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: П. А. Харьков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

Задача: Необходимо провести исследование скорости выполнения и потребления оперативной памяти в программе по лабороторной работе 2. В случае выявления ошибок или явных недочётов, требуется их исправить.

1 Описание

Во время написания программы могут возникнуть ошибки, которые не будут видны во время работы программы, но после из-за них могут начаться проблемы. К примеру, если не освобождать память в куче, может произойти утечка памяти, которая, скопившись в большом объёме, может вызвать нехватку памяти для работы самой программы. Или же какая-то часть кода будет медленно работать из-за частых вызовов какой-нибудь функции или бесполезной работы. Это существенно снижает степень взаимодействия с программой.
Профилирование программы является одним из важнейших этапов написания программы. И, к счастью, для этого существует множество средств, таких как: утилита gprof, библиотека dmalloc, утилиты valgrind и shark, и многие другие. Я буду рассматривать только две утилиты: gprof и valgrind. Я буду использовать gprof для замера времени работы каждой функции и работы моей программы в целом, а также посмотрю на граф вызовов функций, чтобы проверить, есть ли какая-то программа, которая вызывалась много раз, чтобы обратить внимание на её оптимизацию. А valgrind я буду использовать для того, чтобы найти утечки памяти при работе программы и все ошибки, связанные с управлением памятью.

2 Дневник выполнения работы

Проверим работоспособность нашей программы на маленьком тесте, состоящем из 7 строк, который покрывает все возможные операции с деревом:

```
p.kharkov$ cat tests/small
+ x 6
+ c 63
c
+ k 10
-k
! Save res
! Load res
p.kharkov$  ./laba2 <tests/small
OK
OK
OK: 63
OK
OK
OK
OK
```

Программа отработала правильно и не вызвала никаких ошибок.

Затем проведём исследование на утечки памяти в программе. Это мы будем делать, используя утилиту valgrind. Тестировать на наличие утечки сначала будем на том же тесте, на котором проверяли работоспособность программы.

```
p.kharkov$ valgrind ./laba2 <tests/small
==1136== Memcheck,a memory error detector
...
==1136== Conditional jump or move depends on uninitialised value(s)
==1136==    at 0x10B008: TTree::VTree::Insert(char*,unsigned long long&) (in
laba2)
==1136==    by 0x10A658: main (in laba2)
==1136==
==1136== Conditional jump or move depends on uninitialised value(s)
==1136==    at 0x10ADFE: TTree::VTree::InsertFixTree(TTree::VNode*) (in laba2)
==1136==    by 0x10B0F0: TTree::VTree::Insert(char*,unsigned long long&) (in
laba2)
...
==1136== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 0 from 0)
```

Оказалось, что в нашей программе были найдены целых 8 ошибок при работе с памятью! Узнать подробнее об ошибках мы можем, добавив ключ –track-origins=yes, благодаря которому мы выясним, где именно происходит ошибка.

```
p.kharkov$ valgrind --track-origins=yes ./laba2 <tests/small
==1150== Memcheck,a memory error detector
...
==1150== Conditional jump or move depends on uninitialised value(s)
==1150==    at 0x10B008: TTree::VTree::Insert(char*,unsigned long long&) (in
laba2)
==1150==    by 0x10A658: main (in laba2)
==1150==  Uninitialised value was created by a heap allocation
==1150==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux
-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1150==    by 0x10B066: TTree::VTree::Insert(char*,unsigned long long&) (in
laba2)
==1150==    by 0x10A658: main (in laba2)
...
```

Поняв, что ошибка происходит в функции Insert при создании неинициализированной переменной в куче, мы с легкостью находим её и устраняем. Теперь снова вернемся к исследованию на утечки памяти в программе:

```
p.kharkov$ valgrind ./laba2 <tests/small
==1175== Memcheck,a memory error detector
...
==1175== HEAP SUMMARY:
==1175==     in use at exit: 618 bytes in 3 blocks
==1175==   total heap usage: 13 allocs,10 frees,93,098 bytes allocated
==1175==
==1175== LEAK SUMMARY:
==1175==    definitely lost: 304 bytes in 1 blocks
==1175==    indirectly lost: 304 bytes in 1 blocks
==1175==      possibly lost: 0 bytes in 0 blocks
==1175==    still reachable: 10 bytes in 1 blocks
==1175==         suppressed: 0 bytes in 0 blocks
==1175==
==1175== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как мы можем заметить не вызвалось ни одной ошибки, так что исправлена была правильная ошибка. Но теперь мы видим, что в нашей программе действительно существуют утечки: definitely lost означает, что на какую-то область памяти нет ни

одного указателя; indirectly lost означает, что существует указатель на эту область памяти, но мы никак не можем взаимодействовать с самим указателям; still reachable означает, что мы выделили память в куче, указатель на память остался, но память не была освобождена. Возможно, в нашем случае это значит, что узел-родитель был потерян, а из-за этого был потерян узел-ребёнок. Чтобы узнать более подробную информацию об утечках мы добавим ключ –leak-check=full, который выводит более подробную информацию об утечка, и ключ –show-leak-kinds=all, который показывает все виды утечек.

