Debugging mit GDB und Valgrind

Praktikum „C-Programmierung“

Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Eugen Betke, Nathanael Hübbe, Michael Kuhn, Jannek Squar

2019-11-25

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Motivation

Debugging

- Beispiel
- Verwendung von GDB

Valgrind

- Memcheck
- Callgrind
- Massif

Quellen

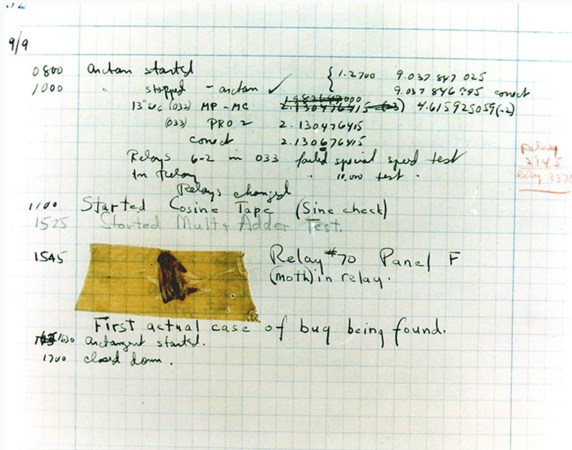


Abbildung 1: Dokumentation eines “echten” Computer-Bugs [Com47]

Bugs

- Debugging im HPC-Umfeld potentiell schwierig:
  - Komplexes Zusammenspiel: Rechenknoten, Netzwerk, Speicher, Software-Stack, etc.
  - Nichtdeterminismus
- Bug-Sorten:
  - Syntax-Error
  - Runtime-Error
- Problematisch, wenn Programm nicht abstürzt
- Bugfreiheit ist unentscheidbar

Motivation

Debugging

- Beispiel
- Verwendung von GDB

Valgrind

- Memcheck
- Callgrind
- Massif

Quellen

recursion.c

```
1  int testFunc(int i)
2  {
3      if(i > 0)
4      {
5          return testFunc(i-1)+i;
6      }
7      else if(i == 0)
8      {
9          return 0;
10     }
11 }
```

Erste Maßnahmen

- Mit `-Wall` kompilieren
- `printf` an strategischen Punkten einfügen
- Unterstützung durch Tools:
  - Deterministischer, kleinschrittiger Code-Durchlauf
  - Valgrind
  - GDB (GNU Debugger)
    - Open Source
    - Command line Debugger
    - Backend für andere Debugger

Erste Maßnahmen - Beobachtungen

- Programm stürzt nicht ab

```
$ ./recursion
[...]
$ echo $?
0
```

- Ausgabe verändert sich mit unterschiedlichen Optimierungsleveln
- `printf` beeinträchtigt Übersicht
- `-Wall` gibt wichtigen Hinweis

Erste Schritte mit GDB

- Kompilieren mit `-Og -g` oder `-Og -ggdb`[1]
- Interessante GDB-Flags:
  - Übergabe von Parametern: `--args ./app arg1 ... argN`
  - Ordner mit Sourcen: `--directory=DIR`
  - Weitere Literatur: `gdb --help` und `man gdb`
- Neukompilierung während Debugging möglich

[1] Alternativ `-O0 -g` verwenden

How to GDB (I)

Ausführung

- **r**un
- **c**ontinue
- **n**ext
- **s**tep
- finish
- **q**uit/**k**ill

Ausgabe

- **p**rint [/f] [var]
- display [/f] var
- undisplay n
- enable/disable/info display

How to GDB (II)

Break-/Watch-Points

- **b**reak [file:]line | [file:]func [if condition]
- **tb**reak ...
- watch var
- info b
- disable/enable/delete n
- clear [[file:]line|[file:]func]
- delete [n]

Programm-Stack

- **back**t**race [n]
- frame [n]
- up/down [n]

Weiteres[2]

- An laufendes Programm anhängen:
  - `gdb ./runningProcess PID`
  - `attach PID` von gdb Shell

