Praktische Aspekte der Informatik

Moritz Mühlhausen
Prof. Marcus Magnor
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
• Finding memory leaks with
  ▪ valgrind
  ▪ AddressSanitizer

• Optimizing performance with
  ▪ gprof
  ▪ callgrind
main.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;

}
• 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
Valgrind

1. Compile with \(-g\) \(-00\) flags
   - \(-g\) produces debug information
   - \(-00\) means no compiler optimizations

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

3. Wait...

4. Decipher summary
==419== Invalid write of size 4
==419== at 0x400926: main (in /home/valgrindsample/test)
==419== Address 0x5aa6cd0 is 0 bytes after a block of size 80 alloc'd
==419== at 0x4C2B800: operator new[](unsigned long) (in /usr/lib/...)
==419== by 0x40088E: main (in /home/valgrindsample/test)
==419==
==419== HEAP SUMMARY:
==419== in use at exit: 72,784 bytes in 2 blocks
==419== total heap usage: 2 allocs, 0 frees, 72,784 bytes allocated
==419==
==419== LEAK SUMMARY:
==419== definitely lost: 80 bytes in 1 blocks
==419== indirectly lost: 0 bytes in 0 blocks
==419== possibly lost: 0 bytes in 0 blocks
==419== still reachable: 72,704 bytes in 1 blocks
==419== suppressed: 0 bytes in 0 blocks
==419== Rerun with --leak-check=full to see details of leaked memory
==419==
==419== For counts of detected and suppressed errors, rerun with: -v
==419== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
1. Compile with \(-g \ -fsanitize=address\) flags
   - \(-g\) produces debug information
   - \(-fsanitize=address\) links AddressSanitizer

2. Run your program
   \(./application\)
AddressSanitizer

==32397==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60700000e000 at pc 0x400cfa bp 0x7ffeb6c77150 ... WRITE of size 4 at 0x60700000e000 thread T0
  #0 0x400cfa in main (/home/valgrindsample/test+0x400cfa)
  #1 0x7fcf29548ec4 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21ec4)
  #2 0x400ad8 (/home/valgrindsample/test+0x400ad8)

0x60700000e000 is located 0 bytes to the right of 80-byte region [0x60700000dfb0,0x60700000e000)
allocated by thread T0 here:
  #0 0x7fcf29cbd1bf in operator new[] (usr/lib/x86_64-linux-gnu/libasan.so.1+0x551bf)
  #1 0x400bbe in main (/home/valgrindsample/test+0x400bbe)
  #2 0x7fcf29548ec4 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21ec4)

SUMMARY: AddressSanitizer: heap-buffer-overflow ??:0 main
Shadow bytes around the buggy address:
0x0c0e7fff9bb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9bc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9bd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9be0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9bf0: fa fa fa fa fa fa 00 00 00 00 00 00 00 00 00 00 00 00 =]
0x0c0e7fff9c00: [fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9c10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9c20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9c30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9c40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0e7fff9c50: fa fa 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
Contiguous container OOB: fc
ASan internal: fe

==32397==ABORTING
1. Compile with \texttt{-g -pg flags}
   \begin{itemize}
   \item \texttt{-g} produces debug information
   \item \texttt{-pg} generates extra code for analysis with \texttt{gprof}
   \end{itemize}

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”
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.
1. Compile with `-g -00` 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
   ```
Profiling with callgrind

kcachegrind

Diagram showing the profiling of callgrind with a breakdown of various functions and their execution percentages.