Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

Лабораторная работа №3
по предмету "Дискретный анализ"

Студент: Елистратова П.А.
Преподаватель: Макаров Н.К.
Группа: М8О-210Б-21
Дата:
Оценка:
Подпись:

Москва 2023 г.

Оглавление

Цель работы.....3
Постановка задачи.....3
Описание.....3
Ход работы.....4
Вывод.....6

Цель работы

Исследовать качество программы с помощью утилит gprof и valgrind.

Постановка задачи

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.

Описание

Для проведения профайлинга будем использовать утилиты `gprof` и `valgrind`.

`Gprof` - инструмент анализа производительности приложений. Он использует гибрид инструментария и выборки и был создан как расширенная версия более старого инструмента "`prof`". В отличие от `prof`, `gprof` способен собирать и распечатывать ограниченные графики вызовов. Чтобы начать пользоваться `gprof`, нужно сначала скомпилировать программу с ключом `-pg`:

```
g++ lab2.cpp -pg
```

Запустим программу, подав на вход файл `input.txt`, где находится 300000 команд добавления ключей, их проверки и удаления:

```
./a.out < input.txt.
```

После выполнения будет сгенерирован файл `gmon.out`. В нём содержится вся информация, которую `gprof` смог собрать о программе во время её исполнения. Чтобы её просмотреть выполним команду:

```
gprof ./a.out
```

`Valgrind` - инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, а также профилирования. Это один из самых популярных инструментов для профилирования программ. Чтобы приступить к его использованию, нам достаточно выполнить команду:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose --log-file=valgrind-out.txt ./a.out < input.txt
```

Ход работы

Запустим утилиту **`gprof`**:

```
polina@pelis:~/diskran/lab2$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 55.00      0.39     0.39   299999     1.30     1.30  BTreeNode::search(char*, unsigned long long&)
 11.28      0.47     0.08   300000     0.27     0.27  StringToLower(char*)
  7.05      0.52     0.05    99343     0.50     0.60  BTreeNode::InsertNonFull(InputData&)
  5.64      0.56     0.04    18294     2.19     2.19  BTreeNode::BorrowFromNext(int)
  5.64      0.60     0.04    16501     2.43     2.43  BTreeNode::merge(int)
  4.23      0.63     0.03   115822     0.26     1.13  BTreeNode::deletion(InputData&)
  2.82      0.65     0.02    34576     0.58     0.58  BTreeNode::BorrowFromPrev(int)
  2.82      0.67     0.02    16478     1.22     2.36  BTreeNode::RemoveFromNonLeaf(int)
  1.41      0.68     0.01    99344     0.10     1.81  BTree::deletion(InputData&)
  1.41      0.69     0.01    99344     0.10     0.10  BTreeNode::RemoveFromLeaf(int)
  1.41      0.70     0.01    16507     0.61     0.61  BTreeNode::BTreeNode(bool)
  1.41      0.71     0.01     8881     1.13     1.13  BTreeNode::GetPredecessor(int)
  0.00      0.71     0.00   300000     0.00     1.30  BTree::search(char*, unsigned long long&)
  0.00      0.71     0.00    99344     0.00     0.60  BTree::insert(InputData&)
  0.00      0.71     0.00    65711     0.00     1.39  BTreeNode::FillNode(int)
  0.00      0.71     0.00    16501     0.00     0.61  BTreeNode::SplitChild(int, BTreeNode*)
  0.00      0.71     0.00     3937     0.00     0.00  BTreeNode::GetSuccessor(int)
  0.00      0.71     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z13StringToLowerPc
  0.00      0.71     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.71     0.00        1     0.00     0.00  BTree::BTree()
  0.00      0.71     0.00        1     0.00     0.00  BTree::~BTree()
```

Проанализировав вывод утилиты, мы видим, что чаще всего программе приходится вызывать функции `search` и `StringToLower`. Действительно, ведь перед любым действием мы сначала приводим входящую строку к нижнему регистру, а затем ищём её в дереве, чтобы не вставить\удалить повторяющееся значение. Функция поиска скорее всего работает так долго из-за сравнения строк большой длины. На третьем месте по времени выполнения функция `InsertNonFull`. В этой функции происходит сравнение ключей, нахождение места для вставки и сдвиг значений, чтобы вставить входящее, что тоже занимает время.

Запустим утилиту **`valgrind`** со следующими флагами:

--*leak-check=full*: будет подробно показана каждая отдельная утечка

--*show-leak-kinds=all*: Показать все «определенные, непрямые, возможные, достижимые» виды утечек в «полном» отчете.

--*track-origins=yes*: предпочитать полезный результат скорости. Это отслеживает происхождение неинициализированных значений, что может быть очень полезно при ошибках памяти.

--*verbose*: Может рассказать вам о необычном поведении вашей программы.

--*log-file*: Запись в файл.

