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. О. Дубинин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2018

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 выплонения 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

Профилирование позволяет изучить, как программа ведет себя и как расходует своё время, какие функции вызывали другие функции, пока программа исполнялась. Это позволяет выяснить почему программа работает медленно, либо выявить ошибки. Также по итогам профилирования можно составить список участков кода – кандидатов на оптимизацию: в него войдут долго исполняющиеся и часто исполняющиеся.

При этом стоит учитывать, что внедрение профилятора в программу может повлиять на скорость её выполнения. Если разница в выполнении недопустима высока или необходимо высокоточное определение времени исполнения, стоит использовать инструменты, которые не влияют на выполнение программы, а наблюдают выполнение уже готовой и скомпилированной.

2 GPROF

Название расшифровывается как GNU Profiler. Данная утилита позволяет отобразить профильную статистику, которая накапливается приложением во время работы.
Чтобы использовать gprof необходимо:

- использовать аргумент «-pg» при сборке проекта;
- запустить исполняемый файл (создаётся файл gmon.out);
- использовать утилиту, передав в параметры имя исполняемого файла и gmon.out с флагом «-p».

Данные gprof на тестах:
Функции выполняющиеся дольше всего

```
Flat profile:

Each sample counts as 0.01 seconds.
time   name
42.88  TTree::LoadSub(_IO_FILE*,TNode*&)
28.59  TTree::SaveSub(_IO_FILE*,TNode*)
14.29  TTree::CreateNode(TData)
7.15   ToLower(char)
7.15   TTree::DeleteNode(TNode*&)
```

Первый столбец цифр обозначает какое количество времени в процентах от общего времени выполнения выполнялась та или иная функция.
Количество вызовов функций

```
Each sample counts as 0.01 seconds.
calls      us/call  us/call  name
1880241     0.01     0.01  ToLower(char)
1877709     0.00     0.00  IsLetter(char)
845161      0.05     0.05  TTree::CreateNode(TData)
144730      0.00     0.00  TTree::GetHeight(TNode*)
46128       0.00     0.00  TTree::BalFact(TNode*)
26237       0.00     0.00  TTree::SetHeight(TNode*)
22791       0.00     0.00  TTree::Balance(TNode*)
7491        0.00     0.00  TTree::Search(TData&)
7482        0.00     0.00  TTree::SearchSub(TNode*&,TData&)
2505        0.00     0.05  TTree::Insert(TData)
```

```
2504          0.00     0.05  TTree::InsertSub(TNode*,TData)
1290          7.76     7.76  TTree::DeleteNode(TNode*&)
1289         15.52    46.48  TTree::LoadSub(_IO_FILE*,TNode*&)
1289          0.00    54.24  TTree::Load(char const*)
1218         57.50    57.50  TTree::SaveSub(_IO_FILE*,TNode*)
1218          0.00    57.50  TTree::Save(char const*)
882           0.00     0.00  TTree::RotateLeft(TNode*)
841           0.00     0.00  TTree::RotateRight(TNode*)
```

Вставка осуществлялась 2505 раз, а удаление 1290 раза. При этом операция преобразования символа в нижний регистр была использована 1880241 раз.

Исходя из этих данных наиболее приоритетными для оптимизации являются операции работы с с преобразованием символа, т.к. они вызываются очень часто, и по-возможности повышение эффективности операций работы с диском, как самая трудоёмкая часть программы.

3 VALGRIND

Valgrind – средство профилирования, предназначенное для отладки использования памяти, обнаружения утечек памяти. С помощью инструментов Valgrind идентифицирует ошибки памяти и потоков, а также позволяет своевременно обнаружить переполнение стека и архивов. Дополнительно можно создать профиль кэша и памяти кучи с целью определения факторов, которые помогут улучшить скорость работы приложения и уменьшить использование памяти. Valgrind анализирует работу приложения, выполняя его не на основном процессоре, а на его симуляторе, точно определяет задействованные процессы и отправляет собранную статистику в заданный файл. Время работы Valgrind зависит от используемых инструментов, но обычно занимает значительно большее время – в 4 и более раз дольше по сравнению с выполнением кода напрямую.

Valgring включает следующие опции.

- Memcheck – выявление ошибок при работе с памятью.
- Cachegrind – профилировщик кэш-памяти.
- Callgrind – профилировщик кода.
- Massif – профилировщик «кучи» (heap).
- Helgrind – анализ функционирования многопоточных приложений, в т.ч. на «состояния гонок» (race conditions).

