Konzepte von Betriebssystemkomponenten

WS 2003/2004

Kernel-Debugging

Christian Plöger,

09.02.2004

Überblick

- Einführung „Debugging“
- Debugging Konzepte
- Dynamische Debugger
- Wie Debugger arbeiten

Überblick

- Kernel-Debugging
  - Problematik
  - Verschiedene Kernel-Debugger kurz vorgestellt
  - KGDB

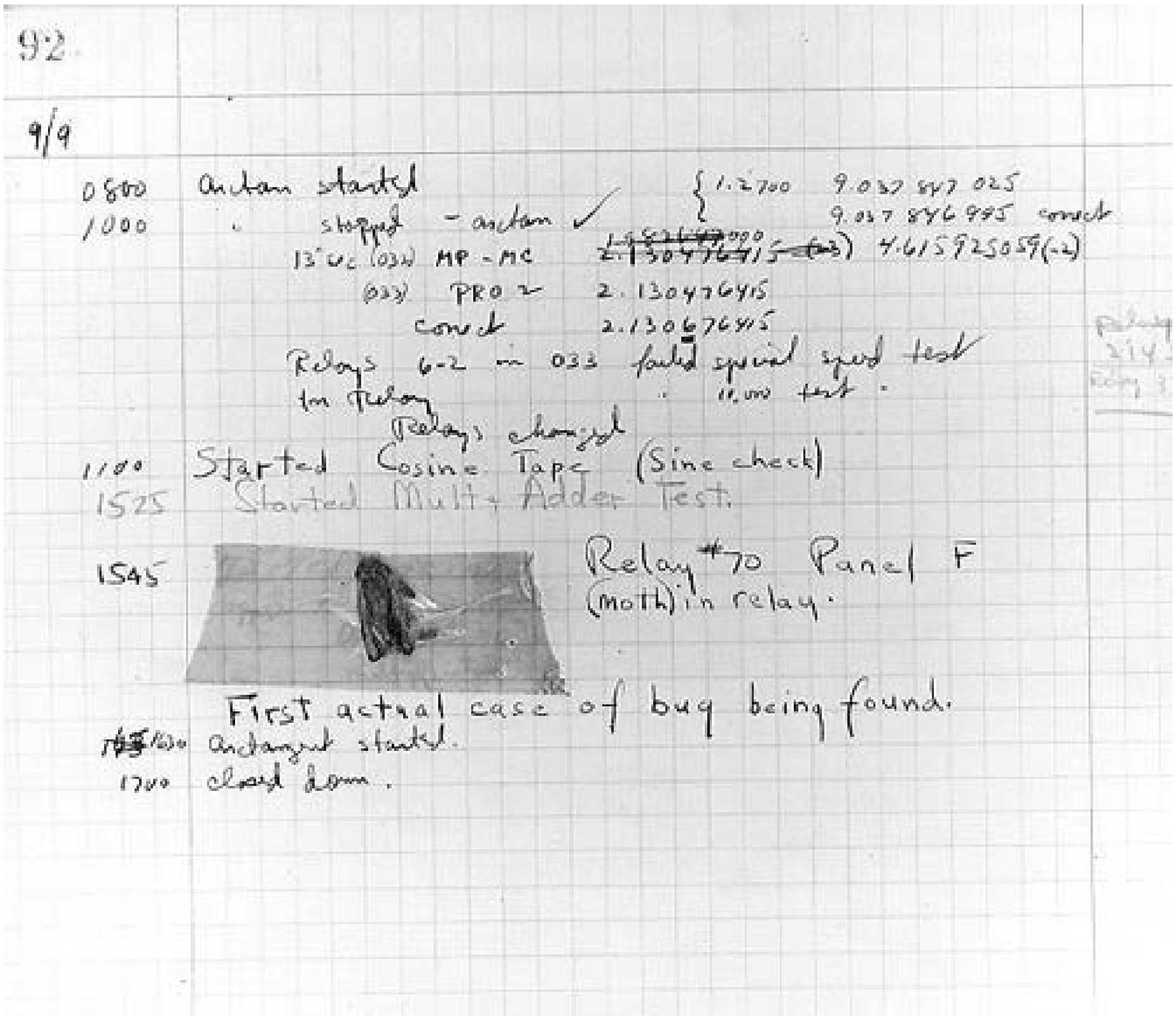
Einführung - Was ist ein Bug?

