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: А. С. Бычков
Преподаватель: Н. К. Макаров
Группа: М8О-201Б

Дата:
Оценка:
Подпись:

Москва, 2024

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).

1 Описание

1 gprof

Согласно, [1] «gprof» подсчитывает количество времени, потраченное на каждую процедуру. Далее эти времена распространяются по краям графа вызовов. Циклы обнаруживаются, и выполняются вызовы цикла для разделения времени цикла.

В результате анализа доступны несколько форм результатов.

Плоский профиль показывает, сколько времени ваша программа потратила на каждую функцию и сколько раз эта функция вызывалась. Если вы просто хотите знать, какие функции сжигают больше всего циклов, это кратко изложено здесь.

Граф вызовов показывает для каждой функции, какие функции ее вызывали, какие другие функции она вызывала и сколько раз. Также есть оценка того, сколько времени было потрачено на подпрограммы каждой функции. Это может подсказать места, где вы можете попытаться исключить вызовы функций, отнимающие много времени.

Аннотированный исходный список представляет собой копию исходного кода программы, помеченную количеством выполнений каждой строки программы.

2 valgrind

Согласно, [2] «valgrind» — гибкая программа для отладки и профилирования исполняемых файлов Linux. Он состоит из ядра, которое обеспечивает синтетический процессор в программном обеспечении, и ряда инструментов отладки и профилирования. Архитектура является модульной, поэтому новые инструменты можно создавать легко и без нарушения существующей структуры.

2 Исходный код

1 gprof

Первым делом создадим профилировочный файл, для этого используем созданный в прошлой работе, бенчмарк, но удалим из него использование $std :: map$, таким образом, эта программа будет просто делать миллион случайных действим с красно-чёрным деревом.

```
g++ -pg -Wall -Werror -Wpedantic -Wextra benchmark.cpp
./a.out >/dev/null
gprof ./a.out >profile
```

Получим следующий вывод:

% time	cumulative seconds	self seconds	calls	name
18.18	0.06	0.06	1333395	RB::RB::find
12.12	0.10	0.04	-	main
10.61	0.14	0.04	29379617	bool std::operator< (string const&, string const&)
7.58	0.16	0.03	29379617	operator<(pair<string, uint64_t> const&, pair<string, uint64_t> const&)
6.06	0.18	0.02	8013049	operator>(pair<string, uint64_t> const&, pair<string, uint64_t> const&)

Здесь отражена только часть функций, которая делает самый большой вклад, а так же, их сигнатуры немного изменены для удобочитаемости.

Как видно, самый часто выполняемые функции - $find$ и операторы сравнения пар и строк. Легко объянить, почему так происходит. Функция $find$ у меня вызывается как и при поиске, так и при в ставке и при удалении. Именно эта функция спускается по дереву, чтобы найти, где необходимый элемент. Так же, эта функция в свою очередь сравнивает пары ключ-значение, а из сравнения пар вытекает сравнение строк.

Так же, много времени затратилось на $main$ из-за того, что в нём происходит считывание миллиона строк.

Никаких оптимизаций в плане скорости не нужно, т.к. эта версия работает уже быстрее $std :: map$.

2 valgrind

Уменьшим количество входных строк до 10000, т.к. «valgrind» сильно замедляет работу программы. Затем запустим программу для проверки на утечки памяти:

```
g++ -Wall -Werror -Wpedantic -Wextra benchmark.cpp
valgrind --leak-check=full --show-leak-kinds=all ./a.out
==129462== Memcheck,a memory error detector
==129462== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==129462== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==129462== Command: ./a.out
==129462==
Time RB: 190 ms
==129462==
==129462== HEAP SUMMARY:
==129462==     in use at exit: 0 bytes in 0 blocks
==129462==   total heap usage: 2,840 allocs,2,840 frees,3,425,640 bytes allocated
==129462==
==129462== All heap blocks were freed --no leaks are possible
==129462==
==129462== For lists of detected and suppressed errors,rerun with: -s
==129462== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как видно, никаких утечек памяти нет.

3 Выводы

С утилитой «valgrind» я был знаком раньше, знал, что с ее помощью можно находить утечки памяти, но когда начал читать про нее *manpage*, понял, что её применение куда шире, например - профилирование.

Про профилирование до этой лабораторной работы я ничего не знал. Я смог познакомиться с новым инструментом - «gprof» и концепцией профилирования. Теперь я буду часто использовать её, т.к. очень полезно знать, что же у тебя в программе занимает больше всего времени, чтобы по возможности соптимизировать эту часть кода.

Кроме того, я убедился, что моё красно-черное дерево работает верно, в нём нет утечек памяти, а большую часть работы программы выполняется процедура $find$, которая используется у меня везде.

Список литературы

[1] *gprof - manual page.*
URL: `https://linux.die.net/man/1/gprof` (дата обращения: 28.04.2024).

[2] *valgrind - manual page.*
URL: `https://linux.die.net/man/1/valgrind` (дата обращения: 28.04.2024).