Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Дискретный анализ" №3

Студент: Кострюков Е.С.

Преподаватель: Макаров Н.К.

Группа: М8О-207Б-22

Дата: 27.05.2024

Оценка:

Подпись:

Москва 2024 г.

Оглавление

Цель работы

Исследование качества программ

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.

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).

Общие сведения о программе

Программа представлена файлом – lab2.cpp

Используемые утилиты

gprof (GNU Profiler) — это утилита для профилирования программ, написанных на языках C, C++, Fortran и других, поддерживаемых компиляторами GNU. Она используется для анализа производительности программ с целью выявления "узких мест" и оптимизации кода.

Основные возможности gprof:

1. Сбор статистики:
   - Профайлинг с помощью счётчиков: определяет, сколько раз каждая функция вызывается.
   - Профайлинг с отслеживанием времени: измеряет, сколько времени программа проводит в каждой функции и её потомках.

2. Анализ и представление данных:
   - Создание отчётов о производительности, которые включают информацию о затраченном времени и частоте вызовов функций.

- Представление информации в виде текстовых отчётов, которые помогают разработчикам визуализировать, какие части программы являются наиболее ресурсоёмкими.

Этапы использования gprof:

1. Компиляция программы с профилирующей информацией:
   g++ -pg -Wall -Wextra -Wpedantic lab2.cpp

2. Запуск программы:
   ./a.out < input.txt

3. Анализ данных с помощью gprof:
   gprof a.out gmon.out > result

Интерпретация отчёта

Отчёт, сгенерированный gprof, состоит из нескольких секций:

1. Flat Profile (плоский профиль):
   - Показывает, сколько времени и сколько раз вызывалась каждая функция.
   - Информация представлена в табличной форме и сортирована по убыванию времени выполнения.

2. Call Graph (граф вызовов):
   - Детализированный граф вызовов, показывающий отношения между функциями.
   - Включает информацию о том, какие функции вызывают другие и сколько времени затрачивается на эти вызовы.

Valgrind — это мощный инструмент для анализа и отладки программ, который используется в основном для выявления ошибок памяти и проблем производительности. Он поддерживает несколько инструментов для различных видов анализа, таких как обнаружение утечек памяти, профилирование кэша процессора и трассировка выполнения программы.

Основные возможности Valgrind

1. Memcheck:
   - Инструмент для обнаружения утечек памяти, неправильного использования динамической памяти и использования неинициализированной памяти.
   - Наиболее широко используемый инструмент Valgrind.

2. Callgrind:
   - Инструмент для профилирования производительности, который собирает информацию о количестве инструкций, выполняемых каждой функцией, и о количестве обращений к кэшу.
   - Полезен для оптимизации кода и анализа производительности.

3. Cachegrind:
   - Инструмент для профилирования кэша процессора, который измеряет пропуски кэша данных и инструкций.
   - Помогает оптимизировать использование кэша и улучшить производительность программы.

4. Helgrind:
   - Инструмент для выявления состояний гонки в многопоточных программах.
   - Полезен для отладки программ с многопоточными взаимодействиями.

5. DRD (Dynamic Race Detector):
   - Альтернативный инструмент для обнаружения состояний гонки и других проблем многопоточности.

6. Massif:
   - Инструмент для профилирования кучи, который анализирует использование динамической памяти и помогает выявлять утечки и избыточное использование памяти.

Этапы использования Valgrind:

1. Установка Valgrind:
   - Valgrind доступен для большинства дистрибутивов Linux и macOS. Установить его можно с помощью менеджера пакетов:
   sudo apt-get install valgrind # Для Debian/Ubuntu

2. Запуск программы с Valgrind:
   - Для запуска программы с использованием Valgrind просто используйте следующую команду:
   valgrind ./my_program

3. Использование конкретного инструмента:
   - Чтобы использовать определённый инструмент Valgrind, добавьте его имя в команду:
   valgrind --tool=memcheck ./my_program # Запуск с Memcheck
   valgrind --tool=callgrind ./my_program # Запуск с Callgrind

4. Анализ отчётов:
   - После выполнения программы Valgrind создаст отчёт, который можно проанализи-

ровать для выявления и исправления проблем.

