Praktische Aspekte der Informatik

Moritz Mühlhausen
Prof. Marcus Magnor

https://graphics.tu-bs.de/teaching/ss19/padi/
Optimization

valgrind, gprof, and callgrind
Warning!
The following slides are meant to give you a very superficial introduction.

If you want to learn more, have a look at:

http://valgrind.org/info/tools.html
https://sourceware.org/binutils/docs/gprof
Outline

• Finding memory leaks with
  ▪ valgrind
  ▪ AddressSanitizer

• Optimizing performance with
  ▪ gprof
  ▪ callgrind
main.cpp

```cpp
int main(int argc, char** args) {

    // We create an array of numbers 0 - 19...
    int * mem = new int[20];
    for (int i = 0; i < 20; ++i)
        mem[i] = i;

    // ... but then we make a terrible mistake!
    mem = mem+1;
    for (int i = 0; i < 20; ++i)
        mem[i] = i;

}
```

https://graphics.tu-bs.de/teaching/ss19/padi/
• Valgrind
  ▪ Virtual CPU
  ▪ Detects areas of memory that are lost
  ▪ Several other tools in valgrind suite
  ▪ Linux only

• Simulation approach
  ▪ Quite accurate
  ▪ Very slow
  ▪ Interleaved multithreading
  ▪ Architecture dependent
1. Compile with \(-g\) \(-O0\) flags
   - \(-g\) produces debug information
   - \(-O0\) means no compiler optimizations

2. Run your program in \texttt{valgrind}
   \texttt{valgrind --leak-check=yes ./application}

3. Wait…

4. Decipher summary

https://graphics.tu-bs.de/teaching/ss19/padi/
#include <stdlib.h>

void f(void)
{
    int* x = (int*)malloc(10 * sizeof(int));
    x[10] = 0;    // problem 1: heap block overrun
    // problem 2: memory leak -- x not freed
}

int main(void)
{
    f();
    return 0;
}
==6384== Memcheck, a memory error detector
==6384== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==6384== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==6384== Command: ./Test
==6384==
==6384== Invalid write of size 4
==6384==  at 0x4007C7: f() (test.cpp:6)
==6384==  by 0x4007D8: main (test.cpp:11)
==6384==  Address 0x5204068 is 0 bytes after a block of size 40 alloc'd
==6384==  at 0x4C2DB8F: malloc (in /usr/lib/.../valgrind/vgpreload_memcheck-amd64-linux.so)
==6384==  by 0x4007BA: f() (test.cpp:5)=6384==  by 0x4007D8: main (test.cpp:11)

==6384==
==6384== HEAP SUMMARY:
==6384==     in use at exit: 40 bytes in 1 blocks
==6384==  total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==6384==
==6384== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6384==  at 0x4C2DB8F: malloc (in /usr/lib/.../valgrind/vgpreload_memcheck-amd64-linux.so)
==6384==  by 0x4007BA: f() (test.cpp:5)
==6384==  by 0x4007D8: main (test.cpp:11)

==6384== LEAK SUMMARY:
==6384==  definitely lost: 40 bytes in 1 blocks
==6384==  indirectly lost: 0 bytes in 0 blocks
==6384==  possibly lost: 0 bytes in 0 blocks
==6384==  still reachable: 0 bytes in 0 blocks
==6384==  suppressed: 0 bytes in 0 blocks
==6384==
==6384== For counts of detected and suppressed errors, rerun with: -v
==6384== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0))

https://graphics.tu-bs.de/teaching/ss19/padi/
1. Compile with `-g -fsanitize=address` flags
   - `-g` produces debug information
   - `-fsanitize=address` links AddressSanitizer

2. Run your program
   ```bash
   ./application
   ```
AddressSanitizer

=================================================================
==6541==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60400000dff8 at pc 0x00000004008be bp 0x7ffca129b4d0 sp 0x7ffca129b4c0
WRITE of size 4 at 0x60400000dff8 thread T0
  #0 0x4008bd in f() /.../test.cpp:6
  #1 0x4008cf in main /.../test.cpp:11
  #2 0x7f44d125582f in __libc_start_main (/lib/.../libc.so.6+0x2082f)
  #3 0x400798 in _start (/.../build/Test+0x400798)

0x60400000dff8 is located 0 bytes to the right of 40-byte region [0x60400000dfd0,0x60400000dff8) allocated by thread T0 here:
  #0 0x7f44d1697602 in malloc (/usr/lib/.../libasan.so.2+0x98602)
  #1 0x40087a in f() /.../test.cpp:5
  #2 0x4008cf in main /.../test.cpp:11
  #3 0x7f44d125582f in __libc_start_main (/lib/.../libc.so.6+0x2082f)

SUMMARY: AddressSanitizer: heap-buffer-overflow /.../test.cpp:6 f()
Shadow bytes around the buggy address:
  0x0c087fff9ba0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9bb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9bc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9bd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9be0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9bf0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9c00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9c10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9c20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9c30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c087fff9c40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  => 0x0c087fff9bf0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
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
  Heap right redzone: fb
  Freed heap region: fd
  Stack left redzone: f1
  Stack mid redzone: f2
  Stack right redzone: f3
  Stack partial redzone: f4
  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
==6541==ABORTING

https://graphics.tu-bs.de/teaching/ss19/padi/
Problem 1 fixed, run again:

=================================================================
==  8345 ==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:
  #0 0x7fd38d0e6602 in malloc (/usr/lib/.../libasan.so.2+0x98602)
  #1 0x40082a in f() /.../valgrind/test.cpp:5
  #2 0x40083a in main /.../valgrind/test.cpp:11
  #3 0x7fd38cca482f in __libc_start_main (/lib/.../libc.so.6+0x2082f)

SUMMARY: AddressSanitizer: 40 byte(s) leaked in 1 allocation(s).
1. Compile with \(-g\) \(-pg\) flags
   - \(-g\) produces debug information
   - \(-pg\) generates extra code for analysis with \texttt{gprof}

2. Run your program
   \texttt{./application}

3. Program executes; \texttt{gmon.out} is created

4. Run again in \texttt{gprof}
   \texttt{gprof -p ./application gmon.out}

5. Interpret the “flat profile”

https://graphics.tu-bs.de/teaching/ss19/padi/
Profiling with gprof

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
</tr>
<tr>
<td>37.88</td>
<td>2.11</td>
<td>2.11</td>
<td>100</td>
<td>21.13</td>
</tr>
<tr>
<td>36.62</td>
<td>4.16</td>
<td>2.04</td>
<td>100</td>
<td>20.43</td>
</tr>
<tr>
<td>25.67</td>
<td>5.59</td>
<td>1.43</td>
<td>100</td>
<td>14.32</td>
</tr>
</tbody>
</table>

• gprof samples with a certain frequency.  
  → We call each function 100 times to get “statistically valid” timings.

https://graphics.tu-bs.de/teaching/ss19/padi/
Profiling with callgrind

1. Compile with `-g -O0` flags

2. Run valgrind using the callgrind tool
   ```
   valgrind -tool=callgrind ./application
   ```

3. Program executes; `callgrind.out.*` is created.

4. Generate a textual overview
   ```
   callgrind_annotate callgrind.out.x
   ```
   or generate a graphical overview
   ```
   kcachegrind callgrind.out.x
   ```

https://graphics.tu-bs.de/teaching/ss19/padi/
Profiling with callgrind

kcache_grind

Diagram showing the profiling results with `kcache_grind`.