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

Выполнил студент группы М08-207Б-18 МАИ *Скворцов Кирилл.*

Условие

1) 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 выполнения 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 недочётов. Сравнение работы исправленной программы с предыдущей версии. Общих выводов о выполнении лабораторной работы, полученном опыте. Минимальный набор используемых средст должен содержать утилиту gprof и библиотеку dmalloc, однако их можно заменять на любые другие аналогичные или более известные утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov).

2) Вариант лабораторной работы №2: 1 (AVL tree).

Дневник отладки

Инструментирование

1) Valgrind

Это утилита, предназначенная для отладки использования памяти, обнаружения утечек памяти, а также профилирования. Название «valgrind» взято из германоскандинавской мифологии, где обозначает название главного входа в Вальгаллу. Утилита состоит из ядра, которое обеспечивает эмуляцию процессора и ряда различных вспомогательных модулей, каждый из которых является инструментом для отладки или профилирования. В состав пакета Valgrind входит множество инструментов:

1) Инструмент по умолчанию (и наиболее используемый) — Memcheck. Вокруг почти всех инструкций Memcheck вставляет дополнительный код инструментирования, который отслеживает законность и адресуемость операций с памятью.
2) Addrcheck — облегченная версия Memcheck, работающая гораздо быстрее и потребляющая меньше памяти, но и обнаруживающая меньшее количество типов ошибок. Этот инструмент был удален в версии 3.2.0.
3) Massif — профилировщик кучи.

4) Helgrind и DRD — инструменты, способные отслеживать состояние гонки и подобные ошибки в многопоточном коде.
5) Cachegrind — профилировщик кэша.
6) Callgrind — профилировщик кода, может использовать графический интерфейс KCacheGrind.
7) SGCheck — экспериментальный инструмент для поиска схожих ошибок по аналогии с memcheck, но с тем отличием, что ищет ошибки в стеке, а не в куче.

Отладка с помощью valgrind:

Самые частые ошибки, возникающие у в моей программе, которые выявлял valgrind, это:

1.

==5128== Conditional jump or move depends on uninitialised value(s)
==5128== at 0x4C32D08: strlen (in /usr/lib/valgrind/vgpreload$_m emcheck - amd64 - linux.so$)
== 5128 == $by 0x10A91A : TNode :: TNode(char*, unsignedlonglong, int)(AVL.cpp : 17)$
== 5128 == $by 0x10A1B5 : DoReadFile(std :: basic_ifstream < char, std :: char_traits < char >>, bool)(AVL.cpp : 283)$
== 5128 == $by 0x10A6F3 : main(AVL.cpp : 356)$

Эта ошибка была связана с тем, что при выделении новой памяти для считывания ключа из файла следующей строчкой кода проверялось, была она выделена (т.е. обработка возможной ошибки) для этого массива. Если убрать эту проверку (и все аналогичные), то valgrind не будет считать это некорректной работой, т.к. не будет иметь значение, проинициализированна память или нет. Исправление - инициализация выделенной памяти.

2.

==5400== HEAP SUMMARY:
==5400== in use at exit: 122,922 bytes in 8 blocks
==5400== total heap usage: 19 allocs, 11 frees, 195,850 bytes allocated
==5400==
==5400== LEAK SUMMARY:
==5400== definitely lost: 40 bytes in 1 blocks
==5400== indirectly lost: 2 bytes in 1 blocks
==5400== possibly lost: 0 bytes in 0 blocks
==5400== still reachable: 122,880 bytes in 6 blocks
==5400== suppressed: 0 bytes in 0 blocks

Эта ошибка связана с тем, что память, выделенная динамически, не очищалась. Она выделялась с помощью new в рекурсивной функции, но после перехода на следующий уровень рекурсии или по возвращении из нее, не очищалась. Следовательно, терялось

n блоков памяти (где n - число вершин дерева). В данном примере показана ошибка, возникающая при считывании словаря из файла.

3.

==3727== Invalid read of size 1

==3727== at 0x4C32D04: strlen (in /usr/lib/valgrind/vgpreload$_m emcheck - amd64 - linux.so$)

==3727== by 0x10A9BA: TNode::TNode(char*, unsigned long long, int) (AVL.cpp:17)

==3727== by 0x10A23E: DoReadFile(std::basic$_i fstream < char, std :: char_t raits < char >>, bool)(AVL.cpp : 283)

==3727== by 0x10A792: main (AVL.cpp:356)

==3727== Address 0x5ba0531 is 0 bytes after a block of size 1 alloc'd

==3727== at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload$_m emcheck-amd64 - linux.so$)

==3727== by 0x10A154: DoReadFile(std::basic$_i fstream < char, std :: char_t raits < char >>, bool)(AVL.cpp : 267)

==3727== by 0x10A792: main (AVL.cpp:356)

Даная ошибка возникала при некорректном считывании данных из файла. Валгринд выводит ошибку, которая означает, что для считывания переменной используется память, которая не была до этого выделена, очищена или как-либо использовалась. Размер этой памяти - 1 байт, соответственно, решение - выделять больше памяти на 1 байт больше при считывании какого-то типа данных (в моем случае это увеличение размера массива символов для ключа).