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
  int *dynArray = (int *)malloc(sizeof(int)*10);
  int staticArray[10];
  for(int i=0; i < 10; i++)
  {
    dynArray[i] = i;
    staticArray[i] = i;
  }
  printf("Fertig!\n");
  return 0;
}
```

- set var=value
- print *pointer
- print *dynArray@length
- info threads
- thread n
- GDB-Frontends [Kal18]
- Online-Variante [ss18]
- GDB Quick Reference [Pes00]
- Blog: How Does a C Debugger Work? [Pou14]

[2] memory.c

Motivation

Debugging

Beispiel

Verwendung von GDB

Valgrind

Memcheck

Callgrind

Massif

Quellen

Was ist ... ? [NS07b]

- *"Valgrind is a dynamic binary instrumentation (DBI) framework"*
- Valgrind core + tool plug-in = Valgrind tool

  Core: Instrumentierung der Client-Applikation

  Plug-In: Durchführung der Analyse
- Einfache Bedienung
  - Sourcecode unverändert
  - Robust und weit verbreitet
  - Vielseitige Kontrollmöglichkeiten
- Kombinierte Ausführung mit gdb möglich [Hee16]
- Open Source (GNU General Public License, version 2)

Funktionsweise [NS07b]

- Dynamic Binary Instrumentation $\rightarrow$ Arbeit auf Maschinencode
    - Kein Neukompilieren notwendig
    - Kein Zugriff auf Source-Code notwendig
- Längere Laufzeit durch Emulation
    - Simuliert *Gast*-CPU
    - Shadow Values ("Software-Wrapper") [NS07a]
    - Kernel bleibt Black-Box $\rightarrow$ Systemcalls teuer
    - Serialisiert Threads

  $\Rightarrow$ meist schlechtere Laufzeit um mindestens eine Größenordnung

Valgrind-Tools

- **Memcheck** (Test auf Speicherfehler)
- Cachegrind (Statistik über Cache-Nutzung)
- **Callgrind** (Erstellt Call-Graph)
- **Massif** (Statistik über Heap-Nutzung)
- Helgrind (Test auf Race Conditions)
- DRD (Test auf Fehler beim Multithreading)

Verwendung

Kompilieren:

```
1 $ gcc -g -o app app.c
```

Ausführen:

```
1 $ valgrind --tool=<tool> [valgrind-options] ./app [app-options]
```

Hinweis

- Kompilieren mit `-g` Flag nicht vergessen!
- Hilfe: `man valgrind` oder `valgrind --help`

Motivation aus *Memory (Stack and Heap)*

Fehlerhafte Speicherverwendung:

- Memory leak
- Verwendung nach `free()`
- `free()` auf nicht-dynamischen Speicher
- Zugriff auf nicht-allokierten Speicher
- Mehrfache Speicherfreigabe mit `free()`

Außerdem:

- Verwendung nicht-initialisierter Werte
- ...

Motivation: Memcheck [Manc][3]

- Default-Tool von Valgrind
- Manuelle Verwaltung von dynamischen Speicher in C
  - Vorteil: User kann Speicher-Zugriffe optimieren
  - Nachteil: User kann sich das Leben beliebig schwer machen

$\Rightarrow$ Memcheck prüft Programm auf fehlerhafte Speicher-Nutzung

```
1 $ valgrind ./app [app-options]
```

[3] 02_heap

Motivation: Callgrind [Mana][4]

- Visualisierung des Programmablaufs
- Erkennung potentieller Bottlenecks
- Genauer als einfaches Sampling
- Relativ großer Overhead zur Laufzeit kurzer Methoden
- Ergebnisse in `callgrind.out.<pid>`
- Programme zur Auswertung
  - Texteditor (nicht!)
  - `callgrind_annotate`
  - `kcachegrind`

