

Inhaltsverzeichnis

- 1 Einleitung
 - ◆ 1.1 Warum wir das debuggen nicht gelernt haben - Ist die Vorlesung unvollständig?
 - ◆ 1.2 Generelles zur Sorgfältigkeit
 - ◆ 1.3 Debuggen und Fehlersuche
- 2 Einschub: IDE (z.B. Anjuta)
 - ◆ 2.1 Nützliche Features
- 3 gdb mit IDE
- 4 gdb
- 5 Tools
 - ◆ 5.1 valgrind
 - ◇ 5.1.1 memcheck
 - ◇ 5.1.2 massif
 - ◇ 5.1.3 cachegrind
 - ◇ 5.1.4 helgrind
 - ◇ 5.1.5 Links
 - ◆ 5.2 strace
 - ◆ 5.3 mtrace
 - ◇ 5.3.1 Links
 - ◆ 5.4 gcov/gcc
 - ◇ 5.4.1 Links

Einleitung

Warum wir das debuggen nicht gelernt haben - Ist die Vorlesung unvollständig?

Didaktisch ist es wenig sinnvoll sich mit seinen Debugging-Tools im Gepäck auf ein Problem zu stürzen. Verlässt man sich blind darauf, dass ein Fehler durch Debugging-Techniken gefunden wird, neigt man schnell dazu unachtsam zu werden. In der Praxis gibt es zahlreiche Fälle wo Fehler durch Debugging nur schwer oder gar nicht gefunden werden können bzw. ein Debugging von vornherein nicht möglich ist (z.B. unregelmässig auftretende Fehler durch Race-Conditions, Unterschiede im Debug-Modus <-> bei normale Ausführung oder bei exotische Plattformen). Zudem entwickelt sich beim Finden von Fehlern ohne Debugger, durch reines Lesen und Verstehen des Quelltextes, ein ausgeprägteres Verständnis. Trotz alledem sollte man natürlich einmal "zugesehen" haben, was genau passiert wenn ein Bug auftritt und wodurch er verursacht wird. Dafür sind Debugging-Tools unerlässlich.

Generelles zur Sorgfältigkeit

Anders als bei den meisten Hochsprachen prüft ein C-Compiler i.A. nur die wichtigsten Kriterien des Quellcodes. Der Programmierer trägt die volle Verantwortung für seinen Code und fehlende

Compilerwarnung sind kein Garant für fehlerfreien Code. Je robuster bzw. fehlertoleranter der Code ist, desto zuverlässiger wird er funktionieren. Peinlich konservative Fehlerbehandlung erleichtert die Fehlersuche ungemein (bzw. ermöglicht sie erst). Eine redseelige Funktion, die ausführlich über die ausgeführten Aktionen berichtet, erleichtert nicht nur einem selbst das debuggen, sondern ermöglicht auch die Kommunikation zwischen User und Entwickler bei Fehlern, die nur auf dem fremden System auftreten.

Das alles setzt natürlich immer noch eine sorgfältige Planung voraus. Code kann natürlich durch sorgfältige Programmierweise nicht verbessert werden, wenn die Idee oder die Planung schon im Ansatz Fehler aufweisen.

- Links:
 - ◆ [Making wrong code look wrong](#) (englisch)
 - ◆ [Catching integer overflows in C](#) (englisch)

Debuggen und Fehlersuche

Es gibt zahllose unterschiedliche Arten von Bugs. Selten findet ein schlimmer Bug seine Ursache in einem einzigen Programmierfehler. Meist tritt er nur unter bestimmten, selten eintretenden Voraussetzungen auf. Die Voraussetzungen können von so komplexer Natur sein, dass das Auftreten zufällig scheint. Man muss in jedem Fall aufs Neue entscheiden, mit welchen Methoden und Werkzeugen man sich auf die Fehlersuche begibt. Z.B. ist es nicht in jedem Fall sinnvoll tonnenweise Code zu wälzen. Verlässt man sich andererseits ausschließlich auf sein Toolset, kann die wahre Ursache eines Bugs verborgen bleiben, so dass der Fehler zwar beseitigt scheint, in Wahrheit aber nur ein einziges Auftreten eliminiert wurde und noch zahlreiche Situationen existieren, in denen sich der Bug noch meldet. Obwohl es unzählige Arten von Bugs gibt, so gibt es doch, vor allem in C eine gewisse Zahl an "Kinderkrankheiten" die immer wieder auftreten. Für das finden dieser Art von Fehlern sind die folgenden Tools gut geeignet.

