Proseminar
Konzepte von Betriebssystemkomponenten (KVBK) WS 2003/2004

Kernel-Debugging

von Christian Plöger
am 09.02.2004

1. Einführung - Debugging bzw. Entfehlern von Programmen

1.1 Begriffsklärung

1.1.1 Was ist ein Bug?

Im engeren Sinne versteht man unter einem Bug einen Programmierfehler, der sich zur Ausführungszeit des Programms bemerkbar macht. Man kann diese Definition jedoch abschwächen und Fehler bei der Analyse der Anforderungen und im Programmentwurf als Bugs bezeichnen.

1.1.2 Was ist Debugging?

Der Begriff „Debugging“ bezeichnet die Tätigkeit der *Lokalisierung* und *Beseitigung* von Fehlern in Softwaresystemen.

1.2 Debugging Konzepte

Generell können zwei Debugging-Methoden unterschieden werden:

Statisches Debugging beinhaltet jede Art von Fehlersuche in Dokumenten, also in der Anforderungsspezifikation, dem Design oder dem Quelltext.

Dynamisches Debugging hingegen bezeichnet die Fehlersuche in Programmen während oder nach ihrer Ausführung. (Im Folgenden nur „Debugging“/debuggen)

1.2.1 Dynamisches Debugging

Dynamisches Debugging setzt zur Ausführungszeit des Programms an und ist die am häufigsten anzutreffende Methode der Fehlersuche. Während beim Testen Fehler durch ihre *Wirkung* auf das beobachtbare Programmverhalten gefunden werden, ist das Ergründen der *Ursache* für dieses (fehlerhafte) Verhalten das wichtigste Ziel beim dynamischen debuggen.

1.2.2 "Manuelles" Debugging mit Debug-Ausgaben

Hierzu werden Anweisungen vom Programmierer selbst in den Quelltext eingefügt, die Ausgaben machen, welche Rückschlüsse auf den Programmverlauf und Zustand zulassen.

1.3 Der (dynamische) Debugger

Wesentlich bequemer ist der Einsatz spezieller Software, der Debugger, welche eine kontrollierte Ausführung und Überwachung eines in der Ausführung befindlichen Programms erlaubt.
Doch auch hier muss differenziert werden. So lassen sich die Debugger anhand der bereitgestellten Abstraktionsebene in zwei Klassen einteilen, die Debugger auf Maschinenebene und auf Quellsprachenebene (engl. „Source Level Debugger“).

1.3.1 Debugger auf Maschinenebene

Die Möglichkeiten von Debuggern auf *Maschinenebene* beschränken sich im wesentlichen auf die Ausgabe der Maschinenregister, des Programmzählers sowie Datenbereichen des Speichers in tabellarischer Form. Sie werden heute nur noch selten verwendet, da die meisten Quellsprachen-Debugger diesen Modus ebenso anbieten.

1.3.2 Debugger auf Quellsprachenebene

... werden, weil sie auf Symbolen wie Variablen- und Funktionsnamen der Quellsprache arbeiten, auch *symbolische Debugger* genannt.
Um dies zu ermöglichen müssen Programme, die mit einem solchen Werkzeug untersucht werden sollen, bereits beim Übersetzen mit zusätzlichen Informationen angereichert werden. (Beispielsweise durch Veerwendung von „-g“ beim *gcc*)

1.4 Kernfunktionalitäten von Debuggern

1.4.1 Haltepunkte

Haltepunkte (engl. „breakpoints“) erlauben dem Benutzer, ein Programm an definierten Programmpunkten anzuhalten, um seinen Zustand zu untersuchen.

Quelltext-Haltepunkte (engl. „code breakpoints“) stoppen die Programmausführung, wenn eine bestimmte Zeile im Quelltext erreicht wird.
Daten-Haltepunkte (engl. „data breakpoints“ oder „watchpoints“) hingegen, wenn eine bestimmte Variable referenziert wird.

1.4.2 Einzelschritt-Ausführung

Um den Programmablauf genau verfolgen zu können, gibt es in vielen Debuggern die Möglichkeit, das Programm in *Einzelschritten* ablaufen zu lassen. Festzulegen ist die Granularität der Ausführungsschritte. Es bieten sich hier verschiedene sinnvolle Möglichkeiten. Der GNU-Debugger „gdb“ beispielsweise fasst im Quellsprachenebenenmodus alle Anweisungen einer Quelltextzeile zu einem Schritt zusammen.

