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: М. А. Бронников
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2019

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

Для отладки работы с памятью мы будем пользоваться утилитой **valgrind**. Как сказано в [2]: «Valgrind — инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, а также профилирования.» Наиболее интересным из инструментов Valgrind является Memcheck. «Проблемы, которые может обнаружить Memcheck, включают в себя:

- попытки использования неинициализированной памяти,
- чтение/запись в память после её освобождения,
- чтение/запись за границами выделенного блока,
- утечки памяти

Ценой этого является потеря производительности.»[2]

В профилировании различают 2 основных метода:

1. *Определяющее профилирование.* «Целевая программа модифицируется (чаще всего перекомпилируется с определенными флагами - -pg для gcc). - в код внедряются вызовы библиотеки профилирования, собирающие информацию о выполняемой программе и передающие ее профилировщику»[1]. Утилита **gprof**.
2. *Статистическое профилировании.* « исполнение программы периодически (100-1000 раз в секунду) останавливается, управление передается профилировщику, который анализирует текущий констекст исполнения. Затем, используя отладочную информацию или таблицы символов из исполняемого файла программы и загруженных библиотек, определяется исполняемая в текущий момент функция и, если возможно, последовательность вызовов в стеке»[1]. Утилита **perf**.

2 Журнал отладки

Из основных событий разработки следует выделить следующие:

1. *10.03.2019 в 21:49* — проверка программы утилитой **valgrind** показала утечку памяти в программе:

```
max@max-X550CC:~/DA/lab2$ valgrind --leak-check=full ./prog <file >hello
==3304== Memcheck,a memory error detector
==3304== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==3304== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==3304== Command: ./prog
==3304==
==3304==
==3304== HEAP SUMMARY:
==3304==     in use at exit: 9,019,920 bytes in 53,021 blocks
==3304==   total heap usage: 109,419 allocs,56,398 frees,18,629,146 bytes
allocated
==3304==
==3304== 248 bytes in 1 blocks are possibly lost in loss record 1 of 4
==3304==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-l
==3304==    by 0x1092FA: Tree_Insert (in /home/max/DA/lab2/prog)
==3304==    by 0x10B663: main (in /home/max/DA/lab2/prog)
==3304==
==3304== 9,019,120 (8,098,788 direct,920,332 indirect) bytes in 47,663
blocks are definitely lost in loss record 4 of 4
==3304==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-l
==3304==    by 0x1092FA: Tree_Insert (in /home/max/DA/lab2/prog)
==3304==    by 0x10B663: main (in /home/max/DA/lab2/prog)
==3304==
==3304== LEAK SUMMARY:
==3304==    definitely lost: 8,098,788 bytes in 47,663 blocks
==3304==    indirectly lost: 920,332 bytes in 5,356 blocks
==3304==      possibly lost: 248 bytes in 1 blocks
==3304==    still reachable: 552 bytes in 1 blocks
==3304==         suppressed: 0 bytes in 0 blocks
==3304== Reachable blocks (those to which a pointer was found) are not
shown.
==3304== To see them,rerun with: --leak-check=full --show-leak-kinds=all
==3304==
```

```
==3304== For counts of detected and suppressed errors,rerun with: -v
==3304== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Проблема оказалась в том, что я забыл освободить память по указателю при удалении узла дерева, после исправления функции удаления ошибка была устранена:

```
max@max-X550CC:~/DA/lab2$ gcc prog.c -o prog
max@max-X550CC:~/DA/lab2$ valgrind --leak-check=full ./prog <file >hello
==4264== Memcheck,a memory error detector
==4264== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==4264== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==4264== Command: ./prog
==4264==
==4264==
==4264== HEAP SUMMARY:
==4264==     in use at exit: 552 bytes in 1 blocks
==4264==   total heap usage: 109,418 allocs,109,417 frees,18,607,172 bytes
allocated
==4264==
==4264== LEAK SUMMARY:
==4264==    definitely lost: 0 bytes in 0 blocks
==4264==    indirectly lost: 0 bytes in 0 blocks
==4264==      possibly lost: 0 bytes in 0 blocks
==4264==    still reachable: 552 bytes in 1 blocks
==4264==         suppressed: 0 bytes in 0 blocks
==4264== Reachable blocks (those to which a pointer was found) are not
shown.
==4264== To see them,rerun with: --leak-check=full --show-leak-kinds=all
==4264==
==4264== For counts of detected and suppressed errors,rerun with: -v
==4264== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

2. *11.03.2019 в 00:52* — Функция вставки в дерево работала излишне долго. Вот что показывал **gprof** на тестовом файле на 110000 добавлений и удалений.

```
max@max-X550CC:~/DA/lab2$ gprof prog gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
```

```
%   cumulative   self              self     total
time   seconds   seconds    calls  ns/call  ns/call  name
34.49      0.10     0.10   110000   909.38   909.38  Tree_Insert
34.49      0.20     0.10                             main
17.25      0.25     0.05   110000   454.69   454.69  Search_Tree
13.80      0.29     0.04   110000   363.75   363.75  Tree_Delete
0.00       0.29     0.00    52967     0.00     0.00  Left_Rotate
0.00       0.29     0.00    52902     0.00     0.00  Right_Rotate
0.00       0.29     0.00        2     0.00     0.00  Destroy_Tree
0.00       0.29     0.00        1     0.00     0.00  Load_Tree
0.00       0.29     0.00        1     0.00     0.00  Save_Tree
```