```
p.kharkov$ valgrind --leak-check=full --show-leak-kinds=all ./laba2 <tests/small
==1177== Memcheck,a memory error detector
...
==1177==
==1177== 10 bytes in 1 blocks are still reachable in loss record 1 of 3
==1177==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
vgpreload_memcheck-amd64-linux.so)
==1177==    by 0x10AB78: __static_initialization_and_destruction_0(int,int)
(in laba2)
==1177==    by 0x10AB99: _GLOBAL__sub_I__Z7ToLowerPcS_ (in laba2)
==1177==    by 0x10C18C: __libc_csu_init (in laba2)
==1177==    by 0x4A7603F: (below main) (libc-start.c:264)
...
```

Рассмотрим первую ошибку: это утечка still reachable, то есть мы выделили в куче память, сохранили указатель на неё, но не освободили. Как мы можем понять, это глобальная переменная, так как, если была бы локальной, то указатель был бы потерян при завершении функции. Найдя её переходим к утечкам directly и indirectly lost.

```
==1177== 608 (304 direct,304 indirect) bytes in 1 blocks are definitely lost
in loss record 3 of 3
==1177==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64
-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1177==    by 0x10B066: TTree::VTree::Insert(char*,unsigned long long&) (in
laba2)
==1177==    by 0x10A658: main (in laba2)
```

Как мы видим, они связаны между собой, то есть, скорее всего был удален указатель на узел дерева, а из-за этого был потерян указатель на его ребенка. К сожалению, мы можем увидеть только выделение памяти в куче, а не место, где был потерян указатель, так что необходимо проверить все строчки, где мы освобождаем память

или можем потерять указатель на ребёнка. Успешно найдя ошибку, переходим к следующему этапу.

После того, как мы устранили все ошибки и утечки по памяти, проведём исследование скорости выполнения программы. Тест состоит из 10^5 образцов на поиск, вставку и удаления узлов в красно-чёрное дерево и исследовать будем с помощью утилиты gprof. Для того, чтобы использовать её, необходимо скомпилировать нашу программу с ключом -pg, а затем запустить скопмилированную программу и передать ей тестовый файл на вход. Это необходимо, чтобы создался файл gmon.out, в котором хранится вся необходимая информация для работы утилиты.

```
p.kharkov$ make
g++ -std=c++14 -pg -pedantic -Wall -Wextra -c tree.cpp -o tree.o
g++ -std=c++14 -pg -pedantic -Wall -Wextra -c sltree.cpp -o sltree.o
g++ -std=c++14 -pg -pedantic -Wall -Wextra main.cpp tree.o sltree.o -o laba2
p.kharkov$ ./laba2 <tests/01.t >/dev/null
```

Сначала выясним сколько раз вызывалась каждая функция и сколько в сумме она работала. Если какая-то функция работала долго, то мы постараемся её оптимизировать. Для того, чтобы получить таблицу с данными по времени используем утилиту gprof с ключом -p:

```
p.kharkov$ gprof laba2 -p
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
88.37      0.15     0.15     3370    44.58    44.58  TTree::VTree::Insert(char*,
unsigned long long&)
5.89      0.16     0.01    10000     1.00     1.00  ToLower(char*,char*)
5.89      0.17     0.01                             main
0.00      0.17     0.00     3342     0.00     0.00  TTree::VTree::Delete(char*)
0.00      0.17     0.00     3288     0.00     0.00  TTree::VTree::Search(char*)
0.00      0.17     0.00     2260     0.00     0.00  TTree::VNode::VNode()
...
```

Получив таблицу, мы можем заметить, что 88% процессорного времени работала функция для вставки узла в красно-чёрное дерево. На ее работу было потрачено 0.15 секунды, и в среднем за вызов - 45 микросекунд, в то время как другие функции в сумме работали менее 0.01 секунды.
Для того, чтобы узнать, в какой функции вызывалась функция вставки узла, мы можем посмотреть на стек вызовов с помощью grprof с ключом -q:

```
p.kharkov$ gprof laba2 -q
Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 5.87% of 0.17 seconds