- Fehler in Hard- oder Software
  - Entwurfsfehler
  - Implementierungsfehler
  - Produktionsfehler (Hardware)
- Verwendung von „Bug“ im Vortrag:
  - Programmierfehler, welcher sich erst zur Laufzeit des Programms offenbart

Einführung - Was ist ein Bug?

- Warum heißen Bugs, Bugs?
  - Engl. „bug“: Käfer
  - Seit Ende 19. Jh. Beschreibung für Fehler in Automaten
  - Herkunft unklar
- 1944 prägte Grace Hopper Begriff „Debugging“
  - Fand bei Fehlersuche im Mark I eine Motte deren Flügel das Einlesen der Löcher der Papierrolle blockierten

Corpus Delicti



Einführung - Was ist Debugging?

- Lokalisieren und Beseitigen von Fehlern
- Software welche Debugging anderer Software ermöglicht bezeichnet man als Debugger

Einführung – Debugging Konzepte

Zwei Debugging-Ansätze sind zu unterscheiden:

- Statisches Debugging
  - Fehlersuche/-beseitigung „vor Laufzeit“
- Dynamisches Debugging
  - Fehlersuche/-beseitigung während oder nach der Ausführung

Einführung – Dynamisches Debugging

Realisierung des „Debugging“ durch:

- Kontrollierte Ausführung
- ... und Überwachung eines Programms

Zwei Gruppen dynamischer Debugger:

- Debugger auf Maschinenebene
- Debugger auf Quellsprachenebene

Einführung – Dynamisches Debugging

Debugger auf Maschinenebene

Reinform wird nur noch selten verwendet

- Erlauben Verfolgung des Programmgeschehens nur in Assemblercode
- Vorteil
  - Untersuchung von „Closed Source“ Software möglich
- Nachteile
  - Unübersichtlicher Assemblercode
  - Genaue Kenntnisse der Rechnerarchitektur nötig

Einführung – Dynamisches Debugging

Debugger auf Quellsprachenebene

Auch symbolische Debugger genannt

- Verfolgung des Programmgeschehens in Quellsprache oder Maschinencode
- Verwendung der Symbole des Quelltextes (Variablennamen, Funktionsnamen,...)
- Auswertung von Ausdrücken in Quellsprache
- Ausführung bzw. Überspringen kompletter Funktionen/Prozeduren

Einführung – Dynamisches Debugging

Debugger auf Quellsprachenebene

Voraussetzung:

- Zu untersuchendes Programm muss mit Debugging-Symbolen/-Information angereichert werden

Nachteile:

- Neuübersetzung des Programms nötig
- Quellcode muss vorliegen
- Resultierendes Binary kann erheblich größer sein als „Original“

Einführung – Kernfunktionalitäten

Um gezielte Fehlersuche zu ermöglichen stellen Debugger folgende Möglichkeiten zur Verfügung:

- Haltepunkte
- Einzelschrittausführung
- Umgebungsinformationen auslesen
- Post-Mortem-Debugging
- (Veränderung des Programmverlaufs)

Einführung – Kernfunktionalitäten

Haltepunkte – engl. „breakpoints“

- Quelltexthaltepunkte
  - Stoppen Programmausführung bei Erreichen einer festgelegten Quelltextzeile
- Datenhaltepunkte (meist „watchpoints“)
  - Stoppen Programmausführung bei Referenzierung einer bestimmten Variable
- Bedingte Haltepunkte
  - Stoppen Programmausführung „nur wenn eine bestimmte Bedingung erfüllt ist“

Einführung – Kernfunktionalitäten

Einzelschrittausführung – engl. „single stepping“

- Debugger führt genau eine Anweisung, oder einen bestimmten Block von Anweisungen aus (einen Schritt) und stoppt dann

Möglichkeiten auf Quellsprachenebene:

- Funktionen/Prozeduren können als ein Schritt behandelt werden (step over)
- Funktionen/Prozeduren können betreten werden, danach erfolgt Halt (step into)

Einführung – Kernfunktionalitäten

Umgebungsinformationen auslesen

Nach Anhalten/Absturz des Programms

- Wo sind wir?
  - Anzeige des Programmzählers, passende Quelltextstelle in der Quelltextanzeige
- Wie kamen wir hierher?
  - Auswertung des Programmstapels, meist durchlaufbare Stackanzeige
- Programmzustand?
  - Untersuchen der Variablenbelegung

