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. О. Ильин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №1

Задача: Необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить.

Минимальный набор используемых средств 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

Результатом лабораторной работы является отчёт, состоящий из:

1. Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
2. Выводов о найденных недочётах.
3. Сравнение работы исправленной программы с предыдущей версией.
4. Общих выводов о выполнении лабораторной работы, полученном опыте.

2 Дневник выполнения работы

Основные этапы создания программы:

1. Реализация первоначальной версии программы
2. Поиск логических ошибок и их устранение
3. Выявление утечек памяти и их устранение
4. Тестирование производительности
5. Оптимизация неэффективных участков кода

3 Используемые средства

1 LLDB

В процессе тестирования первой версии программы были обнаружены множественные ошибки в реализациях функций AVL-дерева, например, в функциях поворота, добавления. Использование дебаггера облегчило поиск ошибок, поскольку позволяло поэтапно проверять каждый вызов той или иной функции, а также производить остановку программы в потенциально ошибочных частях кода. Был использовал дебаггер LLDB, который является частью проекта LLVM. Имеет все необходимые для дебаггера команды: *b (break), c (continue), s (step in), n (next), bt (backtrace)*. Использовался как в консольном режиме, так и в интегрированной среде Xcode. Стоит отметить, что использования в консольном режиме, необходимо скопилировать программу с ключом *-g*.

2 Valgrind

Valgrind – фреймворк для динамического бинарного инструментирования. Хорошо известен как мощное средство поиска ошибок работы с памятью. Он состоит из ядра (core) и утилит на основе этого ядра: Примеры утилит:

- memcheck – основной модуль, обеспечивающий обнаружение утечек памяти, и прочих ошибок, связанных с неправильной работой с областями памяти.
- cachegrind – анализирует выполнение кода, собирая данные о (не)попаданиях в кэш, и точках перехода (когда процессор неправильно предсказывает ветвление). Эта статистика собирается для всей программы, отдельных функций и строк кода.
- callgrind – анализирует вызовы функций и позволяет построить дерево вызовов функций.
- massif – позволяет проанализировать выделение памяти различными частями программы.
- helfrind – анализирует выполняемый код на наличие различных ошибок синхронизации, при использовании многопоточного кода, использующего POSIX Threads.

Для его использования необходимо скомпилировать программу с ключом *-g* и вызвать valgrind от исполняемого файла. Эта утилита не только сообщит о наличии утечек, но и покажет, где эта память была выделена. Использование дополнительных ключей, таких как *--leak-check* и других позволяет настраивать количество выводимой информации об утечках.