index % time    self  children    called     name
<spontaneous>
[1]    100.0    0.01    0.16                 main [1]
0.15    0.00    3370/3370        TTree::VTree::Insert(char*,unsigned long long&)
[2]
0.01    0.00   10000/10000       ToLower(char*,char*) [3]
0.00    0.00    3342/3342        TTree::VTree::Delete(char*) [10]
0.00    0.00    3288/3288        TTree::VTree::Search(char*) [11]
0.00    0.00       1/1           TTree::VTree::VTree() [24]
0.00    0.00       1/1           TTree::VTree::~VTree() [25]
-----------------------------------------------
0.15    0.00    3370/3370        main [1]
[2]     88.2    0.15    0.00    3370         TTree::VTree::Insert(char*,
unsigned long long&) [2]
0.00    0.00    2260/2260        TTree::VNode::VNode() [12]
0.00    0.00    2260/2260        TTree::VTree::InsertFixTree(TTree::VNode*)
[13]
-----------------------------------------------
0.01    0.00   10000/10000       main [1]
[3]      5.9    0.01    0.00   10000         ToLower(char*,char*) [3]
...
```

Как мы можем заметить именно в функции main вызывалась функция вставки 3370 из 3370, то есть все разы. Также мы можем заметить, что функции, которые вызывает функция Insert, работали меньше 0.01 секунды в сумме. Значит в теле самой функции, находится код, замедляющий работу программы.
Благодаря этому анализу я выявил, где программа медленно работает, и исправил это.

3 Выводы о найденных недочётах

Ошибки, которые я допустил в программе могут быть допущены любым человеком из-за невнимательности. К примеру, такой является неинициализированная переменная или неосвобождённая память в куче. И из-за этих ошибок при больших данных могли возникнуть непредвиденный никем ошибки. А также неэффективная работа моей функции Insert сильно замедляла мою программу.
Но, к счастью, проверив свою программу с помощью утилит gprof и valgrind я смог исправить эти недочёты. Программа стала работать гораздо быстрее и не образовывались никакие утечки в памяти.

4 Сравнение работы исправленной программы с предыдущей версии

В начале на 7 образцах запросов у меня было 8 ошибок на неправильное управление с памятью и на выходе в куче не было освобождено 618 байт памяти:

```
==1150== HEAP SUMMARY:
==1150==     in use at exit: 618 bytes in 3 blocks
==1150==   total heap usage: 13 allocs,10 frees,93,098 bytes allocated
==1150==
==1150== LEAK SUMMARY:
==1150==    definitely lost: 304 bytes in 1 blocks
==1150==    indirectly lost: 304 bytes in 1 blocks
==1150==      possibly lost: 0 bytes in 0 blocks
==1150==    still reachable: 10 bytes in 1 blocks
==1150==         suppressed: 0 bytes in 0 blocks
==1150==
==1150== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 0 from 0)
```

После всех исправлений, результат работы valgrind выглядит так, то есть не возникло ошибок при работе с памятью и все утечки были закрыты:

```
==1227== HEAP SUMMARY:
==1227==     in use at exit: 0 bytes in 0 blocks
==1227==   total heap usage: 12 allocs,12 frees,93,088 bytes allocated
==1227==
==1227== All heap blocks were freed --no leaks are possible
==1227==
==1227== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Также до исправлений программа на 10^5 запросах работала в сумме 0.17 секунды, так как функция Insert работала медленно:

```
%   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
88.37      0.15     0.15     3370    44.58    44.58  TTree::VTree::Insert(char*,
unsigned long long&)
5.89      0.16     0.01    10000     1.00     1.00  ToLower(char*,char*)
5.89      0.17     0.01                             main
0.00      0.17     0.00     3342     0.00     0.00  TTree::VTree::Delete(char*)
0.00      0.17     0.00     3288     0.00     0.00  TTree::VTree::Search(char*)
...
```

После того, как я оптимизировал работу функции Insert программа отработала за 0.01 секунду, а функция Insert работала меньше 0.01 секунды:

```
%   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
100.15      0.01     0.01                             main
0.00      0.01     0.00    10000     0.00     0.00  ToLower(char*,char*)
0.00      0.01     0.00     3370     0.00     0.00  TTree::VTree::Insert(char*,
unsigned long long&)
0.00      0.01     0.00     3342     0.00     0.00  TTree::VTree::Delete(char*)
0.00      0.01     0.00     3288     0.00     0.00  TTree::VTree::Search(char*)
0.00      0.01     0.00     2260     0.00     0.00  TTree::VNode::VNode()
...
```

5 Выводы

В мире существует огромное количество утилит для профилирования программы и каждая из них по своему полезна. Нельзя рассмотреть программу только на утечки в памяти или только на скорость её работы: необходимо комбинировать анализ программы с помощью различных утилит, так как каждая может показать детали о неточностях, непоказанных ранее другими утилитами. И обязательно нужно проверять программу, так как нельзя признать её завершённой, если есть хоть какие-нибудь ошибки, ведь они могут возникнуть при длительной работе программы.
Выполнив третью лабораторную работу по курсу «Дискретный анализ», я научился исследовать свой код на утечки памяти и скорость работы программы с помощью утилит. Как мне кажется, это важнейшая лабораторная работа, так как оптимизация и исследование на ошибки пригодится мне в любой программе, независимо от языка программирования и задачи. Также было интересно проверить, насколько я был внимателен при написании программы в таких вещах, как выделение с освобождением памяти и в оптимизации функций.

Список литературы

[1] *Утилита valgrind*
URL: `https://valgrind.org/docs/manual/mc-manual.html` (дата обращения 01.12.2020).

[2] *Утилита gprof*
URL: `https://www.opennet.ru/man.shtml?topic=gprof` (дата обращения 01.12.2020).