- Например, Memcheck выдаёт подробную информацию об утечках памяти и неправильном использовании памяти.

Отчёты

Gprof

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
50.00	0.01	0.01	83712	0.12	0.12	Treap::find(Node*, char const*)
50.00	0.02	0.01	6521	1.53	1.53	Treap::split(Node*, char const*, Node*&, Node*&)
0.00	0.02	0.00	83712	0.00	0.24	Treap::process_command(char const*)
0.00	0.02	0.00	83712	0.00	0.12	Treap::find(char const*)
0.00	0.02	0.00	13842	0.00	0.00	Node::Node(char const*, unsigned long, unsigned int*)
0.00	0.02	0.00	13842	0.00	0.00	Treap::erase(Node*, char const*)
0.00	0.02	0.00	13842	0.00	0.00	Treap::erase(char const*)
0.00	0.02	0.00	13842	0.00	0.00	Treap::merge(Node*, Node*)
0.00	0.02	0.00	13842	0.00	0.72	Treap::insert(Node*, Node*)
0.00	0.02	0.00	13842	0.00	0.72	Treap::insert(char const*, unsigned long)
0.00	0.02	0.00	1	0.00	0.00	Treap::Treap()
0.00	0.02	0.00	1	0.00	0.00	Treap::~Treap()

Из этого отчета видно, что:

- Функция Treap::find(Node*, char const*) и Treap::split(Node*, char const*, Node*&, Node*&) занимают по 50% времени выполнения программы и составляют по 0.01 секунды каждая.

- Остальные функции потребляют незначительное количество времени или вообще не потребляют время выполнения.

```
Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 50.00% of 0.02 seconds

index % time    self  children    called     name
                                  <spontaneous>
[1]    100.0    0.00    0.02             main [1]
               0.00    0.02   83712/83712      Treap::process_command(char const*) [2]
               0.00    0.00       1/1          Treap::Treap() [18]
               0.00    0.00       1/1          Treap::~Treap() [19]
-----------------------------------------------
               0.00    0.02   83712/83712      main [1]
[2]    100.0    0.00    0.02   83712        Treap::process_command(char const*) [2]
               0.00    0.01   83712/83712      Treap::find(char const*) [4]
               0.00    0.01   13842/13842      Treap::insert(char const*, unsigned long) [6]
               0.00    0.00   13824/13842      Treap::erase(char const*) [16]
-----------------------------------------------
                         247946          Treap::find(Node*, char const*) [3]
               0.01    0.00   83712/83712      Treap::find(char const*) [4]
[3]     50.0    0.01    0.00   83712+247946  Treap::find(Node*, char const*) [3]
                         247946          Treap::find(Node*, char const*) [3]
-----------------------------------------------
               0.00    0.01   83712/83712      Treap::process_command(char const*) [2]
[4]     50.0    0.00    0.01   83712        Treap::find(char const*) [4]
               0.01    0.00   83712/83712      Treap::find(Node*, char const*) [3]
-----------------------------------------------
                          23472          Treap::insert(Node*, Node*) [5]
               0.00    0.01   13842/13842      Treap::insert(char const*, unsigned long) [6]
[5]     50.0    0.00    0.01   13842+23472   Treap::insert(Node*, Node*) [5]
               0.01    0.00    6521/6521       Treap::split(Node*, char const*, Node*&, Node*&) [7]
                          23472          Treap::insert(Node*, Node*) [5]
-----------------------------------------------
               0.00    0.01   13842/13842      Treap::process_command(char const*) [2]
[6]     50.0    0.00    0.01   13842        Treap::insert(char const*, unsigned long) [6]
               0.00    0.01   13842/13842      Treap::insert(Node*, Node*) [5]
```

```
          0.00    0.00   13842/13842      Node::Node(char const*, unsigned long, unsigned int*) [14]
-----------------------------------------------
                  11190          Treap::split(Node*, char const*, Node*&, Node*&) [7]
          0.01    0.00    6521/6521       Treap::insert(Node*, Node*) [5]
[7]     50.0    0.01    0.00    6521+11190   Treap::split(Node*, char const*, Node*&, Node*&) [7]
                  11190          Treap::split(Node*, char const*, Node*&, Node*&) [7]
-----------------------------------------------
          0.00    0.00   13842/13842      Treap::insert(char const*, unsigned long) [6]
[14]     0.0    0.00    0.00   13842       Node::Node(char const*, unsigned long, unsigned int*) [14]
-----------------------------------------------
                  23533          Treap::erase(Node*, char const*) [15]
          0.00    0.00   13842/13842      Treap::erase(char const*) [16]
[15]     0.0    0.00    0.00   13842+23533   Treap::erase(Node*, char const*) [15]
          0.00    0.00   13842/13842      Treap::merge(Node*, Node*) [17]
                  23533          Treap::erase(Node*, char const*) [15]
-----------------------------------------------
          0.00    0.00      18/13842      Treap::~Treap() [19]
          0.00    0.00   13824/13842      Treap::process_command(char const*) [2]
[16]     0.0    0.00    0.00   13842       Treap::erase(char const*) [16]
          0.00    0.00   13842/13842      Treap::erase(Node*, char const*) [15]
-----------------------------------------------
                    29           Treap::merge(Node*, Node*) [17]
          0.00    0.00   13842/13842      Treap::erase(Node*, char const*) [15]
[17]     0.0    0.00    0.00   13842+29     Treap::merge(Node*, Node*) [17]
                    29           Treap::merge(Node*, Node*) [17]
-----------------------------------------------
          0.00    0.00       1/1          main [1]
[18]     0.0    0.00    0.00       1       Treap::Treap() [18]
-----------------------------------------------
          0.00    0.00       1/1          main [1]
[19]     0.0    0.00    0.00       1       Treap::~Treap() [19]
          0.00    0.00      18/13842      Treap::erase(char const*) [16]
-----------------------------------------------
```

