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. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 05.12.2020
Оценка: хорошо
Подпись:

Москва, 2020

Лабораторная работа №3

Задача: Требуется разработать программу, осуществляющую ввод пар «ключ-значение», их упорядочивание по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод отсортированной последовательности.

Используемые утилиты: valgrind, gprof, gcov.

1 Описание

Valgrind. Согласно [1], valgrind - это многоцелевой инструмент профилирования кода и отладки памяти для Linux. Чаще всего утилита используется для обнаружения ошибок при работе с памятью.

Инструмент memcheck утилиты valgrind позволяет обнаружить следующие виды ошибок:

- Invalid read / Invalid write - неверное использование указателей, выход за границы массива, считывание или запись в ячейку памяти, не принадлежащей программы
- Conditional jump or move depends on uninitialised value(s) - использование неинициализированных переменных
- Invalid free / delete / delete[] - ошибка в использовании функций для освобождения памяти. Например, повторное освобождение памяти или освобождение памяти, выделенной через new, при помощи free.
- Memory leaks - выделенная память не освобождается после завершения работы программы, возникает утечка памяти. Большое количество утечек памяти может негативно сказываться на работе операционной системы

Утилита valgrind выводит пользователю список всех найденных ошибок. К каждой ошибке прикрепляется состояние стека вызовов на момент её обнаружения. При помощи этой информации пользователь может определить, какая функция вызвала ту или иную ошибку. В случае обнаружения утечки памяти valgrind может вывести состояние стека вызовов на момент выделения памяти для проблемной переменной и её адрес. В некоторых случаях это помогает быстрее устранить ошибку.

Gprof. Многие программы обладают жёсткими требованиями по производительности, поэтому работу многих функций приходится оптимизировать по максимуму. Однако не всегда оптимизация функции приведёт к значительному росту производительности: если данная функция вызывается всего пару раз, то её оптимизация будет не очень уж и полезной. Чтобы определить слабые места в коде, необходимо воспользоваться профилировщиком. Я буду рассматривать утилиту gprof (gnu profiler).

Профилировщики собирают данные во время работы программы. Большинство из них работают по следующему принципу. С некоторой периодичностью профилировщик останавливает программу, записывает текущее состояние стека вызовов и возобновляет её работу. После завершения сбора данных он собирает статистику, по которой можно понять, сколько процентов от общего времени работы программы занимает время работы какой-либо функции. Функции, который имеют наибольшую

долю рабочего времени, являются первоочерёдными кандидатами на оптимизацию. Стоит заметить, что для достоверности собираемой информации программу под профилировщиком следует запускать на больших тестах, охватывающих все функции программы.

Gcov. Чтобы убедиться в том, что наши тесты охватывают проверку всех функций программы, необходимо проверить покрытие кода. Покрытие покажет, какие строчки кода выполнялись в процессе работы программы. Если при запуске программы с большим тестом некоторые строки кода не были тронуты, то либо в них закралась ошибка, либо они вообще лишние. Для проверки покрытия можно использовать утилиту gcov (gnu coverage). Для простоты в лабораторной работе я буду взаимодействовать с gcov не через консоль, а через среду разработки CLion.

2 Консоль

Рассмотрим использование утилиты valgrind. Я буду запускать её с ключами –tool=memcheck (чтобы явно указать, что мне требуется анализ работы с памятью), –leak-check=full (чтобы утилита вывела мне подробную информацию о каждой найденной ошибке), –show-leak-kinds=all (чтобы утилита вывела мне абсолютно все найденные ошибки). Программа предварительно будет скомпилирована с флагом -g (чтобы valgrind вывел более точную позицию возникновения ошибок).

```
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O3 -lm -pg -g main.cpp
TBTree.h TNode.h -o btree
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ valgrind -tool=memcheck -leak-check=full
-show-leak-kinds=all ./btree <test >result
==3201== Memcheck,a memory error detector
==3201== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==3201== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3201== Command: ./btree
==3201==
==3201==
==3201== HEAP SUMMARY:
==3201== in use at exit: 0 bytes in 0 blocks
==3201== total heap usage: 15,447 allocs,15,447 frees,28,818,928 bytes allocated
==3201==
==3201== All heap blocks were freed - no leaks are possible
==3201==
==3201== For counts of detected and suppressed errors,rerun with: -v
==3201== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind показал, что в моей программе нет проблем при работе с памятью. При написании кода для второй лабораторной работы я столкнулся с несколькими десятками ошибок в valgrind. В основном это были утечки памяти и выходы за границы массива. При отладке все ошибки были исправлены. Сейчас, как видно в протоколе утилиты, моя программа корректно работает с памятью и является безопасной.

Теперь попробуем прогнать мою программу через профилировщик. Для того чтобы gprof собрал необходимую информацию о работе программы, требуется скомпилировать её с ключом -pg и запустить её на достаточно большом тесте. Профилировщик соберёт необходимую информацию в файл gmon.out. Затем нужно вызвать утилиту gprof, которая на основе собранных данных сформирует отчёт.