```
1 $ valgrind --tool=callgrind ./app [app-options]
```

[4] 03_callgraph

Motivation: Callgrind

File View Go Settings Help

Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch

Flat Profile

Search: Search Query (No Grouping)

Incl.	Self	Called	Function	Location
100.00	0.01	(0)	0x0000000000002110	ld-2.30.so
62.25	0.51	1	_dl_start	ld-2.30.so
61.74	0.31	1	_dl_sysdep_start	ld-2.30.so
46.73	0.56	1	dl_main	ld-2.30.so
37.60	13.33	4	_dl_relocate_object	ld-2.30.so
36.72	0.01	1	_start	03_callgraph-example.x
36.72	0.03	1	(below main)	libc-2.30.so
35.79	0.06	1	main	03_callgraph-example.x: 03_callgraph-example.c
32.59	0.06	3	_dl_runtime_resolve_xsave	ld-2.30.so
31.48	0.06	1	myAllocation	03_callgraph-example.x: 03_callgraph-example.c
31.13	0.01	1	malloc_hook_ini	libc-2.30.so
30.80	0.04	1	ptmalloc_init.part.0	libc-2.30.so
30.35	30.33	1	_dl_addr	libc-2.30.so
24.66	9.22	101	_dl_lookup_symbol_x	ld-2.30.so
15.44	11.23	101	do_lookup_x	ld-2.30.so
14.58	14.58	1	__GI___tunables_init	ld-2.30.so
4.71	0.00	3	0x0000000004002090	(unknown)
4.71	0.04	3	_dl_catch_exception	ld-2.30.so
4.60	0.21	3	_dl_map_object	ld-2.30.so
4.26	0.06	10	myPrint	03_callgraph-example.x: 03_callgraph-example.c
4.22	2.32	92	check_match	ld-2.30.so
4.13	0.43	1	_dl_map_object_deps	ld-2.30.so
3.78	1.46	10	__vfprintf_internal	libc-2.30.so
3.41	0.02	2	openaux	ld-2.30.so
3.37	0.14	9	printf	libc-2.30.so
2.43	2.43	153	strcmp	ld-2.30.so
1.90	1.05	2	_dl_map_object_from_fd	ld-2.30.so
1.77	0.77	30	_IO_file_xsputn@@GLIBC_2...	libc-2.30.so
1.41	0.03	1	handle_preload_list	ld-2.30.so
1.27	0.01	1	do_preload	ld-2.30.so
1.25	0.00	1	0x00000000040020f0	(unknown)
1.25	0.01	1	_dl_catch_error	ld-2.30.so
1.22	0.01	1	map_doit	ld-2.30.so
1.15	0.01	1	_dl_receive_error	ld-2.30.so
1.14	0.00	1	version_check_doit	ld-2.30.so
1.14	0.03	1	_dl_check_all_versions	ld-2.30.so
1.11	0.45	1	open_path	ld-2.30.so
1.10	0.70	4	_dl_check_map_versions	ld-2.30.so
1.09	0.19	1	_dl_init_paths	ld-2.30.so
1.01	0.04	1	_dl_init	ld-2.30.so
0.98	0.08	4	call_init.part.0	ld-2.30.so
0.86	0.19	15	_IO_file_overflow@@GLIBC...	libc-2.30.so
0.83	0.00	1	exit	libc-2.30.so
0.82	0.05	1	__run_exit_handlers	libc-2.30.so
0.82	0.10	3	_dl_fixup	ld-2.30.so
0.78	0.11	1	init_cacheinfo	libc-2.30.so
0.74	0.54	18	open_verify.constprop.0	ld-2.30.so
0.67	0.08	3	handle_intel.constprop.0	libc-2.30.so
0.61	0.13	1	_dl_fini	ld-2.30.so
0.61	0.15	1	_dl_load_cache_lookup	ld-2.30.so
0.59	0.59	6	intel_check_word.isra.0	libc-2.30.so

main

Types Callers All Callers Callee Map Source Code

#	Ir	Source
0		--- From '/home/dreistein/Git/02_WR/c-praktikum/2019-2020/slides/06_debugging_and_valgrind/code/03_callgraph-example.c' ---
25		}
26		
27		int main(int argc, char const *argv[])
28	0.00	{
29		int *myOutput;
30	0.00	size_t const mySize = myAllocation(myOutput);
	31.48	1 call to 'myAllocation' (03_callgraph-example.x: 03_callgraph-example.c)
31		
32	0.02	for(size_t i = 0; i < mySize; i++)
33		{
34	0.03	myPrint(myOutput[i]);
	4.26	10 calls to 'myPrint' (03_callgraph-example.x: 03_callgraph-example.c)
35		}
36		
37	0.00	return 0;
38	0.00	}

_start 35.79 %
1 x
(below main) 35.79 %
1 x
main 35.79 %
1 x
10 x
myAllocation 31.48 %
myPrint 4.26 %
1 x
9 x
_dl_runtime_resolve_xsave 31.42 %
printf 3.37 %
10 x
__vfprintf_internal

Parts Callees Call Graph All Callees Caller Map Machine Code

callgrind.out.22380 [1] - Total Instruction Fetch Cost: 220 114

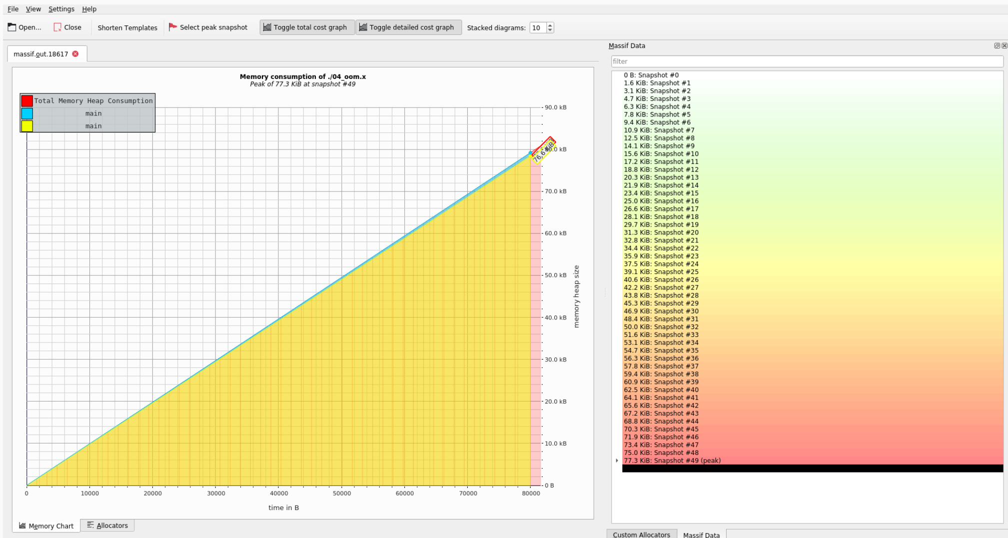
Motivation: Massif [Manb][5]

- Misst Speicherverbrauch
  - "Wirklich verwendeter"Heap-Speicher
  - Indirekt verwendeter Heap-Speicher (Metadaten, alignment, usw.)
  - Stack (nicht per default)
- Ergebnisse in `massif.out.<pid>`
- Rückgriff auf gdb notwendig bei oom
- Programme zur Auswertung:
  - `ms_print`
  - `massif-visualizer`

