

Лабораторная работа № 3 по курсу дискретного анализа: Исследование качества программ

Выполнил студент группы М8О-208Б-20 МАИ *Каширин Кирилл*.

Условие

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:

1. Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
2. Выводов о найденных недочётах.
3. 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`).

Метод решения

Для исследования потребления памяти будем использовать утилиту `Valgrind`. Это инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, проверки потокобезопасности, а также профилирования. Наиболее используемым инструментом в этой утилите является `Memcheck`. Проблемы, которые может обнаружить `Memcheck`, включают в себя:

1. Попытки использования неинициализированной памяти
2. Чтение/запись в память после её освобождения
3. Чтение/запись за границами выделенного блока
4. Утечки памяти

Для отображения профильной статистики, которая накапливается во время приложения используем утилиту `gprof`. Профилирование позволяет понять, где программа расходует свое время и какие функции вызывали другие функции, пока программа

исполнялась. Эта информация может указать на ту часть программы, которая выполняется медленнее, чем ожидалось. Эту часть можно в первую очередь оптимизировать, если это возможно.

Для того, чтобы проверить, что тесты охватывают проверку всех функций кода, необходимо проверить покрытие кода с помощью утилиты `gcov`. Если в отчете после выполнения программы утилита показывает, что некоторые строки кода не использовались, то либо в этих строках может быть ошибка (например, в условии и тогда, либо эти строки лишние, либо тест не смог охватить этот случай в коде). `Lcov` — графический интерфейс для `gcov`. Он собирает файлы `gcov` для нескольких файлов с исходниками и создает комплект HTML страниц с кодом и сведениями о покрытии.

Дневник отладки

Пропишем в консоли `valgrind --leak-check=full ./a.out` для запуска утилиты `valgrind`. Флаг `--leak-check=full` включает функцию обнаружения утечек памяти.

```
kirill@LAPTOP-F153AKTP:~/da/lab2$ valgrind --leak-check=full ./a.out < test.txt
```

```
=437== Memcheck, a memory error detector
==437== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==437== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==437== Command: ./a.out
==437==
==437==
==437== HEAP SUMMARY:
==437==    in use at exit: 122,880 bytes in 6 blocks
==437==    total heap usage: 152 allocs, 146 frees, 203,595 bytes allocated
==437==
==437== LEAK SUMMARY:
==437==    definitely lost: 0 bytes in 0 blocks
==437==    indirectly lost: 0 bytes in 0 blocks
==437==    possibly lost: 0 bytes in 0 blocks
==437==    still reachable: 122,880 bytes in 6 blocks
==437==    suppressed: 0 bytes in 0 blocks
==437== Reachable blocks (those to which a pointer was found) are not shown.
==437== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==437==
==437== For counts of detected and suppressed errors, rerun with: -v
==437== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Можно заметить, что в `ERROR SUMMARY` не было обнаружено какой-либо ошибки. В `LEAK SUMMARY` прописано о том, что осталось 122,880 bytes in 6 blocks категории `still reachable`. Память, относящийся к этой категории не относится к утечки памяти, эта категория означает, что блоки не были освобождены, но они могли бы быть теоретически освобождены, потому что программа все еще отслеживала указатели на

эти блоки памяти. После использования флага `-show-reachable=yes` отчёт показал, что это из-за функции `ios_base::sync_with_stdio(false);`, которая отключает синхронизацию `iostreams` с `stdio`.

После исправления ошибки ещё раз запустим `valgrind`

```
kirill@LAPTOP-F153AKTP:~/da/lab2$ valgrind --leak-check=full ./a.out < a.txt
```

```
==5657== Memcheck, a memory error detector
==5657== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5657== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==5657== Command: ./a.out
==5657==
==5657==
==5657== HEAP SUMMARY:
==5657==       in use at exit: 0 bytes in 0 blocks
==5657==   total heap usage: 148 allocs, 148 frees, 85,835 bytes allocated
==5657==
==5657== All heap blocks were freed -- no leaks are possible
==5657==
==5657== For counts of detected and suppressed errors, rerun with: -v
==5657== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как видим, утечек памяти больше не обнаружено.