Valgrind

```
==63146== Process terminating with default action of signal 27 (SIGPROF)
==63146==    at 0x4C338B2: __open_nocancel (open64_nocancel.c:39)
==63146==    by 0x4C4385F: write_gmon (gmon.c:393)
==63146==    by 0x4C4420A: _mcleanup (gmon.c:467)
==63146==    by 0x4B5E371: __cxa_finalize (cxa_finalize.c:82)
==63146==    by 0x109356: ??? (in /home/evgeny/Рабочий стол/Лабы/4_sem/DiscrAn/Codes/a.out)
==63146==    by 0x40010F1: _dl_call_fini (dl-call_fini.c:43)
==63146==    by 0x4005577: _dl_fini (dl-fini.c:114)
==63146==    by 0x4B5EA65: __run_exit_handlers (exit.c:108)
==63146==    by 0x4B5EBAD: exit (exit.c:138)
==63146==    by 0x4B411D0: (below main) (libc_start_call_main.h:74)
==63146==
==63146== HEAP SUMMARY:
==63146==     in use at exit: 91,992 bytes in 4 blocks
==63146==   total heap usage: 13,846 allocs, 13,842 frees, 4,189,224 bytes allocated
==63146==
==63146== LEAK SUMMARY:
==63146==    definitely lost: 0 bytes in 0 blocks
==63146==    indirectly lost: 0 bytes in 0 blocks
==63146==      possibly lost: 0 bytes in 0 blocks
==63146==    still reachable: 91,992 bytes in 4 blocks
==63146==         suppressed: 0 bytes in 0 blocks
==63146== Rerun with --leak-check=full to see details of leaked memory
==63146==
==63146== For lists of detected and suppressed errors, rerun with: -s
==63146== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Из выведенной информации видно, что ошибок и утечек памяти в программе нет.

Вывод

В ходе лабораторной работы я приобрёл практические навыки использования утилит valgrind и gprof, а также научился применять их для анализа и оптимизации работы программ. Я смог на практике оценить производительность и выявить узкие места в коде, что значительно улучшило моё понимание процессов профилирования и отладки.