

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: Р. Р. Гаптулхаков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б
Дата: 03.12.20
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 недочётах.
- Сравнение работы исправленной программы с предыдущей версией.
- Общих выводов о выполнении лабораторной работы, полученном опыте. Минимальный набор используемых средств должен содержать утилиту `gprof` и библиотеку `dmalloc`, однако их можно заменять на любые другие аналогичные или более развитые утилиты (например, `Valgrind` или `Shark`) или добавлять к ним новые (например, `gcov`).

Используемые утилиты: `valgrind` (`callgrind`, `kcache-grind`), `gcov` (`lcov`, `genhtml`).

1 Описание

Использование различных утилит профилировки и отслеживания утечек памяти, позволяет программистам сильно оптимизировать код. Я буду использовать две утилиты `valgrind`. Профилирование будем выполнять с помощью **`callgrind`**.

В качестве дополнения посмотрим на покрытость моего кода при помощи утилиты **`gcov`**.

Тест, на котором производилось исследование, состоит из 5000000 строк и включает вставку, удаление, поиск, сохранение и загрузку из файла.

2 Дневник отладки

Для начала протестируем программу с помощью **valgrind**-а, используя при этом ключи `-leak-check=full -show-leak-kinds=all -log-file=valgrind-out.txt`, которые включают функцию обнаружения утечки, показывают все типы утечек и выводят результат в выходной файл соответственно.

```
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
valgrind --leak-check=full --show-leak-kinds=all --log-file=valgrind-out.txt  
./solution <./tests/01.t >mytest.txt
```

Результаты:

```
==7805== Memcheck, a memory error detector  
==7805== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==7805== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info  
==7805== Command: ./solution  
==7805== Parent PID: 7637  
==7805==  
==7805== error calling PR_SET_PTRACER, vgdb might block  
==7805==  
==7805== HEAP SUMMARY:  
==7805==      in use at exit: 0 bytes in 0 blocks  
==7805==    total heap usage: 4,141,790 allocs, 4,141,790 frees, 1,628,799,445  
bytes allocated  
==7805==  
==7805== All heap blocks were freed --no leaks are possible  
==7805==  
==7805== For counts of detected and suppressed errors, rerun with: -v  
==7805== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как видно, никаких ошибок с утечками памяти нет. Но при выполнении 2 лабораторной работы, я получал множество ошибок, в следствие исправлял их. Вывод: программа работает корректно без утечек. Для профилирования программы нужно её скомпилировать с ключами «-g -Ofast -no-pie». Профилировать необходимо на больших тестах.

```
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
g++ -g -Ofast -no-pie main.cpp bst.cpp -o solution  
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
valgrind --tool=callgrind ./solution <01.t >mytest.txt
```

```
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
callgrind_annotate callgrind.out.15018 >doc.txt
```

Посмотрим на наш результат:

В репорте можно увидеть информацию о том, сколько времени длился профилирование, сколько было снято проб, сколько проб снято по функции, и процентное соотношение работы функции от всей программы. Выведем часть результата.

```
Timerange: Basic block 0 -76883415  
Trigger: Program termination  
Profiled target: ./solution (PID 6810,part 1)  
Events recorded: Ir  
Events shown: Ir  
Event sort order: Ir  
Thresholds: 99  
Include dirs:  
User annotated:  
Auto-annotation: on
```

```
-----  
Ir  
-----
```

```
284,164,632 (100.0%) PROGRAM TOTALS
```

```
-----  
Ir file:function  
-----
```

```
58,453,560 (20.57%) ???::std::istreambuf_iterator<char,std::char_traits<char>>std::num  
long>(std::istreambuf_iterator<char,std::char_traits<char>>,std::istreambuf_iterator<  
long&) const [/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28]  
43,356,415 (15.26%) bst.cpp:bst::Tree::Insert(char*,unsigned long,bst::TreeNode*)  
[/home/nikita/LR2-DA/solution]  
21,335,007 ( 7.51%) ???::std::basic_istream<char,std::char_traits<char>>& std::operat  
[/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28]  
15,371,497 ( 5.41%) bst.cpp:bst::Tree::Search(char*,bst::TreeNode*)'2 [/home/nikita/L  
13,952,554 ( 4.91%) ???::std::istream::sentry::sentry(std::istream&,bool) [/usr/lib/x
```

Теперь посмотрим на покрытость моего кода. Покрытие - это процент задействованности строчек кода. Для этого воспользуемся утилитой **gcov**. Скомпилируем наш файл `main.cpp`, `bst.cpp` при помощи флага «`--coverage`».

```
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
g++ --coverage -g main.cpp bst.cpp -o cover  
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
gcov main.cpp >cover.txt  
rusya@DESKTOP-K0H2IC0:/mnt/c/Users/fynex/Desktop/continuous/DA/LR2/source$  
gcov bst.cpp >cover2.txt
```

Результат оказался крайне неожиданным, покрытие для моего кода в файле `main.cpp` составляет 35

3 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я научился работать с утилитами **valgrind (callgrind)** и **gcov**.

Valgrind - мощная утилита, позволяющая существенно ускорить отладку и профилирование программ. Она позволяет отлично находить различные утечки памяти выделенны на куче. Я обязательно буду пользоваться ей и в дальнейшем при профилировании и отладке программ.

Сегодня требования к производительности ПО очень разные, но вероятно не секрет, что многие приложения имеют очень жесткие требования по скорости выполнения. Что касается анализа производительности, то можно сказать, что во многом успех программы зависит от скорости производительности, а в некоторых программах основной упор делается на это.

Анализ производительности - очень трудоемкий процесс, поэтому были созданы инструменты, которые способны упростить его.

Список литературы

[1] *Работа с dmalloc*

URL: <http://alexott.net/ru/linux/valgrind/DMalloc.html> (дата обращения: 17.11.2020).

[2] *Valgrind Frequently Asked Questions*

URL: <https://www.valgrind.org/docs/manual/faq.html#faq.deflost> (дата обращения: 15.11.2020).