Например, рассмотрим посылку номер 1944. При вызове valgrind получаем следующее:

```
parallels@parallels:/media/psf/Home/Documents/Labs/da/da_lab_03/solutions/1944$
valgrind --leak-check=full --show-leak-kinds=all ./solution  >/dev/null
==12025== Memcheck,a memory error detector
==12025== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==12025== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12025== Command: ./solution
==12025==
+ abcde 13124
+ lkjhg 5678
-abcde
-lkjhg
```

```
==12025== Conditional jump or move depends on uninitialised value(s)
==12025==    at 0x10B140: operator>>(std::istream&,TString&) (string.cpp:121)
==12025==    by 0x1095FC: main (main.cpp:17)
==12025==
==12025==
==12025== HEAP SUMMARY:
==12025==     in use at exit: 123,504 bytes in 10 blocks
==12025==   total heap usage: 26 allocs,16 frees,198,524 bytes allocated
==12025==
==12025== 56 bytes in 1 blocks are indirectly lost in loss record 1 of 10
==12025==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreloac
==12025==    by 0x109FF8: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:127)
==12025==    by 0x10A089: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:133)
==12025==    by 0x10A112: TAvlTree::Insert(TString const&,unsigned long const&)
(avl.cpp:143)
==12025==    by 0x1096AE: main (main.cpp:23)
==12025==
==12025== 256 bytes in 1 blocks are indirectly lost in loss record 2 of 10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x10AD65: TString::TString(TString const&) (string.cpp:23)
==12025==    by 0x109B8C: TAvlNode::TAvlNode(TString const&,unsigned long const&)
(avl.cpp:19)
==12025==    by 0x10A00E: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:127)
==12025==    by 0x10A112: TAvlTree::Insert(TString const&,unsigned long const&)
(avl.cpp:143)
==12025==    by 0x1096AE: main (main.cpp:23)
==12025==
==12025== 256 bytes in 1 blocks are indirectly lost in loss record 3 of 10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x10AD65: TString::TString(TString const&) (string.cpp:23)
==12025==    by 0x109B8C: TAvlNode::TAvlNode(TString const&,unsigned long const&)
(avl.cpp:19)
==12025==    by 0x10A00E: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:127)
==12025==    by 0x10A089: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:133)
==12025==    by 0x10A112: TAvlTree::Insert(TString const&,unsigned long const&)
(avl.cpp:143)
```

```
==12025==    by 0x1096AE: main (main.cpp:23)
==12025==
==12025== 624 (56 direct,568 indirect) bytes in 1 blocks are definitely lost
in loss record 4 of 10
==12025==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreloa
==12025==    by 0x109FF8: TAvlTree::InsertInNode(TAvlNode*,TString const&,unsigned
long const&) (avl.cpp:127)
==12025==    by 0x10A112: TAvlTree::Insert(TString const&,unsigned long const&)
(avl.cpp:143)
==12025==    by 0x1096AE: main (main.cpp:23)
==12025==
==12025== 8,192 bytes in 1 blocks are still reachable in loss record 5 of 10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocat
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4EDFB80: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== 8,192 bytes in 1 blocks are still reachable in loss record 6 of 10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocat
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4EDFBA1: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== 8,192 bytes in 1 blocks are still reachable in loss record 7 of 10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x4F2C097: std::basic_filebuf<char,std::char_traits<char>>::_M_allocat
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F29E72: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4EDFBC2: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== 32,768 bytes in 1 blocks are still reachable in loss record 8 of
10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgprelo
==12025==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_a
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
```

```
==12025==    by 0x4EDFC37: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== 32,768 bytes in 1 blocks are still reachable in loss record 9 of
10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreloc
==12025==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_a
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4EDFC51: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== 32,768 bytes in 1 blocks are still reachable in loss record 10 of
10
==12025==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreloc
==12025==    by 0x4F2DE7A: std::basic_filebuf<wchar_t,std::char_traits<wchar_t>>::_M_a
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4F2A052: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==12025==    by 0x4EDFC6B: std::ios_base::sync_with_stdio(bool) (in /usr/lib/x86_64-li
==12025==    by 0x109590: main (main.cpp:7)
==12025==
==12025== LEAK SUMMARY:
==12025==    definitely lost: 56 bytes in 1 blocks
==12025==    indirectly lost: 568 bytes in 3 blocks
==12025==      possibly lost: 0 bytes in 0 blocks
==12025==    still reachable: 122,880 bytes in 6 blocks
==12025==         suppressed: 0 bytes in 0 blocks
==12025==
==12025== For counts of detected and suppressed errors,rerun with: -v
==12025== Use --track-origins=yes to see where uninitialised values come from
==12025== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Видим, что при добавлении двух элементов и их удалении имеем 2 ошибки, причем точно потеряно 56 байт в 1 блоке (указатель удалён, а память нет), а также valgrind нашел 3 указателя на области памяти, для которых нет указателей на начало памяти. Проанализировав входные данные и вывод valgrind об месте инициализации утерянных кусков памяти, переходим к рассмотрению метода удаления вершины из дерева и перегруженного оператора ввода:

```
std::istream& operator>>(std::istream& is, TString& lhs) {
    if (lhs.buffer) {
        delete[] lhs.buffer;
    }

    lhs.buffer = new char[256];
    lhs.capacity = 256;

    is >> lhs.buffer;
    size_t size = 0;
    while(lhs.buffer[size] != '\0') {
        ++size;
    }
    lhs.size = size;
    return is;
}
```

При выделении памяти под строку, если размер строки меньше 256, то не все ячейки памяти буду проинициализированны, что вызывает ошибку. Исправляется инициализацией всех ячеек памяти нулевым байтом при её выделении.

```
TAvlNode* TAvlTree::SubRemove(TAvlNode* node, const TString& key) {
    if (!node) {
        std::cout << "NoSuchWord\n";
        return nullptr;
    }

    if (key < node->key) {
        node->left = SubRemove(node->left, key);
    } else if (key > node->key) {
        node->right = SubRemove(node->right, key);
    } else {
        //
        TAvlNode* rSon = node->right;
        TAvlNode* lSon = node->left;

        if (!rSon) {
            std::cout << "OK\n";
            return lSon;
        } else if (!lSon) {
            std::cout << "OK\n";
            return rSon;
```