Einführung – Kernfunktionalitäten

Post-Mortem-Debugging

- Untersuchung des letzten Zustand des Programms vor dem Absturz
- Debugger liest ein, vom OS zum Zeitpunkt des Absturzes erzeugtes, Speicherabbild ein
  - Variablen
  - Aufrufkeller (Stack)

Einführung – Kernfunktionalitäten

Veränderung des Programmverlaufs

Einige Debugger erlauben es direkte Veränderungen vorzunehmen:

- Variablenwerte
- Register
- Programmzähler (Instruction Pointer)
  - Überspringen von einzelnen Instruktionen oder ganzen Programmblöcken

Wie funktionieren Debugger

Grundlagen (Linux/UNIX)

- Normalerweise führt Betriebssystem Programme als Prozesse aus
- Betriebssystem stellt Schnittstelle zur Kontrolle der Ausführung von Prozessen durch andere Prozesse:
- *ptrace*()
  - Verschiedene Parameter ermöglichen Untersuchung und Kontrolle eines Prozesses

Wie funktionieren Debugger

Grundlagen (Linux/UNIX) – *ptrace*()

- PTRACE_TRACEME:
  - Der aufrufende Prozess wird zur kontrollierten Ausführung vorbereitet
- PTRACE_PEEKTEXT,..._PEEKDATA:
  - Daten aus dem Speicher des Prozesses lesen
  - -TEXT greift auf das Codesegment,
  - -DATA auf das Datensegment zu
- PTRACE_POKETEXT,..._POKEDATA:
  - -TEXT: in das Codesegment schreiben
  - -DATA: in das Datensegment schreiben

Wie funktionieren Debugger

Grundlagen (Linux/UNIX)

Was passiert?

- Debugger ruft *fork*() auf
- Kind-Prozess
  - *ptrace*(PTRACE_TRACEME)
  - Ersetzt sich selbst mit *exec* durch das zu untersuchende Programm
- Debugger (Eltern-Prozess)
  - Kann nun mit *ptrace*() das Kind steuern

Wie funktionieren Debugger

Haltepunkte

- Debugger fügt Haltepunkt-Anweisungen in den Maschinencode des Programms
  - Prozessor löst bei deren Auftreten eine Unterbrechung aus => Kernel unterbricht Prozess mit Signal
  - Signal wir vom Debugger aufgefangen
- Prozess kann nun untersucht werden
- Manche Prozessoren (Pentium Familie) bieten Debug-Register zur Speicherung von Haltepunkten

Wie funktionieren Debugger

Einzelschrittausführung

Debugger schickt PTRACE_SINGLESTEP

- Nach jeder Ausführung eines Maschinenbefehls wird Prozess angehalten
  - Implementierung auf x86 durch setzen des TF (*trap flags*) im *eflags* Register
  - Prozessor wirft eine „Debug Exception“, Handler setzt TF zurück, erzwingt Halten des Prozesses
  - SIGCHLD an Eltern-Prozess(Debugger)

Wie funktionieren Debugger

Arbeiten auf Quellsprachenebene

- Beim Compilieren des Programms müssen Debug-Information/-Symbole eingeflochten werden
- Dazu muss der Compiler mit entsprechenden Parametern gestartet werden
  - *gcc –g* beim GNU C Compiler
- Die Informationen werden im sog. STABS-Format (engl. „symbol table“) gespeichert

Kernel-Debugging

Problematik

- Ein Betriebssystemkern (Kernel) ist normalerweise sehr groß und ...
- durch seine Komplexität sehr unübersichtlich

- **Es gibt keine übergeordnete Schicht welche die Ausführung des Kernels kontrolliert**

Kernel-Debugging

Vorgehensweisen – Debug-Ausgaben

- Einfügen von Ausgabeanweisungen an interessanten Stellen des Quelltextes
- Im Kernel wird dafür *printk*() zur Verfügung gestellt
- Syntax ähnlich wie *printf*()
  - *printk*(KERN_DEBUG,“Aktuelle Codestelle: %s: %i\n“,__FILE__,__LINE_&_)
- Performance & Informationsverlust bei übermäßigem Gebrauch von *printk*
  - *klogd* (falls aktiv) schreibt die Daten über *syslogd* sofort auf die Festplatte