```
==12294== LEAK SUMMARY:
==12294==    definitely lost: 0 bytes in 0 blocks
==12294==    indirectly lost: 0 bytes in 0 blocks
==12294==      possibly lost: 0 bytes in 0 blocks
==12294==    still reachable: 122,880 bytes in 6 blocks
==12294==         suppressed: 0 bytes in 0 blocks
==12294==
==12294== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

На данный момент ошибок не обнаружено, потому что я следила за ошибками, пока делала лабораторную работу №2.

Но сейчас я воспроизведу ошибки с которыми столкнулась в основном из-за невнимательности во время реализации задания лабораторной №2.

1.

```
==12670== Conditional jump or move depends on uninitialised value(s)
==12670==    at 0x4A8E1E3: tolower (ctype.c:46)
==12670==    by 0x1094A3: StringToLower(char*) (search.cpp:17)
==12670==    by 0x10B2BC: main (search.cpp:611)
==12670==  Uninitialised value was created by a stack allocation
==12670==    at 0x10B1DB: main (search.cpp:598)
```

```
==12670== 242 errors in context 4 of 4:
==12670== Conditional jump or move depends on uninitialised value(s)
==12670==    at 0x4A8E1E3: tolower (ctype.c:46)
==12670==    by 0x1094A3: StringToLower(char*) (search.cpp:17)
==12670==    by 0x10B2BC: main (search.cpp:611)
==12670==  Uninitialised value was created by a stack allocation
==12670==    at 0x10B1DB: main (search.cpp:598)
==12670==
==12670== ERROR SUMMARY: 786 errors from 4 contexts (suppressed: 0 from 0)
```

Ошибка возникала из-за того что в функции StringToLower я не учла, что строки передаются не всегда максимальной длины и в tolower() оказываются неинициализированные значения. Исправила тем, что в цикле теперь иду до длины строки.

Было:

```
const unsigned short MAX_KEY_LENGTH = 257;
const unsigned short MAX_ARRAY_LENGTH = 2 * TREE_DEGREE - 1;


void StringToLower(char* string_key)
{
    // int length = strlen(string_key);
    for (int i = 0; i < MAX_KEY_LENGTH; ++i)
    {
        string_key[i] = tolower(string_key[i]);
    }
}
```

Стало:

```
void StringToLower(char* string_key)
{
    int length = strlen(string_key);
    for (int i = 0; i < length; ++i)
    {
        string_key[i] = tolower(string_key[i]);
    }
}
```

2.

```
==14116== LEAK SUMMARY:
==14116==    definitely lost: 166,296 bytes in 6,929 blocks
==14116==    indirectly lost: 41,942,432 bytes in 42,585 blocks
==14116==      possibly lost: 17,136 bytes in 7 blocks
==14116==    still reachable: 122,880 bytes in 6 blocks
==14116==         suppressed: 0 bytes in 0 blocks
==14116==
==14116== ERROR SUMMARY: 11 errors from 11 contexts (suppressed: 0 from 0)
```

Посмотрев вывод, я поняла, что не очищала массив указателей на детей в фунциях merge (когда склеила две ноды в одну, а память под уже склеенной ноды не высвобождала) и deletion (в случае, когда корень оказался пуст). После исправлений ошибок не осталось.

Вывод

Выполнив данную лабораторную работу, я познакомилась с очень полезными утилитами `valgrind` и `gprof`, которые помогают отлаживать свою программу путем анализа времени и памяти программы. Анализ времени позволяет увидеть, сколько времени занимает каждая из функций, что поможет для оптимизации кода. Анализ потребления памяти позволяет выявить неочевидные утечки памяти, которые благодаря утилите можно будет исправить. Эти инструменты позволяют совершенствовать свой код, делая его более стабильным и производительным.