Для отображения профильной статистики, которая накапливается во время приложения используем утилиту `gprof`: Профилирование состоит из нескольких шагов:

1. Компилирование программы с флагом профилирования: `g++ -pg main.cpp`
2. Исполнение программы для порождения файла данных о профиле `./a.out`
3. Запуск `'gprof'` для анализа данных о профиле. `gprof a.out gmon.out`

Flat profile:

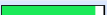
Each sample counts as 0.01 seconds.

%	cumulative	self	self	total		
time	seconds	seconds	calls	us/call	us/call	name
28.58	0.02	0.02	26730	0.75	1.12	BTree::SearchNode
14.29	0.06	0.01	6699	1.49	1.49	BTree::InsertNode
14.29	0.07	0.01				main
0.00	0.07	0.00	2592597	0.00	0.00	bool operator!=
0.00	0.07	0.00	314173	0.00	0.00	bool std::operator<
0.00	0.07	0.00	78665	0.00	0.00	bool std::operator>
0.00	0.07	0.00	59389	0.00	0.00	BTree::BinarySearchInNode
0.00	0.07	0.00	40019	0.00	0.00	bool std::operator==
0.00	0.07	0.00	37419	0.00	0.00	DictPair::DictPair()
0.00	0.07	0.00	37419	0.00	0.00	DictPair::~~DictPair()
0.00	0.07	0.00	25797	0.00	0.00	DictPair::operator=

0.00	0.07	0.00	7679	0.00	0.00	BNode::BNode()
0.00	0.07	0.00	7679	0.00	0.00	BNode::~~BNode()
0.00	0.07	0.00	6699	0.00	2.62	BTree::Insert(DictPair&)
0.00	0.07	0.00	7	0.00	0.00	BTree::LoadFile
0.00	0.07	0.00	4819	0.00	0.00	operator<
0.00	0.07	0.00	10	0.00	0.00	BTree::SaveFile
0.00	0.07	0.00	3834	0.00	0.00	BTree::SplitChild
0.00	0.07	0.00	13	0.00	0.00	BTree::DeleteFromNode
0.00	0.07	0.00	3	0.00	11.60	BTree::Deleting
0.00	0.07	0.00	1	0.00	0.00	BTree::DeleteTree(BNode*)
0.00	0.07	0.00	1	0.00	0.00	BTree::BTree()
0.00	0.07	0.00	1	0.00	0.00	BTree::~~BTree()
0.00	0.12	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZgtR8DictPairS0_

Можно заметить, что большинство вызовов приходится на сравнение нод словаря, поскольку оно происходит почти во всех функциях, реализованных BTree. Также большинство времени приходится на функцию поиска ноды в словаре. Это объясняется тем, что перед тем как вставить или удалить ноду, нужно проверить, есть ли эта нода в словаре.

Для исследования покрытия кода используем утилиту gcov: для начала скомпилируем программу с флагом -coverage, который нужен для анализа покрытия кода. После запустим программу, чтобы в процессе работы она записала информацию о фактическом покрытии кода на данном запуске. Чтобы было удобнее посмотреть результаты покрытия, с помощью lscov сгенерируем отчёт в виде HTML-страницы.

Directory	Line Coverage ↕		Functions ↕	
/home/kicill/da/lab2		92.1 %	339 / 368	100.0 % 23 / 23

Исходя из отчета, установлено, что покрытие кода составляет 92%, что является хорошим показателем. Непокрытыми остались блоки функций, которые обрабатывают ошибки, а также один блок условия при удалении элемента из BTree. Этот блок оказался лишним и был удален.

Выводы

В результате этой лабораторной работы я познакомился с утилитами valgrind, grof, gcov, которые позволяют отлаживать программы и оптимизировать их. Набор этих сведений, полученных этими утилитами, дают подробные сведения о программе, о её недостатках. Утилита valgrind помогла мне найти утечку памяти и исправить её, grof выполнил профилирование кода, а gcov показал покрытие кода.