2) Address Sanitizer

AddressSanitizer (или Asean) — это инструмент программирования с открытым исходным кодом от Google. В отличие от valgrind'а инструмент использует другую идею — инструментирование на этапе компиляции. Address sanitizer состоит из модуля инструментирования в компиляторе и библиотеки времени выполнения.

Ошибки, которые он определяет:
Использование памяти после освобождения
Переполнение буфера кучи
Переполнение буфера стека
Глобальное переполнение буфера
Некорректное использование после выполнения функций
Ошибки порядка инициализации
Утечка памяти

Пример ошибки, которую помог выявить address sanitizer:

```
=================================================================
==4199==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000011 at pc 0
x55d6e5440019 bp 0x7ffe7a89bc50 sp 0x7ffe7a89bc40
WRITE of size 1 at 0x602000000011 thread T0
 #0 0x55d6e5440018 in TNode::TNode(char*, unsigned long long, int) /home/fefso/diskran/2_lab/AVL.cpp:20
 #1 0x55d6e543daa1 in Insert(TNode*, char*, unsigned long long) /home/fefso/diskran/2_lab/AVL.cpp:92
 #2 0x55d6e543f7e3 in main /home/fefso/diskran/2_lab/AVL.cpp:316
 #3 0x7ff9b9fb9b96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
 #4 0x55d6e543d3f9 in _start (/home/fefso/diskran/2_lab/a.out+0x23f9)

0x602000000011 is located 0 bytes to the right of 1-byte region [0x602000000010,0x602000000011)
allocated by thread T0 here:
    #0 0x7ff9baa0a618 in operator new[](unsigned long) (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe0618)
    #1 0x55d6e543ff2e in TNode::TNode(char*, unsigned long long, int) /home/fefso/diskran/2_lab/AVL.cpp:18
    #2 0x55d6e543daa1 in Insert(TNode*, char*, unsigned long long) /home/fefso/diskran/2_lab/AVL.cpp:92
    #3 0x55d6e543f7e3 in main /home/fefso/diskran/2_lab/AVL.cpp:316
    #4 0x7ff9b9fb9b96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
```

Суть ее заключалась в том, что при запуске конструктора, который выделяет память под новую вершину, выделялось меньше памяти, чем требовалось для корректной инициализации. При копировании ключей происходило обращение к неинициализированному, не используемому никак куску памяти. Простыми словами, шло обращение за границы массива char.

Профилирование

3) Gprof

Gprof — это инструмент анализа производительности для приложений Unix. Он использовал гибрид инструментирования и выборки, и был создан как расширенная версия более старого инструмента "prof".
В код внедряются вызовы библиотеки профилирования (куски кода, собирающие информацию о времени ее выполнения). Наиболее часто замеряется время выполнения каждой функции.

Очевидным минусом гарантированного профилирования является необходимость перекомпиляции программы, что не удобно, а иногда и невозможно (если нет исходных файлов). Также гарантированное профилирование значительно влияет на скорость исполнения программы, делая получение её реального времени исполнения затруднительным. Пример работы моей программы с тестовыми данными размером 10^6 :

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 36.39      0.04     0.04   100000     0.00     0.00  DoToLower(char*)
 18.19      0.06     0.02   100000     0.00     0.00  Insert(TNode*, char*, unsigned long long)
  9.10      0.07     0.01  9563704     0.00     0.00  GetHeight(TNode*)
  9.10      0.08     0.01  3127871     0.00     0.00  GetBalance(TNode*)
  9.10      0.09     0.01    99523     0.00     0.00  TNode::TNode(char*, unsigned long long, int)
  9.10      0.10     0.01    99523     0.00     0.00  TNode::~TNode()
  9.10      0.11     0.01        2     5.00    10.01  DeleteTree(TNode*)
  0.00      0.11     0.00  1653981     0.00     0.00  CulcHeight(TNode*)
  0.00      0.11     0.00  1653981     0.00     0.00  Max(int, int)
  0.00      0.11     0.00  1555811     0.00     0.00  DoBalance(TNode*)
  0.00      0.11     0.00    24623     0.00     0.00  LeftTurn(TNode*)
  0.00      0.11     0.00    24462     0.00     0.00  RightTurn(TNode*)
  0.00      0.11     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z9DoToLowerPc
  0.00      0.11     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

4) Perf

Perf — инструмент для анализа производительности в ОС Linux. Команда userspace perf представляет собой простой в использовании интерфейс с такими командами, как:

perf stat: получение количества событий
perf record: запись событий для последующей отчетности
perf report: разбивка событий по процессам, функциям и т. д.
perf annotate: аннотирование сборки или исходного кода с помощью счетчиков событий
perf bench: запуск различных микрочастиц ядра

