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: Д.В. Семенов
Преподаватель: А.А. Кухтичев
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №3

Задача: Необходимо провести исследование скорости выполнения и потребления оперативной памяти в лабораторной работе №2. В случае выявления ошибок - исправить их.

Минимальный набор используемых средств должен содержать утилитуgprof и библиотеку dmalloc, однако их можно заменять на любые другиеаналогичные или более развитые утилиты (например, Valgrind или Shark)или добавлять к ним новые (например, gcov).

Вариант используемых программ: Valgrind, gprof, gdb.

1 Описание

Результатом лабораторной работы является отчёт, состоящий из:

1. Дневника выплонения работы, в котором отражено что и когда делалось,
2. Какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
3. Выводов о найденных недочётах.
4. Сравнение работы исправленной программы с предыдущей версией.
5. Общих выводов о выполнении лабораторной работы, полученном опыте.

2 Дневник выполнения работы

Основные этапы написания кода:

1. Реализация первоначальной идеи
2. Выявление ошибок работы программы (как логическая структура, так и работа с памятью)
3. Тестирование программы на корректность работы
4. Тест производительности программы, выявление медленно работающих функций
5. Оптимизация

3 Используемые средства

1 GDB

В процессе работы над программой приходилось очень часто обращаться к дебагеру gdb. Несмотря на его изначально "пугающий"консольный вид, новичку разобраться в нем достаточно просто. Многие ошибки и баги было невозможно отследить просто пристольно прочитывая код и прокручивая ситуации у себя в голове, тут приходил на помощь gdb. Для начала нужно скомпилировать программу с ключом -g, чтобы программа собралась с отладочными символами.

Основные команды для работы: break, s (step), n (next), c (continue), backtrace и др.

Благодаря использованию gdb я существенно сократил время отладки и разработки программы. Его потенциал очень большой (GDB User Manual занимает более 700 страниц), так что теперь я всем советую пользоваться программой.

2 Valgrind

Valgrind - это необходимый инструмент для отслеживания утечек памяти в программах. Сразу приведу несколько примеров, где он меня спасал.

Например, я непавильно обработал границы строки, и когда пропустил программу через эту утилиту, она мне быстро и точно показала, что я неправильно работаю со строкой.

```
1  Node(char *key, unsigned long long value){
2            int i = 0;
3            if(key != nullptr){
4                int len_key = 0;
5                len_key = strlen(key);
6                m_key = (char*)malloc((len_key+1) * sizeof(char));
7                for(i = 0; i < len_key; i++){
8                    m_key[i] = key[i];
9                }
10           }
11           m_key[i] = '\0'; // valgrind lenkey+1
12           m_value = value;
13           color = RED;
14       };
```

Или другой пример. Здесь я случайно использовал функцию free() два раза на одной и той же памяти.

```
1  ~Node(){
2            free(m_key); // valgrind 1- to free, 2 - in deletehidden
3
4        };
```

Valgring умеет находить утечки памяти, может найти недопустимое использование указателя, использование неинициализированных переменных и еще много неприятных моментов. Это одна из самых мощных утилит профилирования программ, которые мне встречались. Также в ее состав входят другие программы (massif - анализ выделения памяти различными частями программы, callgrind - анализ вызова функций, построение дерева функций)

3 gprof

Ну а без gprof ни одна моя программа не прошла бы checker. Чтобы gprof заработал, необходимо скомпилировать программу с ключом -pg, а затем запустить программу

без gprof. Будет создан двоичный файл gmon.out. и теперь можно запустить программу через gprof. Полученный текстовый файл profile вполне читаемый — видно, где и сколько времени проводила программа.

time	seconds	seconds	calls	us/call	us/call	name
60.05	0.03	0.03	5061	5.93	5.93	$Node :: Node(char*, unsignedlonglong)$
20.02	0.04	0.01	4939	2.03	2.03	$TRBTree :: Find(char*, bool*)$
20.02	0.05	0.01				$GetKey(char*)$
0.00	0.05	0.00	5061	0.00	0.00	$TRBTree :: FixInsertRBTree(Node*)$
0.00	0.05	0.00	5061	0.00	5.93	$TRBTree :: Insert_kv(char*, unsignedlonglong)$
0.00	0.05	0.00	4939	0.00	2.03	$TRBTree :: Delete_k(char*)$
0.00	0.05	0.00	1491	0.00	0.00	$TRBTree :: RotateLeft(Node*)$
0.00	0.05	0.00	1490	0.00	0.00	$TRBTree :: RotateRight(Node*)$

А также граф передачи управления

```
 Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 19.98% of 0.05 seconds

index % time self children called name
                                                 <spontaneous>
[1] 80.0 0.00 0.04 main [1]
                0.00 0.03 5061/5061 TRBTree::Insert_kv(char*, unsigned long long) [3]
                0.00 0.01 4939/4939 TRBTree::Delete_k(char*) [5]
-----------------------------------------------
                0.03 0.00 5061/5061 TRBTree::Insert_kv(char*, unsigned long long) [3]
[2] 60.0 0.03 0.00 5061 Node::Node(char*, unsigned long long) [2]
-----------------------------------------------
                0.00 0.03 5061/5061 main [1]
[3] 60.0 0.00 0.03 5061 TRBTree::Insert_kv(char*, unsigned long long) [3]
                0.03 0.00 5061/5061 Node::Node(char*, unsigned long long) [2]
                0.00 0.00 5061/5061 TRBTree::FixInsertRBTree(Node*) [13]
-----------------------------------------------
```

С помощью этой утилиты я убрал ненужные метоты get и set, которые в нашем случае только загромождали код, и вместо них напрямую обращался к данным. Также в 5 лабораторной понял, что формирование строки "на лету"при обходе дерева - неудачный вариант - программа тратит слишком много времени, вместо этого я решил сохранять путь в deque.

4 Выводы

Утилиты профилирования кода невероятно помогают ускорить время разработки программ и повысить их качество. С их помощью намного удобнее и легче искать сбои в работе программ, устранять утечки памяти и больше разбираться в том, как работает код. Я понял, что в современной разработке программного обеспечения нельзя обойтись без вспомогательных программ.

Список литературы

[1] GPROF - https://eax.me/c-cpp-profiling/

[2] Valgrind - http://alexott.net/ru/linux/valgrind/Valgrind.html

[3] GDB - https://eax.me/gdb/