```
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ g++ main.cpp TBTree.h TNode.h -pg -o btree
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./btree <test >result
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ gprof ./btree gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls us/call us/call name
29.18 0.07 0.07 90000 0.78 0.78 CheckString(char*)
20.84 0.12 0.05 2042116 0.02 0.03 operator<(TPair&,char*)
12.51 0.15 0.03 29987107 0.00 0.00 Min(int,int)
12.51 0.18 0.03 535284 0.06 0.08 operator==(TPair&,char*)
12.51 0.21 0.03 90000 0.33 1.27 SearchNode(TNode*,char*,TNode*&,int&)
4.17 0.22 0.01 39809 0.25 0.25 RemoveFromNode(TNode*,int)
4.17 0.23 0.01 29901 0.33 0.37 InsertNode(TNode*,TPair&)
4.17 0.24 0.01 18782 0.53 0.65 Rebalance(TNode*,int&)
0.00 0.24 0.00 536561 0.00 0.00 operator<(TPair&,TPair&)
0.00 0.24 0.00 160149 0.00 0.15 SearchInNode(TNode*,char*)
0.00 0.24 0.00 30000 0.00 1.64 TBTree::Insert(TPair&)
0.00 0.24 0.00 30000 0.00 2.76 TBTree::Remove(char*)
0.00 0.24 0.00 30000 0.00 1.27 TBTree::Search(char*) const
0.00 0.24 0.00 29901 0.00 1.49 RemoveNode(TNode*,char*)
0.00 0.24 0.00 9939 0.00 0.00 TNode::TNode()
0.00 0.24 0.00 4964 0.00 0.25 MergeNodes(TNode*,int)
0.00 0.24 0.00 4964 0.00 0.00 SplitChild(TNode*,int)
0.00 0.24 0.00 1 0.00 0.00 _GLOBAL__sub_I__Z3Minii
0.00 0.24 0.00 1 0.00 0.00 DeleteTree(TNode*)
0.00 0.24 0.00 1 0.00 0.00 FileToTree(TNode*,_IO_FILE*)
0.00 0.24 0.00 1 0.00 0.00 NodeToFile(TNode*,_IO_FILE*)
0.00 0.24 0.00 1 0.00 0.00 __static_initialization_and_destruction_0(int,int)
0.00 0.24 0.00 1 0.00 0.00 TBTree::Deserialize(_IO_FILE*)
0.00 0.24 0.00 1 0.00 0.00 TBTree::Delete()
0.00 0.24 0.00 1 0.00 0.00 TBTree::Serialize(_IO_FILE*)
0.00 0.24 0.00 1 0.00 0.00 TBTree::TBTree()
0.00 0.24 0.00 1 0.00 0.00 TBTree::~TBTree()
```

Отчёт профилировщика достаточно большой и содержит всю необходимую информацию для интерпретации полученных данных. Я прикрепил, пожалуй, самое основное. В первом столбике данной таблице указано, какую часть от общего времени работы программы заняло выполнение каждой функции. Можно увидеть, что около 50% времени работы программы заняла обработка и сравнение ключей (напомню, что в

качестве ключей выступают строки с длиной до 256 символов, а под их обработкой подразумевается приведение всех её элементов к одному регистру).

Объясняется это тем, что функции работы с ключами - одни из немногих функций в программе, работающих за линейное время. При этом они вызываются постоянно: обработка ключа производится в начале выполнения каждой команды, сравнение ключей производится во время поиска, вставки и удаления элементов. Для улучшения производительности программы следует оптимизировать данные функции. Однако стоит заметить, что от линейного времени работы данных функций уйти довольно сложно и оптимизация этих функций повлечет за собой дополнительные расходы памяти.

Проверив свою программу при помощи утилиты gcov, я получил следующие результаты. При выполнении теста, состоящего из 400 тысяч команд разного вида, мой код оказался покрыт на 95%. При этом непокрытыми остались фрагменты функций, выводящих сообщения о системных ошибках. Также при анализе полученных результатов я обнаружил одно лишнее условие в своей программе, которое ни разу не выполнилось. Его тело было удалено.

3 Дневник выполнения работы

1. Исследование программы на ошибки в работе с памятью при помощи valgrind. Ошибок не обнаружено.

2. Профилирование программы при помощи утилиты gprof. Установлено, что наиболее «долгие» функции - функции обработки ключей, оптимизация которых достаточно затратна.

3. Проверка покрытия кода при помощи утилиты gcov. Установлено, что код покрыт тестами на 95%. Непокрытыми остались фрагменты функций, выводящих сообщения о возникновении системных ошибок. Обнаружено лишнее условное выражение, которое ни разу не выполнилось. Его тело было удалено.

4 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я изучил утилиты, которые могут оказаться полезными при поиске ошибок и оптимизации программ, а именно valgrind, gprof, gcov. Этого набора достаточно для получения подробных сведений о программе, которые укажут на её недостатки и направят на верный путь для оптимизации.

Утилита valgrind укажет на ошибки при работе с памятью, gprof выполнит профилирование кода, а gcov покажет покрытие кода тестом.

В среде разработки CLion можно запустить данные утилиты нажатием одной кнопки (вместо печати команд с кучей флагов), что упрощает их использование. Причём CLion умеет анализировать полученные результаты и может сгенерировать более наглядный и интерактивный отчёт. Я уверен, что в дальнейшем не раз воспользуюсь этими утилитами для улучшения своих программ, написанных на C/C++.

Список литературы

[1] *Информация о valgrind.*
URL: `http://cppstudio.com/post/4348/` (дата обращения: 27.11.2020).

[2] *Информация о gprof.*
URL: `https://www.ibm.com/developerworks/ru/library/l-gnuprof/` (дата обращения: 27.11.2020).