Kernel-Debugging

Vorgehensweisen – Beobachten des Systemverhaltens

Indirekte Methode

- *strace* erlaubt Verfolgung aller Systemaufrufe mit deren Eingabe- und Ausgabedaten
  - Im Fehlerfall Anzeige des symbolischen Wertes (z.B.ENOMEM)
  - ... und des passenden Strings(z.B. „Out of Memory“)
- Programm muss nicht mit Debug-Informationen übersetzt worden sein
- Suche begrenzt sich auf Kernel/Treiber

Kernel-Debugging

Vorgehensweisen – Auslesen benötigter Informationen

Nutzung vorhandener Linux/UNIX-Werkzeuge

- *ps*, *netstat*, etc.

Voraussetzung:

- Kein fataler Bug, System sollte noch lauffähig sein

Eingrenzung der Fehlerquelle

(Beispiel *ps* siehe Handout)

Kernel-Debugging

Vorgehensweisen – „Oops“ Auswertung

Kernel-Oops-Meldungen

- Ausgabe durch den Kernel via *printk*()
- Enthalten u.a. Inhalt der CPU-Register, Lage der Seitendeskriptortabellen
- Ausgabe erfolgt in hexadezimaler Schreibweise

klogd und *ksysmoops*

- Decodieren Ooops
- Zeigen letzten Funktionsaufruf, bzw. in welchem Modul bei welchem Offset

Kernel-Debugging

Kernel-Debugger – gdb

Laufender Kernel kann mit GNU Debugger (gdb) untersucht werden:

- Unkomprimierte Version des Kernels (mit Debug-Symbolen) benötigt
- „Nur Lese“-Zugriff
  - => Keine Haltepunkte, Einzelschrittmodus, etc.
- Aufruf:
  - *gdb /usr/src/linux/vmlinux /proc/kcore*
- */proc/kcore* repräsentiert den momentan laufenden Kernel im Speicher

Kernel-Debugging

Kernel-Debugger – gdb – Hinweise

Zu Beachten:

- Der Kernel wird NICHT angehalten
- *gdb* führt internen Zwischenspeicher für gelesene Daten
- Bei wiederholtem Zugriff => Daten aus Zwischenspeicher
- Aktuelle Werte durch Reinitialisierung des Zwischenspeichers
  - *core file /proc/kcore*

Kernel-Debugging

Kernel-Debugger – IKD und kdb

IKD - **I**ntegrated **K**ernel **D**ebugger:

- Umfangreiche Werkzeugsammlung

Beispiele

- Programmzähler Ausgabe
  - Der momentane Programmzähler wird auf einer virtuellen Konsole ausgegeben
- „Soft Lockup“-Detektor
  - Führt einen Zähler beim Aufruf von Kernel-Prozeduren
  - Aufruf des Schedulers setzt den Zähler zurück
  - Läuft die Prozedur zu lange ohne Scheduler Aufruf, wird ein Kernel-Oops erzeugt.

Kernel-Debugging

Kernel-Debugger – IKD und kdb

kdb ist Bestandteil des IKD

- Erlaubt Anhalten des Kernels
- Stellt volle Debuggerfunktionalität zur Verfügung
- gdb Syntax
- Arbeitet auf Maschinencodeebene
- U.U. Verlust von erarbeiteter Information beim totalem Absturz => KDGB

Kernel-Debugging

Kernel-Debugger – Kernel Crash Dump Analyzer

- Kernel-Patch
  - Bei Kernel-Oops wird ein Kopie des Systemzustands sofort auf ein Dump-Gerät geschrieben
- User-Space Werkzeuge
  - Zur Analyse des erzeugten Dumps
  - Normalerweise Debugger ähnliche Möglichkeiten zur „read-only“-Untersuchung

Kernel-Debugging

Kernel-Debugger – User-Mode-Linux

Portierung des Linux Kernels

- Kernel läuft im User-Space
- Hardwareschicht wird durch Systemaufrufe an das „Mutter“-System emuliert
- Kernel lässt sich wie „normales“ Programm untersuchen.

