Лабораторная работа № 3 по курсу дискретного анализа: инструментирование и профилирование

Выполнил студент МАИ группы М8О-201Б *Ефимов Александр.*

Инструментирование

При разработке кода использование обычных выводов и assert'ов чаще всего бывает недостаточны. В частности, большой объем работы с динамической памятью становится сложным из-за отсутствия возможности проверить принадлежность адреса программе (если адрес, лежащий в указателе, выделен программе) или потери адресов по ошибке программиста. Для поиска ошибок, краевых случаев, утечек памяти и оценки производительности используется так называемое инструментирование - отладка программы с помощью инструментов.

1. **Valgrind**

   Один из самых интересных инструментов linux для отладки работы с динамической памятью является *valgrind*. Перед запуском при помощи ключа *–tool* запускается выбранный инструмент (memcheck, callgrind, cachegrind и т.д.), но без него по умолчанию запускается *memcheck* - инструмент для проверки кода во время выполнения программы. Он позволяет отловить такие ошибки, как использование неинициализированных переменных, двойные освобождения, запись в память, невыделенную программе, утечку памяти и т.д.

   Пример обрезанного вывода ошибки с флагами *–leak-check=full –show-leak-kinds=all* для сломанной версии программы второй лабораторной (commit e1aec657d3d42773cb0476a19012c3ce2ff6e7a5), получающей неверные указатели в результате нарушения алгоритма функции слияния:

```
==8609== Process terminating with default action of signal 11 (SIGSEGV)
==8609==  Access not within mapped region at address 0x14D4
==8609==    at 0x10A925: TBTreeNode::Delete(char*) (in /home/rookstar/
    Git/DA-Lab-2/solution)
==8609==    by 0x10A935: TBTreeNode::Delete(char*) (in /home/rookstar/
    Git/DA-Lab-2/solution)
==8609==    by 0x10A935: TBTreeNode::Delete(char*) (in /home/rookstar/
    Git/DA-Lab-2/solution)
==8609==    by 0x10A935: TBTreeNode::Delete(char*) (in /home/rookstar/
    Git/DA-Lab-2/solution)
==8609==    by 0x10AF1B: TBTree::Delete(char*) (in /home/rookstar/Git/
    DA-Lab-2/solution)
==8609==    by 0x109A55: main (in /home/rookstar/Git/DA-Lab-2/solution)
==8609==  If you believe this happened as a result of a stack
```

```
==8609==  overflow in your program's main thread (unlikely but
==8609==  possible), you can try to increase the size of the
==8609==  main thread stack using the --main-stacksize= flag.
==8609==  The main thread stack size used in this run was 8388608.
```

Здесь происходит обращение по указателю к области памяти, невыделенной программе.

Его минусами являются большое испоьзование CPU (valgrind требует дополнительную память для хранения информации о состоянии памяти), замедление программы в 30-50 раз, а также обнаружение ошибок, связанных не с программмой, а с машиной, на которой valgrind работает (например, обнаружение неинициализированных переменных в виртуальной машине).

Результат инструментирования лабораторной 2 с помощью valgrind:

```
==3943== LEAK SUMMARY:
==3943==    definitely lost: 0 bytes in 0 blocks
==3943==    indirectly lost: 0 bytes in 0 blocks
==3943==      possibly lost: 0 bytes in 0 blocks
==3943==    still reachable: 122,880 bytes in 6 blocks
==3943==         suppressed: 0 bytes in 0 blocks
==3943==
==3943== For counts of detected and suppressed errors, rerun with: -v
==3943== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Где 6 блоков отведены на поток в результате работы функции, отвязывающего поток iostream от stdio (они освобожадются только после окончания программы).

2. **Address Sanitize**

Другим инструментом для отладки работы с памятью является *Address Sanitizer* (или коротко *ASan*). В отличие от valgrind, который работает с бинарным кодом (динамическая инструментация), ASan пользуется инструментированием на этапе компиляции (вводит отладочный код в программу). Вызывается он флагом *-fsanitize=address*. Инструментация на этапе компиляции требует перекомпилирования всего проекта, но в замен имеет меньшую нагрузку на CPU (замедляя программу всего лишь в 2-3 раза).

Также как и valgrind, ASan может находить утечку памяти, использование указателей после их освобождения, выход за пределы на куче. В отличие от него, ASan может находить выходы за пределы на стеке и части кода, которые могут не работать на других платформах, и не может находить использование неинициализированных переменных, из-за чего следует рассматривать их как два разных инструмента и использовать их вместе при отладки программы.

И Valgind, и ASan записывают память рабочей программы в дополнительную память (shadow memory), из-за чего Valgrind и ASan нельзя использовать одновременно - одна и та же область памяти резервируется обоими инструментами.

Пример вывода при выходе за пределы массива на стеке:

```
==7457==ERROR: AddressSanitizer: stack-buffer-overflow on address 0
    x7fff0977e344 at pc 0x55e45d87dd93 bp 0x7fff0977e300 sp 0
    x7fff0977e2f0
WRITE of size 4 at 0x7fff0977e344 thread T0
    #0 0x55e45d87dd92 in main /home/rookstar/Testing ground/test.cpp:6
    #1 0x7f3e2e37bb96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.
       so.6+0x21b96)
    #2 0x55e45d87dbf9 in _start (/home/rookstar/Testing ground/a.out+0
       xbf9)

Address 0x7fff0977e344 is located in stack of thread T0 at offset 52 in
    frame
    #0 0x55e45d87dce9 in main /home/rookstar/Testing ground/test.cpp:4

  This frame has 1 object(s):
    [32, 52) 'a' <== Memory access at offset 52 overflows this variable
HINT: this may be a false positive if your program uses some custom
    stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/rookstar/Testing
    ground/test.cpp:6 in main
Shadow bytes around the buggy address:
  0x1000612e7c10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1000612e7c60: 00 00 f1 f1 f1 f1 00 00[04]f2 00 00 00 00 00 00
  0x1000612e7c70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7c90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7ca0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000612e7cb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
```

```
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==7457==ABORTING
```

Здесь происходить выход за пределы массива int размера 5 (обращение по индексу 5). Используя списокк ниже таблицы, можно определить, что два байта до [04] содержат в себе первые четыре элемента массива, а последний элемент содержится в указанном байте, при этом происходит обращение к невыделенной части памяти (второй половине байта).

При работе с программой 2 лабораторной ASan ничего не выдал, что сигнализирует об отсутствии значительных ошибок.

3. **GDB**

GNU Debugger (коротко GDB) - это отладчик программы, позволяющий пошагово пройтись по программе, осмотреть ход выполнения функции, остановившейсь на ней с помощью breakpoint (при этом нужно компилитровать с таблицей символов флагом -g), просмотреть текущее состояние переменных и при отлове SEGFAULT. GBD может быть также подключен к графическим отладчикам.

Его минусом является работа не в режиме реального времени, что позволяет пропустить некоторые ошибки, в частности при работе с потоками или многопроцессорными программами.

Профилироване

Профилированием называется анализ программы во время ее работы для оценки ее сложности и производительности. Профилирование может происходить путем инструментирования (добавления кода, собирающего данные при работе программы) и/или сэмплирования (взятия пробы стэка вызовов через каждый определенный интервал). Сэиплирование позволяет программе работать быстрее за счет того, что добавление дополнительного кода не происходит, но менее точно, так как оценка производительности происходит через интервалы.

GProf

GNU Profiler (коротко *GProf)* - профилирующая программа, которая время работы функций программы в виде двух таблиц - Flat Profile и Call graph. Первая выводит простую таблицу, в которой показыватся, сколько времени заняла каждая функция, количество вызовов, количество миллисекунд за вызов от времени выполнения этой функции и от всего времени выполнения. Пример таблицы:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
42.12      0.08     0.08   127358     0.63     0.86  TBTreeNode::
    Delete(char*)
21.06      0.12     0.04    99999     0.40     0.50  TBTreeNode::
    Insert(char*, unsigned long long)
15.80      0.15     0.03   200000     0.15     0.15  ToLower(char*)
15.80      0.18     0.03    80507     0.37     0.37  TBTreeNode::
    MergePremature(int&)
 5.27      0.19     0.01    31260     0.32     0.32  TBTreeNode::Split
    (int)
 0.00      0.19     0.00   156385     0.00     0.00  TPair::TPair()
 0.00      0.19     0.00   100000     0.00     1.25  TBTree::Delete(
    char*)
 0.00      0.19     0.00   100000     0.00     0.65  TBTree::Insert(
    char*, unsigned long long)
 0.00      0.19     0.00    31269     0.00     0.00  TBTreeNode::
    TBTreeNode()
 0.00      0.19     0.00    27358     0.00     0.00  TBTreeNode::Merge
    (int, char*)
 0.00      0.19     0.00    14654     0.00     0.00  TBTreeNode::
    PredecessorKey()
 0.00      0.19     0.00     6293     0.00     0.00  TBTreeNode::
    SuccessorKey()
 0.00      0.19     0.00        8     0.00     0.00  TBTreeNode::
    TBTreeNode(TPair)
 0.00      0.19     0.00        1     0.00     0.00
    _GLOBAL__sub_I__Z7ToLowerPc
 0.00      0.19     0.00        1     0.00     0.00
    _GLOBAL__sub_I_main
 0.00      0.19     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
 0.00      0.19     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
 0.00      0.19     0.00        1     0.00     0.00  TBTreeNode::Wipe
    ()
 0.00      0.19     0.00        1     0.00     0.00  TBTree::Destroy()
 0.00      0.19     0.00        1     0.00     0.00  TBTree::TBTree()
 0.00      0.19     0.00        1     0.00     0.00  TBTree::~TBTree()
```

