под указатели, в которых хранятся дочерние узлы, но во время вызова деструктора, данный массив не очищался, что и приводило к утечке.

```
./a.out
```

```
=================================================================
==2490==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 91920 byte(s) in 1149 object(s) allocated from:
    #0 0x7fd2a6e0f618 in operator new[](unsigned long)
    (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe0618)
    #1 0x560c3b22bf37 in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeNode()
    (/home/dmitry/ADTaA/Abstract_data_types_and_Algorithms/DataTypes/
    B-Tree/a.out+0x1f37)
    #2 0x560c3b22c2c6 in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeSplitChild(BTree<int, 5>::BTreeNode<int, 5>*, int)
    (/home/dmitry/ADTaA/Abstract_data_types_and_Algorithms/DataTypes/
    B-Tree/a.out+0x22c6)
    #3 0x560c3b22cd8b in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeInsertNonfull(int) (/home/dmitry/ADTaA/
    Abstract_data_types_and_Algorithms/DataTypes/B-Tree/a.out+0x2d8b)
    #4 0x560c3b22ce72 in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeInsertNonfull(int) (/home/dmitry/ADTaA/
    Abstract_data_types_and_Algorithms/DataTypes/B-Tree/a.out+0x2e72)
    #5 0x560c3b22ce72 in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeInsertNonfull(int) (/home/dmitry/ADTaA/
    Abstract_data_types_and_Algorithms/DataTypes/B-Tree/a.out+0x2e72)
    #6 0x560c3b22ce72 in BTree<int, 5>::BTreeNode<int, 5>
    ::BTreeInsertNonfull(int) (/home/dmitry/ADTaA/
    Abstract_data_types_and_Algorithms/DataTypes/B-Tree/a.out+0x2e72)
```

Следующим шагом была проверка линейности моего Б-дерева. Чтобы удостоверится в этом я использовал gprof – утилита, которая позволяет отслеживать время работы программы, ее компонентов, а так же кол-во их вызовов.

```
dmitry@dmitry-VirtualBox:~/ADTaA/Abstract_data_types_and_Algorithms/
DataTypes/B-Tree$ gprof a.out gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    10000     0.00     0.00  BTree<int, 5>
  ::BTReeInsert(int)
```

```
  0.00      0.00      0.00    10000      0.00      0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeInsertNonfull(int)
  0.00      0.00      0.00     2497      0.00      0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeNode()
  0.00      0.00      0.00     2497      0.00      0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::~BTreeNode()
  0.00      0.00      0.00     2491      0.00      0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeSplitChild(BTree<int, 5>::BTreeNode<int, 5>*, i
  0.00      0.00      0.00        1      0.00      0.00  _GLOBAL__sub_
  I__ZgtRK8KeyValueS1_
  0.00      0.00      0.00        1      0.00      0.00  __static_
  initialization_and_destruction_0(int, int)
  0.00      0.00      0.00        1      0.00      0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeDestroy()
  0.00      0.00      0.00        1      0.00      0.00  BTree<int, 5>
  ::BTree()
  0.00      0.00      0.00        1      0.00      0.00  BTree<int, 5>
  ::~BTree()
```

Как видно из результатов тестирования вставка в дерево работает крайне быстро, это связано с тем, что при вставке в б-дерево совершается очень мало переходов от одного узла к другому, в тоже время многие операции проводимые внутри дерева являются атомарными.