Kernel-Debugging

KGDB

Interaktiver Kernel-Debugger

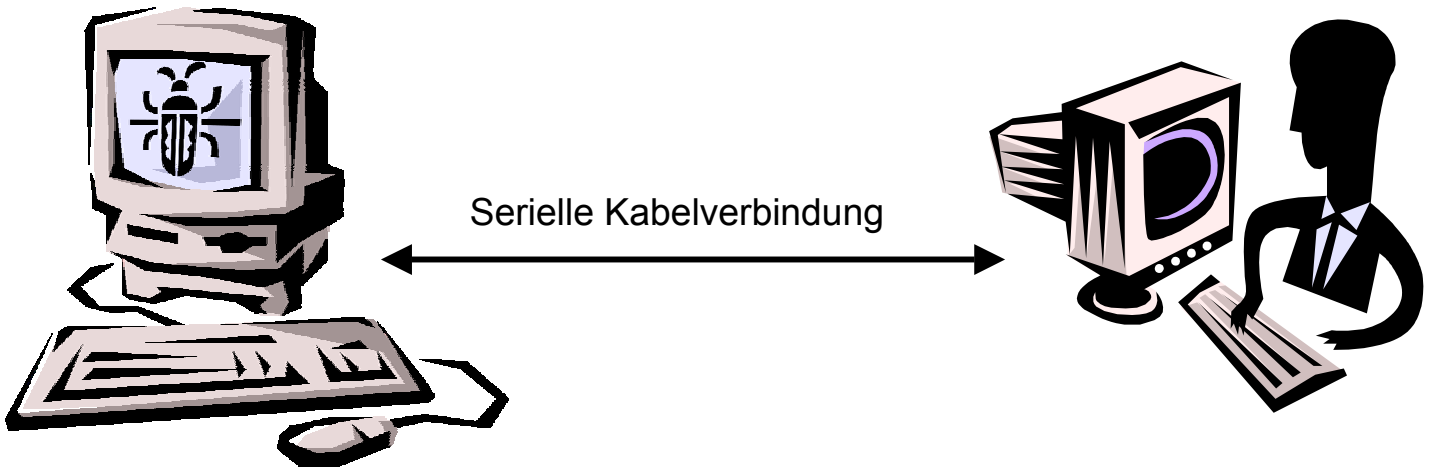
- Volle Debuggerfunktionalität
- Ist eine Variante des *gdb*
- Arbeit auf Quellsprachenebene, Maschinencodeanzeige verfügbar
- Arbeit an Systemen ohne Bildschirm+Tastatur möglich
  - Auch mit *kdb*+“console on serial line“ möglich => Funktionsfähigkeit hängt aber immer noch vom zu untersuchenden System ab

Kernel-Debugging

KGDB – Einrichtung - Hardware

„test“/“target“System

„development“/“monitor“-System



Kernel-Debugging

KGDB – Wie funktioniert kgdb

- Veränderung der IDT
  - debug() Exceptions-Handler (Trap) wird verändert
  - *int3* Exception-Handler wird verändert
- Am monitor-System wird STRG-C gedrückt
- Kernel durch Exception unterbrochen
- *remote-gdb* Situng wird gestartet

Kernel-Debugging

KGDB – Wie funktioniert kgdb

- Angehaltener Kernel kann mit *gdb* untersucht werden
- *continue [c]* am monitor-System, gesicherte Register werden zurückkopiert
- Ausführung wird fortgesetzt, wo sie unterbrochen wurde

Kernel-Debugging

KGDB – Wie funktioniert kgdb - *Einzelschrittausführung*

trap-Anweisung wird nach der nächsten Quelltextzeile eingefügt:

- Registerinhalte auf Stack gesichert
- Weitere *trap*-Anweisung wird sofort wieder an die Stelle nach der nächsten Quelltextzeile eingefügt
- Alte *trap* wird entfernt
- *remote-gdb* wird gestartet
- Nach *c* werden die (veränderten) Register zurückkopiert
- Ausführung des Kernels wird fortgesetzt

Kernel-Debugging

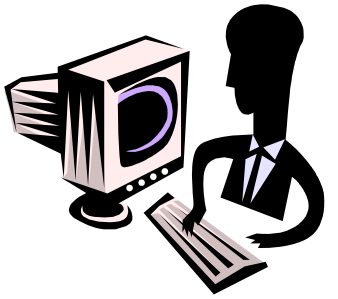
KGDB – Wie funktioniert kgdb - *Haltepunkte*