```
22          } else {
23              //
24              node->right = RemoveMin(node, rSon);
25              std::cout << "OK\n";
26          }
27      }
28      return Balance(node);
29  }
```

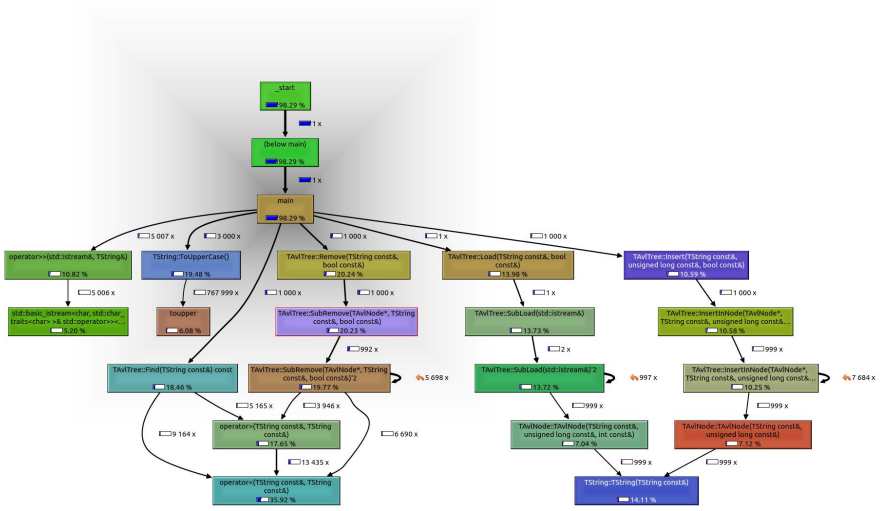
В функции удаления видим, что в случае удаления узла хотя бы с одним ребёнком, забываем удалять данный узел.

```
1  std::ios_base::sync_with_stdio(false);
2  std::cin.tie(nullptr);
```

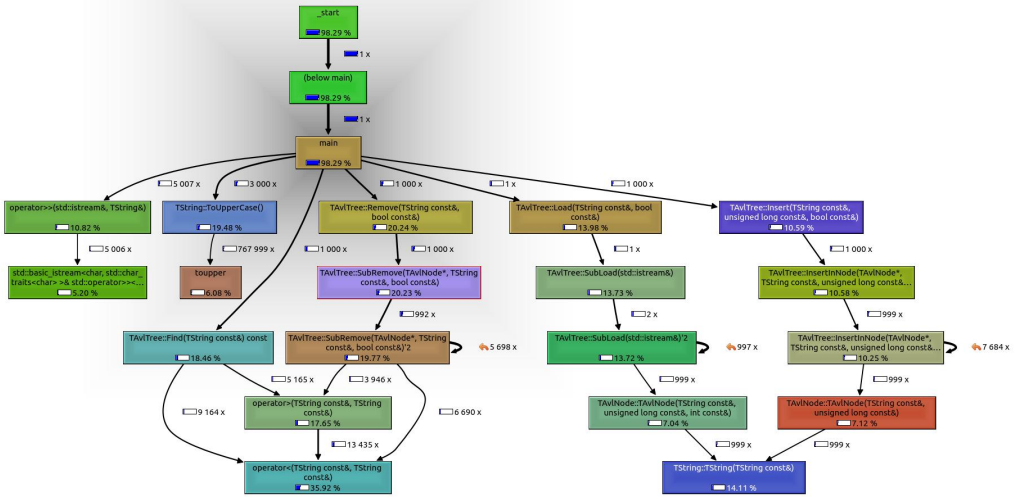
Две данные команды вызывают ошибки типа *still reachable*. В контексте данной лабораторной работы их использование оправдано, так как позволяет ускорить ввод данных. Первая команда отключает синхронизацию потоков C и C++. Вторая отвязывает поток ввода от потока вывода.

3 Callgrind

Данный модуль позволяет собрать информацию о дереве вызова функций в программе. По умолчанию он собирает данные о количестве выполненных инструкций, зависимостях между вызывающей и вызываемой функциями и количество вызовов конкретных функций. Данные собранные модулем выводятся в файл callgrind.out.<pid>, который затем может быть проанализирован с помощью программs kcachegrind.



Можем заметить, что функция удаления работает медленнее, чем функция загрузки. После исправления недочётов, получим следующую картинку



4 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я научился пользоваться некоторыми утилитами профилирования. Понял, насколько данные программы помогают в разработке ПО: помогают упростить поиск ошибок и утечек, а также проанализировать время выполнения функций, чтобы добиться максимальной производительности. Без использования данных утилит, процесс разработки программы усложнится, а требуемое время для проверки программы увеличится.

Список литературы

[1] *Особенности профилирования программ на C++ — Хабр.*
URL: `https://habr.com/ru/post/482040/` (дата обращения: 25.11.2020).

[2] *Valgrind.*
URL: `http://alexott.net/ru/writings/prog-checking/Valgrind.html` (дата обращения: 2.12.2020).