```
1 $ valgrind --tool=massif [--time-unit=B] ./app [app-options]
```

[5] 04_oom

Motivation: Massif



Motivation

Debugging

Beispiel

Verwendung von GDB

Valgrind

Memcheck

Callgrind

Massif

Quellen

Quellen

[Com47] Wikimedia Commons. **The first "computer bug", 1947.** https://commons.wikimedia.org/wiki/File:H96566k.jpg.

[Hee16] Jason Heeris. **Valgrind and gdb: Tame the wild c, 2016.** https://heeris.id.au/2016/valgrind-gdb/.

[Kal18] Marina Kalashina. **Gdb front ends and other tools, 2018.** https://sourceware.org/gdb/wiki/GDB%20Front%20Ends.

[Mana] Valgrind User Manual. **Callgrind: a call-graph generating cache and branch prediction profiler.**

[Manb] Valgrind User Manual. **Massif: a heap profiler.**

[Manc] Valgrind User Manual. **Memcheck: a memory error detector.**

Quellen ...

[NS07a] Nicholas Nethercote and Julian Seward. **How to shadow every byte of memory used by a program.** In *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*. ACM Press, 2007.

[NS07b] Nicholas Nethercote and Julian Seward. **Valgrind.** In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*. ACM Press, 2007.

[Pes00] Roland H. Pesch. **GDB QUICK REFERENCE, 2000.** http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf.

[Pou14] Kevin Pouget. **How does a c debugger work? (gdb ptrace/x86 example), 2014.** https://blog.0x972.info/?d=2014/11/13/10/40/50-how-does-a-debugger-work.

Quellen ...

[ss18] Mritunjay singh sengar. **Onlinegdb - online compiler and debugger for c/c++, 2018.** https://www.onlinegdb.com.