Analog zur Einzelschrittausführung

Trap-Anweisung wird an frei wählbarer Stelle eingefügt

Kernel-Debugging

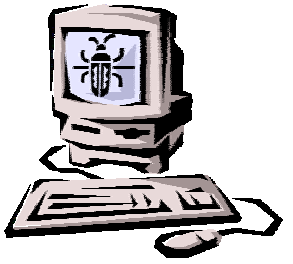
KGDB – Einrichtung – Software – monitor-System



- *gdb* installieren
- Passenden *kgdb* Kernel-Patch einspielen
- Kernel konfigurieren
  - „Kernel Hacking“:
    - "KGDB: Remote (serial) kernel debugging with gdb"
    - "KGDB: Thread analysis„
    - "KGDB: Console messages through gdb„
- Kernel kompilieren

Kernel-Debugging

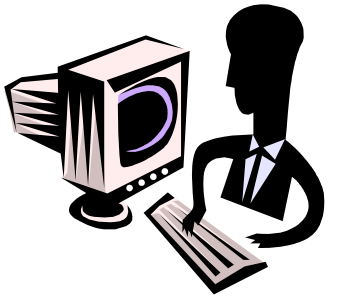
KGDB – Einrichtung – Software – test-System



- Den auf dem monitor-System erstellten Kernel auf test-System installieren
- Kernel mit Argumenten booten:
  - *gdb gdbttyS=ttyS0 gdbbaud=115200*
    - *ttyS0 für die erste serielle Schnittstelle*
- Es erscheint:
  - *+Waiting for connection from remote gdb...*

Kernel-Debugging

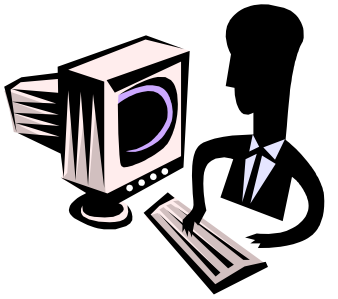
KGDB – Benutzung – monitor-System



- Ins Stammverzeichnis des gepatchten Kernels wechseln (/usr/src/linux)
- *gdb* starten
  - *>gdb vmlinux*
- In der gdb-Kommandozeile Baudrate festlegen:
  - *(gdb) set remotebaud 115200*
- Mit dem test-System verbinden:
  - *(gdb) target remote /dev/ttyS0*
  - Ausgabe: *Remote debugging using /dev/ttyS0*

Kernel-Debugging

KGDB – Benutzung – monitor-System

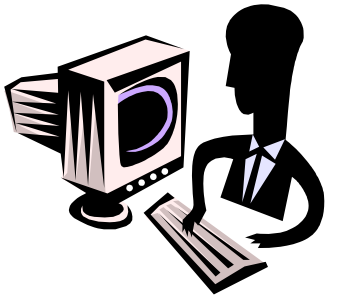


Ab jetzt sind alle gdb Kommandos verwendbar

- *continue* / *c*:
  - Setzt die Ausführung fort
- STRG-C:
  - Kernel auf test-System wird angehalten
- *backtrace* / *bt*:
  - Zeigt den momentanen Aufrufstapel an
- *print* / *p* <Ausdruck>:
  - Wertet den angegeben Ausdruck in der Quellsprache aus

Kernel-Debugging

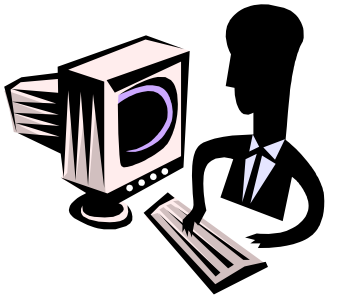
KGDB – Benutzung – monitor-System



- *break* / *bp* (...):
  - Haltepunkte
    - Funktionsname: *bp my_function*
    - Zeilennummer: *bp 42*
    - ...
  - Bedingte Haltepunkte
    - *bp* (...) *if* „Bedingung“
      - Prüft bei erreichen von (...) die „Bedingung“
        - » False: Ausführung wird fortgesetzt bis (...)
        - » True: gestoppt

Kernel-Debugging

KGDB – Benutzung – monitor-System



- *list* / *l* <Zeilennummer/Funktion>:
  - Zeigt 10 Zeilen Quelltext ab Zeilennummer oder ab Beginn der Funktion
- *step* / *s*:
  - Ausführung fortsetzen bis nächste Quelltextzeile erreicht

... *man gdb* !

THE END

Fragen?