В моем случае используются record и report. Пример работы perf для программы из 2-й лабораторной:

```
fefso@fefso-H81M-HD3:~/diskran/2_lab$ c++ -pedantic -Wall -Werror -O2 -fno-omit-frame-pointer AVL.cpp -o avl
fefso@fefso-H81M-HD3:~/diskran/2_lab$ head -n 100000 test.txt | sudo perf record ./avl > /dev/null
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0,044 MB perf.data (821 samples) ]
fefso@fefso-H81M-HD3:~/diskran/2_lab$ sudo perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 821  of event 'cycles:ppp'
# Event count (approx.): 742799094
#
# Overhead  Command  Shared Object       Symbol
# ........  .......  ..................  ..................................................................................
#
    14.24%  avl      avl                 [.] Insert
     8.88%  avl      [kernel.kallsyms]   [k] do_syscall_64
     8.65%  avl      libc-2.27.so        [.] tolower
     6.53%  avl      libc-2.27.so        [.] __GI___strcmp_ssse3
     4.32%  avl      libstdc++.so.6.0.25 [.] std::operator>><char, std::char_traits<char> >
     4.28%  avl      libc-2.27.so        [.] malloc_consolidate
     4.12%  avl      avl                 [.] DoBalance
     3.97%  avl      [kernel.kallsyms]   [k] syscall_return_via_sysret
     3.82%  avl      avl                 [.] tolower@plt
     3.05%  avl      [kernel.kallsyms]   [k] entry_SYSCALL_64
     3.01%  avl      libstdc++.so.6.0.25 [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::
fefso@fefso-H81M-HD3:~/diskran/2_lab$
```

Благодаря perf можно выявить места программы, на которые работают довольно долго. Это позволит избежать TL на чекере и даст понимание человеку, пишущему код, требуется ли оптимизация или исправление части программы.
Отключим быстрый ввод ($std :: ios :: syncwithstdio(false)$) и убедимся в том, что он экономит время работы программы. В 1-й лабораторной, как и во 2-й без него программа может работать слишком долго.

```
fefso@fefso-H81M-HD3:~/diskran/2_lab$ c++ -pedantic -Wall -Werror -O2 -fno-omit-frame-pointer AVL.cpp -o avl
fefso@fefso-H81M-HD3:~/diskran/2_lab$ head -n 100000 test.txt | sudo perf record ./avl > /dev/null
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0,071 MB perf.data (1535 samples) ]
fefso@fefso-H81M-HD3:~/diskran/2_lab$ sudo perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 1K of event 'cycles:ppp'
# Event count (approx.): 1380731193
#
# Overhead  Command  Shared Object        Symbol
# ........  .......  ...................  .............................................
#
    17.62%  avl      libstdc++.so.6.0.25  [.] std::operator>><char, std::char_traits<char>
     9.10%  avl      libc-2.27.so         [.] _IO_sputbackc
     8.99%  avl      libc-2.27.so         [.] _IO_getc
     6.91%  avl      avl                  [.] Insert
     5.37%  avl      [kernel.kallsyms]    [k] do_syscall_64
     4.67%  avl      libc-2.27.so         [.] tolower
     4.43%  avl      libc-2.27.so         [.] __GI___strcmp_ssse3
     4.18%  avl      libc-2.27.so         [.] _IO_ungetc
     3.16%  avl      libstdc++.so.6.0.25  [.] __gnu_cxx::stdio_sync_filebuf<char, std::cha
     2.93%  avl      libstdc++.so.6.0.25  [.] __gnu_cxx::stdio_sync_filebuf<char, std::cha
     2.37%  avl      avl                  [.] tolower@plt
     2.27%  avl      libstdc++.so.6.0.25  [.] getc@plt
     2.14%  avl      [kernel.kallsyms]    [k] syscall_return_via_sysret
     1.92%  avl      avl                  [.] DoBalance
```

Невооруженным глазом видно, что ввод экономит несколько процентов от общего времени программы.

Выводы

Изучение инструментов данной лабораторной работы заняло у меня больше времени, чем написание прошлой лабораторной, для которой эти инструменты, по-хорошему, надо было применять при отладке. Действенную пользу я извлек только из инструмента valgrind. Использование остальных никак не ускорило бы написание или отладку моей программы. Если бы вместо лабораторной работы был написан проект, который гораздо больше по объему и сложнее в реализации, тогда бы точно возникла потребность в других инструментах, но в этой valgrind'а было достаточно. Однако, я открыл для себя такую мощную вещь, как Perf. Он, действительно, наглядно показывает время работы каждой из частей программы, и, вопреки, возможно, мнению других, прост в использовании. В совокупности 2 и 3 лабораторные работы научили меня тому, что

нужно отлаживать свою программу после каких-либо, даже самых малых, изменений. При этом выбор инструмента отладки - выбор пишущего, но хоть какие-то должны быть, т.к. они значительно упрощают поиск ошибок.