Далее проведем тест удаления

```
dmitry@dmitry-VirtualBox:~/ADTaA/Abstract_data_types_and_Algorithms/
DataTypes/B-Tree$ gprof a.out gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    50461     0.00     0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeFindKey(int)
  0.00      0.00     0.00    10000     0.00     0.00  BTree<int, 5>
  ::BTReeInsert(int)
  0.00      0.00     0.00    10000     0.00     0.00  BTree<int, 5>
  ::BTreeDeleteNode(int)
  0.00      0.00     0.00    10000     0.00     0.00  BTree<int, 5>
  ::BTreeNode<int, 5>::BTreeDeleteNode(int)
  0.00      0.00     0.00    10000     0.00     0.00  BTree<int, 5>
```

```
::BTreeNode<int, 5>::BTreeInsertNonfull(int)
0.00      0.00      0.00    10000      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeDeleteFromLeaf(int)
0.00      0.00      0.00     2505      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::Fill(int)
0.00      0.00      0.00     2497      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeNode()
0.00      0.00      0.00     2497      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::~BTreeNode()
0.00      0.00      0.00     2491      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeMerge(int)
0.00      0.00      0.00     2491      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeSplitChild(BTree<int, 5>::BTreeNode<int, 5>*
, int)
0.00      0.00      0.00       14      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeBorrowFromNext(int)
0.00      0.00      0.00        1      0.00      0.00  _GLOBAL__sub_
I__ZgtRK8KeyValueS1_
0.00      0.00      0.00        1      0.00      0.00  __static_
initialization_and_destruction_0(int, int)
0.00      0.00      0.00        1      0.00      0.00  BTree<int, 5>
::BTree()
0.00      0.00      0.00        1      0.00      0.00  BTree<int, 5>
::~BTree()
```

Как видно, удаление – более емкий и сложный процесс, однако даже он занимает не так много времени (с учетом кол-во вставляемых в дерево узлов, которое в данном тесте было 10000). Главное, что меня насторожило – кол-во вызовов функции FindKey, которое сильно больше чем кол-во узлов. Проведем еще один тест

```
dmitry@dmitry-VirtualBox:~/ADTaA/Abstract_data_types_and_Algorithms/
DataTypes/B-Tree$ gprof a.out gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 50.03      0.01     0.01   652187    15.34    15.34  BTree<int, 5>
 ::BTreeNode<int, 5>::BTreeFindKey(int)
 50.03      0.02     0.01    24988   400.47   400.47  BTree<int, 5>
 ::BTreeNode<int, 5>::BTreeMerge(int)
  0.00      0.02     0.00   100000     0.00     0.00  BTree<int, 5>
  ::BTReeInsert(int)
```

```
0.00      0.02      0.00    100000      0.00    200.14  BTree<int, 5>
::BTreeDeleteNode(int)
0.00      0.02      0.00    100000      0.00    200.14  BTree<int, 5>
::BTreeNode<int, 5>::BTreeDeleteNode(int)
0.00      0.02      0.00    100000      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeInsertNonfull(int)
0.00      0.02      0.00    100000      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeDeleteFromLeaf(int)
0.00      0.02      0.00     25007      0.00    400.16  BTree<int, 5>
::BTreeNode<int, 5>::Fill(int)
0.00      0.02      0.00     24995      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeNode()
0.00      0.02      0.00     24995      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::~BTreeNode()
0.00      0.02      0.00     24988      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeSplitChild(BTree<int, 5>::BTreeNode<int, 5>*,
int)
0.00      0.02      0.00        19      0.00      0.00  BTree<int, 5>
::BTreeNode<int, 5>::BTreeBorrowFromNext(int)
0.00      0.02      0.00         1      0.00      0.00  _GLOBAL__sub_
I__ZgtRK8KeyValueS1_
0.00      0.02      0.00         1      0.00      0.00  __static_
initialization_and_destruction_0(int, int)
0.00      0.02      0.00         1      0.00      0.00  BTree<int, 5>
::BTree()
0.00      0.02      0.00         1      0.00      0.00  BTree<int, 5>
::~BTree()
```

Видно, что при увеличении кол-ва тестов в 10 раз, кол-во вызовов функции findkey увеличилось примерно в 10 раз, что говорит о линейности данной операции.

Вывод

Благодаря данной лбораторной работе, на собственном опыте смог разработать свою структуру данных, провести анализ ее свойств, оценить и улучшить результаты ее выполнения, протестировать и отладить программу с помощью вспомогательных средств инструментирования и профилирования.