Einschub: IDE (z.B. Anjuta)

Das Programmieren ohne Integrierte Entwicklungsumgebung ist zwar möglich aber unnötig schwer. Moderne IDEs bieten zahlreiche nützliche Features die einem ermöglichen auch mit großen Projekten ohne viel Aufwand umzugehen.

Nützliche Features

- *Syntax Highlighting* - Quelltext wird farbig dargestellt. - vergessene Anführungsstriche gehören der Vergangenheit an.
- *Klammernprüfung* - Klammerpaare werden hervorgehoben - befindet sich der Cursor bei einer Klammer, wird diese und ihr Pendant farbig dargestellt. Fehlt ein Pendant wird dies deutlich.
- *Code-Collapse* - Gruppierungen (Klammern, Kommentare) können "zusammengefaltet" werden. Nicht benötigte Code-Stücke lassen sich ausblenden. Klammerfehler über mehrere Seiten hinweg werden deutlich.

gdb mit IDE

- Einzelschritt
- Variablen beobachten
- Speichermonitor

gdb

- Problem: Das Programm verhält sich anders wenn es von der shell gestartet wird und wenn es von einem anderen Prozess aus gestartet wird.
 - ◆ Lösung: Baue in deinen Code eine endlosschleife ein:

- 1.
2. `int delay = 0;`
3. `while(delay == 0);`
- 4.

- - ◆ lass das Programm ausführen und "attache" gdb an den Task
 - ◆ setze "delay = 1" (mit "p flp=1") und beobachte das Programm im Einzelschritt-Modus

Tools

valgrind

Verschiedene Tools (`--tool=[memcheck,cachegrind,...]`).

memcheck

Memory-Leak debugging, Speicherschutzverletzungen verfolgen. memcheck **findet nur tatsächlich auftretende Fehler**

- **Optionen:**
 - ◆ `--leak-check=full` schaltet alle Prüfungen ein
- **Beispiele:**
 - ◆ `malloc()` ohne `free()`
 - ◆ off-by-one
 - ◆ unhandled NULL-Pointer

massif

memory/heap profiler.

- **Optionen:**
 - ◆ `--format=<text|html>` [default: text] - Ausgabe als text oder html

- **Beispiele:**

- ◆ valgrind --tool=massif --depth=16 ./sysprakserver
- ◆ gv massif.<pid>.ps

cachegrind

Performance debugging. (fortgeschritten) Analysiert das Verhalten des L1 und L2 Prozessorcaches.

- ◆ L1 Cache misses kosten i.A. ~10 Cycles
- ◆ L2 misses kosten i.A. ~200 Cycles

cachegrind speichert die Analyse in **cachegrind.out.<pid>** und gibt eine Zusammenfassung aus. Mit **cg_annotate** wird der Quellcode in die Analyse-Datei eingefügt und auf stdout ausgegeben. Mit **kcachegrind** kann man die Analyse mit einer grafischen Oberfläche durchführen und visualisieren.

helgrind

Race Condition debugging. **Funktioniert nicht in der aktuellen Version :)**

- **Optionen:**
- **Beispiele:**

Links

- Homepage: <http://valgrind.org>
- Einführung: [Linux-Magazine Article](#) (englisch)
- User-Manual: [manual](#) (englisch)
- [Daikon](#) - MIT's invariant detection system
- [Taintcheck](#) - CMU's exploit detector and analyser
- [Using valgrind](#) - Tutorial mit Beispielen (englisch)

strace

- **System Calls + Aufrufparameter** verfolgen: open(), read(), write(), ...
-> Interaktion des Programms mit dem Betriebssystem
- FAQ...
 - ◆ "Wo blockiert mein Programm?"
 - ◆ "Welche Daten wurden über einen File deskriptor gelesen?"
 - ◆ ...
- Interessante Aufrufparameter
 - ◆ Programm direkt starten: strace <Programm>
 - ◆ An laufendes Programm "anhängen": strace -p <PID>
 - ◆ Nur bestimmte SysCalls anzeigen, oder festlegen: Parameter -e <expr>. Beispiel:
-e trace={file, process, network, signal, ...}
- Anwendungsbeispiel

mtrace

- findet memory-leaks ähnlich wie valgrind
- Muss in den Quellcode eincompiliert werden

Links

- [Wikipedia](#)

gcov/gcc

"Sourcecode-profiling" zur Laufzeitermittlung ermöglicht es nachzuvollziehen welche Funktion wie oft aufgerufen wurde. So ist für jede Zeile des Quelltextes ersichtlich, wie oft diese ausgeführt wurde.

Links

- [profiling mini-HOWTO](#) (deutsch)
- [Guide to Faster, Less Frustrating Debugging](#) (englisch)