cumulative seconds показывает сумму выполнения функции с суммой выполнения всех выше стоящих по таблице функций.

Call graph показывает более подробное описания каждой функции, дополнительно разделяя время функции и время ее детей, подсчитывает количество рекурсивных вы-

зывов. В ней для каждой функции присваивается индекс и, при описании по индексу, показывает порядок вызова функции (т.е. какими функциями вызвается и какие функции вызывает сама).

Обрезанный пример результата второй лабораторной:

```
index % time    self  children    called     name
                                               <spontaneous>
[1]     100.0   0.00    0.19                 main [1]
                0.00    0.13  100000/100000      TBTree::Delete(char*)
                   [2]
                0.00    0.07  100000/100000      TBTree::Insert(char*,
                   unsigned long long) [5]
                0.00    0.00       1/1           TBTree::TBTree() [28]
                0.00    0.00       1/1           TBTree::~TBTree() [29]
-----------------------------------------------
                0.00    0.13  100000/100000      main [1]
[2]      65.8   0.00    0.13  100000           TBTree::Delete(char*) [2]
                0.08    0.03  100000/100000      TBTreeNode::Delete(
                   char*) <cycle 1> [4]
                0.02    0.00  100000/200000      ToLower(char*) [7]
-----------------------------------------------
[3]      57.9   0.08    0.03  100000+54716   <cycle 1 as a whole> [3]
                0.08    0.03  127358+636072      TBTreeNode::Delete(
                   char*) <cycle 1> [4]
                0.00    0.00   27358             TBTreeNode::Merge(int,
                    char*) <cycle 1> [18]
-----------------------------------------------
```

Выводы

Инструментирование программы позволяет значительно упростить процессы отладки и оптимизации программ, найти алгоритмические ошибки программ, выводя полную информацию о ее работе и анализируя возникающие ошибки при работе с памятью (неинициализированные переменные, потерянные указатели и т.д.).

Отладчики памяти помогают хранить информацию о выделенной памяти и обращениям к невыделенной, что достаточно сложно при обычной работе программы.

Профилирование помогает обнаружить места, в которым программа проводит больше всего времени, что упрощает процесс оптимизации программы (и даже места программы, в которых она не должна замедлятся и в которых она зависла).