Однако после полного переписывания алгоритма на новый, я получил следующие данные на том же самом тестовом файле:

```
max@max-X550CC:~/DA/lab2$ gcc -pg prog.c -o prog
max@max-X550CC:~/DA/lab2$ ./prog <file >hello
max@max-X550CC:~/DA/lab2$ gprof prog gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ns/call  ns/call  name
40.01      0.10     0.10                             main
28.01      0.17     0.07   110000   636.57   636.57  Tree_Insert
20.01      0.22     0.05   110000   454.69   454.69  Search_Tree
12.00      0.25     0.03   110000   272.82   272.82  Tree_Delete
0.00       0.25     0.00    53507     0.00     0.00  Left_Rotate
0.00       0.25     0.00    53452     0.00     0.00  Right_Rotate
0.00       0.25     0.00        2     0.00     0.00  Destroy_Tree
0.00       0.25     0.00        1     0.00     0.00  Load_Tree
0.00       0.25     0.00        1     0.00     0.00  Save_Tree
```

3. *11.03.2019 в 14:23* — Однако перед тем как полностью переписать функцию встаки я решил проверить какую модель обхода дерева использовать лучше: на стэке или рекурсией. Для этого я решил сравнить при помощи утилиты **perf** производительность алгоритмов поиска при помощи рекурсии и стэка.
Для рекурсии я получил следующий отчет(*привожу отрывок*):

```
Children      Self  Command  Shared Object      Symbol
```

```
+   70,26%     0,00%  prog     [unknown]          [.] 0x1256258d4c544155
+   70,26%     0,00%  prog     libc-2.27.so       [.] __libc_start_main
+   50,99%    24,54%  prog     prog               [.] main
+   22,81%    22,81%  prog     libc-2.27.so       [.] malloc_consolidate
+   22,81%     0,00%  prog     [unknown]          [.] 0x00005599dda23340
+   10,94%    10,75%  prog     prog               [.] Tree_Insert
+   10,50%     9,39%  prog     libc-2.27.so       [.] getchar
+   10,05%     9,98%  prog     libc-2.27.so       [.] __strcmp_sse2_unaligned
+    8,39%     2,18%  prog     prog               [.] Search_Tree
```

Когда как для стэкового прохода я получил:

```
Children      Self  Command  Shared Object      Symbol
+   86,28%     0,00%  prog     [unknown]          [.] 0x1246258d4c544155
+   86,28%     0,00%  prog     libc-2.27.so       [.] __libc_start_main
+   78,42%    13,87%  prog     prog               [.] main
+   34,92%    34,92%  prog     prog               [.] Tree_Delete
+   21,04%    21,03%  prog     prog               [.] Tree_Insert
+   11,84%    11,84%  prog     libc-2.27.so       [.] cfree@GLIBC_2.2.5
+    6,84%     6,84%  prog     libc-2.27.so       [.] __strcmp_sse2_unaligned
+    3,08%     2,70%  prog     libc-2.27.so       [.] tolower
+    2,81%     2,81%  prog     [kernel.kallsyms]  [k] nmi
+    1,59%     0,22%  prog     libc-2.27.so       [.] __isoc99_scanf
+    1,45%     0,02%  prog     prog               [.] tolower@plt
+    1,36%     1,36%  prog     libc-2.27.so       [.] _IO_vfscanf
+    1,32%     1,32%  prog     libc-2.27.so       [.] getchar
+    0,96%     0,76%  prog     libc-2.27.so       [.] _int_malloc
+    0,92%     0,92%  prog     prog               [.] Search_Tree
```

Как видно, процент остановок на функции поиска в стэковом варианте меньше.
Хоть, возможно, оценивать обход в глубину при помощи сравнения реализаций алгоритмов поиска не совсем корректно, я на основе этих данных принял решение использовать стэковую модель.

3 Выводы

Выполнив *Лабораторную работу №3* по курсу «Дискретный анализ», я научился граммотнее отлаживать свои программы при помощи современных утилит и средств. Некоторыми из них я пользовался и ранее, однако некоторые функции мне были незнакомы, а с большинством я познакомился впервые.
Я узнал как можно быстро и качественно проанализировать свой код на утечки памяти, скорость работы, выходы за пределы массивов. Для этого мне понадобилось изучить некоторые особенности работы утилит, прочитать несколько стататей, которые помогли разобраться в отчетах средств профилирования и средств работы с памятью.

По итогу я получил огромный опыт, который изменил мой взягляд на процесс оценки и отладки программ, а ведь отладка— один из важнейших процессов разработки. Я убежден, что знания, полученные мной, в далнейшем очень сильно пригодятся, ведь мне еще не раз придется писать сложный код со строгими требованиями к его характеристиками, оценка которых без современных средств отладки составляет невероятно сложную и трудоемкую задачу.

Список литературы

[1] *perf : современный Linux профилировщик.*
URL: http://koder-ua.blogspot.com/2012/03/perf-linux.html?m=1 (дата обращения: 16.02.2019).

[2] *Valgrind — Википедия*
URL: https://ru.wikipedia.org/wiki/Valgrind(дата обращения 17.02.2019).

[3] *Ускорение кода при помощи GNU-профайлера*
URL: https://www.ibm.com/developerworks/ru/library/l-gnuprof/(дата обращения 17.02.2019).