Лабораторная работа № 3 по курсу Дискретный Анализ. Исследование качества программ

Выполнил студент группы 08-207 МАИ *Павлов Иван*.

Условие

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.

Дневник выполнения работы

Для выполнения работы я воспользовался утилитами *gprof* и *valgrind*.

Запускаю программу с утилитой *gprof*. Она позволяет увидеть время работы всех функций, реализованных в программе и количество их вызовов, вычисляет процентное соотношение работы конкретной функции от работы всей программы.

Скомпилируем код при помощи команды **g++ main.cpp -pg -o main**. -pg - флаг, который выводит необходимые данные в специальный файл *gmon.out*.

Сгенерируем файл **t.txt**, состоящий из 1000000 команд на поиск, вставку, удаление. Запустим **./main < t.txt**.

После выполнения просмотрим *gmon.out* с помощью команды **gprof main**:

```
 time   seconds   seconds    calls  us/call  us/call  name
41.67      4.80     4.80  3000000     1.60     1.60  String::String
   (char const*)
31.12      8.38     3.58 55653190     0.06     0.06  String::
   operator<(String const&) const
14.32     10.04     1.65 264342764     0.01     0.01  getHeight(
   Tree*)
7.51     10.90     0.86 28552989     0.03     0.03  String::
   operator>(String const&) const
1.48     11.07     0.17  1000000     0.17     2.57  insert(Tree*,
   String&, unsigned long long)
1.13     11.20     0.13  1230544     0.11     0.11  successor(Tree
   *)
1.04     11.32     0.12  1000000     0.12     1.67  find(Tree*,
   String&)
0.87     11.42     0.10  1000000     0.10     2.47  erace(Tree*,
   String&)
0.43     11.47     0.05 73810807     0.00     0.01  balance(Tree*)
0.17     11.49     0.02  4000000     0.01     0.01  String::~String
   ()
```

```
12    0.09      11.50      0.01   1000000      0.01      0.01   String::String
      ()
13    0.09      11.51      0.01    615272      0.02      0.09
      eraceMinimalNode(Tree*)
14    0.09      11.52      0.01    561285      0.02      0.06   leftRotate(Tree
      *)
15    0.00      11.52      0.00   1615272      0.00      0.00   String::
      operator=(String const&)
16    0.00      11.52      0.00   1000000      0.00      0.01   Tree::Tree(
      String const&, unsigned long long)
17    0.00      11.52      0.00   1000000      0.00      0.01   Tree::~Tree()
18    0.00      11.52      0.00    536322      0.00      0.04   rightRotate(
      Tree*)
19    0.00      11.52      0.00         1      0.00      0.00
      __static_initialization_and_destruction_0(int, int)
20    0.00      11.52      0.00         1      0.00      0.00   destroy(Tree*)
```

Видно, что наибольшее время работы занимают функции, связанные с работой дерева, в частности создание узла включает вызов конструктора для создания строки, который занимает большую часть работы программы. Это связано с тем, что я подал на вход 1000000 строк длиной 255.

Теперь запустим программу с утилитой *valgrind*, которая позваляет обнаруживать *RE* и утечки памяти. Для начала просто компилируем программу с флагом -g для отображения строк с ошибками **g++ -g main.cpp -o main**. Затем запускаем утилиту командой **valgrind ./main**. Вывод программы на рабочем коде:

```
1  ==13220==
2  ==13220== HEAP SUMMARY:
3  ==13220==     in use at exit: 0 bytes in 0 blocks
4  ==13220==   total heap usage: 10 allocs, 10 frees, 83,872 bytes
      allocated
5  ==13220==
6  ==13220== All heap blocks were freed -- no leaks are possible
7  ==13220==
8  ==13220== For lists of detected and suppressed errors, rerun with:
      -s
9  ==13220== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
      from 0)
```

Сравнение работы исправленной программы с предыдущей версией.

В процессе решения ЛР2 у меня была *RE*, где я вызывал *fclose* для несуществующего файла. Исправил я эту ошибку, используя *valgrind*. Вывод для нерабочего кода:

```
==13822== Memcheck, a memory error detector
==13822== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13822== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==13822== Command: ./main1
==13822==
==13822== Invalid read of size 4
==13822==    at 0x4B48CFB: fclose@@GLIBC_2.2.5 (iofclose.c:48)
==13822==    by 0x10AD77: main (main1.cpp:433)
==13822==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==13822==
==13822==
==13822== Process terminating with default action of signal 11 (SIGSEGV)
==13822==  Access not within mapped region at address 0x0
==13822==    at 0x4B48CFB: fclose@@GLIBC_2.2.5 (iofclose.c:48)
==13822==    by 0x10AD77: main (main1.cpp:433)
==13822==  If you believe this happened as a result of a stack
==13822==  overflow in your program's main thread (unlikely but
==13822==  possible), you can try to increase the size of the
==13822==  main thread stack using the --main-stacksize= flag.
==13822==  The main thread stack size used in this run was 8388608.
==13822==
==13822== HEAP SUMMARY:
==13822==     in use at exit: 76,800 bytes in 2 blocks
==13822==   total heap usage: 3 allocs, 1 frees, 77,272 bytes allocated
==13822==
==13822== LEAK SUMMARY:
==13822==    definitely lost: 0 bytes in 0 blocks
==13822==    indirectly lost: 0 bytes in 0 blocks
==13822==      possibly lost: 0 bytes in 0 blocks
==13822==    still reachable: 76,800 bytes in 2 blocks
==13822==         suppressed: 0 bytes in 0 blocks
==13822== Rerun with --leak-check=full to see details of leaked memory
==13822==
==13822== For lists of detected and suppressed errors, rerun with:
```

```
    -s
==13822== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
    from 0)
```

Утилита показала, что ошибку вызвал системный вызов $fclose$ и вывел строку ошибки (main1.cpp:433).

Выводы

В ходе выполнения лабораторной работы были изучены инструменты для анализа и оптимизации производительности программного кода.

Утилита *valgrind* позволяет обнаруживать и исправлять ошибки памяти, такие как утечки памяти, чтение или запись в неправильные области памяти, и использование неинициализированных переменных. Также *valgrind* предоставляет информацию о производительности программы, позволяя оптимизировать код. Благодаря инструментам *valgrind* можно существенно улучшить качество и производительность программного кода.

Утилита $gprof$ предназначена для анализа производительности программного кода и определения узких мест. Она позволяет вычислить время выполнения каждой функции и подсчитать количество вызовов каждой функции.

В целом, использование утилит *valgrind* и $gprof$ может значительно повысить качество и производительность программного кода. Работа с этими инструментами является важным компонентом разработки программного обеспечения, их использование позволяет создавать более качественные и эффективные программы.