В процессе написания лабораторных работ наиболее часто я использую флаги −−tool=memcheck и его более подробный аналог −−leak-check

```
art@mars:~/study/semester_3/DA/lab_2$ valgrind --tool=memcheck ./lab2 <tests/01.t
>/dev/null
==6828== Memcheck,a memory error detector
==6828== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==6828== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6828== Command: ./lab2
==6828==
==6828==
==6828== HEAP SUMMARY:
==6828==     in use at exit: 0 bytes in 0 blocks
==6828==   total heap usage: 19,972 allocs,19,972 frees,3,584,473 bytes allocated
==6828==
==6828== All heap blocks were freed --no leaks are possible
```

```
==6828==
==6828== For counts of detected and suppressed errors,rerun with: -v
==6828== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Хотя ошибок не обнаружено проведем более тщательно тестирование.

```
art@mars:~/study/semester_3/DA/lab_2$ valgrind --leak-check=full ./lab2 <tests/01.t
>/dev/null
==6845== Memcheck,a memory error detector
==6845== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==6845== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6845== Command: ./lab2
==6845==
==6845==
==6845== HEAP SUMMARY:
==6845==     in use at exit: 0 bytes in 0 blocks
==6845==   total heap usage: 19,972 allocs,19,972 frees,3,584,473 bytes allocated
==6845==
==6845== All heap blocks were freed --no leaks are possible
==6845==
==6845== For counts of detected and suppressed errors,rerun with: -v
==6845== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ошибок не обнаружено: сколько памяти выделено, столько и освобождено. Однако, количество выделенной памяти очень велико. Почему программа настолько прожорлива. Дело в том, что valgrind в первую очередь ищет возможные утечки и ошибки ценой потери производительности. Программы, запущенные с помощью valgrind используют больший объём памяти (за счет выделения значительных дополнительных расходов памяти). Более того, valgrind, аналогично более старому методу – библиотеке dmalloc, заменяет стандартное выделение памяти языка C++ собственной реализацией, которая, помимо прочего, включает в себя защиту памяти (memory guards) вокруг всех выделенных блоков.

Также стоит учитывать, что HEAP SUMMARY это общая сумма всех выделенных байт, т.е. в процессе выполнения программы они не были использованы одновременно.

4 Дневник отладки

1. 01.11.18; 14:25; Тщательно ознакомился с gprof. Изучил конкретный пример его использования.

2. 02.11.18; 15:45; Произвел тестирование с помощью valgrind. Так как этот инструмент мне знаком, и я его часто использую, тестирование производилось сразу. Результаты оказались впечатляющими.

3. 02.11.18; 16:12; Ознакомился с дополнительными возможностями valgrind, такими как Helgrind, поскольку моя программа работает линейно и последовательно, гонки потоков наблюдатся никак не может.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я изучил методы профилирования и применил их на практике. Профилирование программ является неотъемлемой частью разработки качественных проектов. Каждый профайлер помогает отлаживать ошибки в определённой области. Поэтому следует использовать несколько профайлеров, конкретно:

- valgrind помогает бороться с утечками, и ошибками во время исполнения. При сложной структуре программы, как в этом конкретном случае – сложная структура данных с частым выделением памяти, он оказывает неоценимую помощь в отладке.

- gprof анализирует сколько раз программа использовала какую-либо функцию, а также время, потраченое на её исполнение. Идеально подходит для ускорения программы, и когда программа не укладывается во временные рамки, обращает внимание на необходимость реализации получения бита за константу.

На мой взгляд, этих утилит достаточно для написания хорошего и, главное, рабочего кода. Мои текущие требования к отладке покрываются этими программами.

Список литературы

[1] *Профилирование грограм в среде UNIX*
URL: `http://www.opennet.ru/docs/RUS/gprof/gprof-4.html` (дата обращения: 20.10.2018).

[2] *Valgrind – Википедия*
URL: `URL:https://ru.wikipedia.org/wiki/Valgrind` (дата обращения: 21.10.2018).

[3] *Valgrind*
URL: `http://valgrind.org/docs/manual/` (дата обращения: 21.10.2018).

[4] *Примеры использованя Callgrind*
URL: `https://unicorn.ejudge.ru/instr.pdf` (дата обращения: 28.10.2018).