1.4.3 Umgebungsinformation

Sobald ein Programm angehalten wurde – durch Erreichen eines Haltepunktes oder aufgrund einer Ausnahmesituation (Absturz) – müssen wir herausfinden, wo wir uns momentan im Programm befinden, wie wir an diese Stelle kamen, und wie der aktuelle Gesamtzustand des Programms ist.

Wo sind wir?

Normalerweise werden die zum aktuellen Programmzähler passenden Quelltextkoordinaten (Datei, Zeilen- und Spaltennummer) angezeigt. Viele Debugger besitzen daher eine *Quelltextanzeige*, die einen Ausschnitt des momentan relevanten Quelltextstücks zeigt und die aktuelle Position markiert.

Wie kamen wir hierher?

Um an diese Informationen zu kommen, wertet der Debugger den Aufrufkeller (Stack) aus. Die meisten Debugger bieten eine *Stackanzeige* mit der Möglichkeit, im Stack zu „wandern“.

Programmzustand?

Der Programmzustand wird durch die aktuelle Variablenbelegung beschrieben. Die Variablenanzeige ermöglicht es nun diese zu untersuchen.

1.4.4 Post-Mortem-Debugging

Eine der wichtigsten Eigenschaften von Debuggern: Nach einem Programmabsturz kann man den letzten Zustand des Programms untersuchen. Ein *core dump* kann später von einem Debugger eingelesen werden, so dass man sich ein Bild vom fehlerhaften Programmzustand machen kann.

1.4.5 Veränderung des Programmverlaufs

Mit einigen Debuggern kann man den Programmcode während der Ausführung verändern. Damit kann man gefundene Fehlerursachen während eines Programmhalts reparieren und den Erfolg der Reparatur nach dem Fortsetzen der Ausführung überprüfen.

2. Wie funktionieren Debugger

2.1 Grundlage

Es ist Aufgabe des Betriebssystems, Programme als Prozesse auszuführen. Deshalb benötigt auch der Debugger für das *kontrollierte* Ausführen Unterstützung vom Betriebssystem.
Unter Linux/UNIX bietet das Betriebssystem den Systemaufruf *ptrace*, mit dem ein Kind-Prozess kontrolliert ausgeführt werden kann.

Vorgehensweise:

- Der Debugger-Prozess spaltet sich in einen Vater-Prozess und einen Kind-Prozess. (*fork*())
- Der Kind-Prozess ruft *ptrace*(PTRACE_TRACEME) auf.
- Der Kind-Prozess ersetzt sich mit „*exec*“ durch das auszuführende Programm.

2.2 Einzelschritt-Ausführung

***ptrace*(PTRACE_SINGLESTEP):** Den Schalter zur schrittweisen Ausführung setzen oder löschen. Ist dieser Schalter aktiv, wird nach jeder Ausführung eines Maschinenbefehls der Prozess durch ein Signal angehalten.

Auf x86 Architekturen wird diese Funktion durch Setzen des TF (*trap flags*) im *eflags* Register implementiert:

Sobald dieses Flag gesetzt ist, wird nach jeder Maschinencodezeile eine „Debug“-Exception geworfen. Der entsprechende Exception Handler setzt das Flag wieder zurück, erzwingt ein Halten des aktuellen Prozesses und schickt eine SIGCHLD an seinen Vater-Prozess (Debugger).

2.3 Arbeiten auf Quellsprachenebene

Wie stellt der Debugger die Verbindung zwischen dem Programmspeicher und den im Original-Quelltext benutzten Variablen her?

Dies ist die Aufgabe der beim Übersetzen angelegten Debugging-Informationen, welche in die ausführbare Datei automatisch eingeflochten werden. Dem Compiler muss dies durch entsprechende Parameter mitgeteilt werden (Bsp.: *gcc –g mein_proggy.c*). Diese Informationen liegen im STABS-Format vor (engl. „symbol table“: Symboltabelle).

2.4 Haltepunkte

Vom Debugger werden in den Maschinencode des Programms besondere *Haltepunkte-Anweisungen* eingefügt, bei deren Auftreten der Prozessor eine Unterbrechung auslöst und das Betriebssystem den Prozess mit einem speziellen Signal unterbricht.

Setzt der Debugger die Ausführung fort, muss er die Haltepunkte-Anweisungen zunächst durch die ursprüngliche Anweisung ersetzen. Diese Anweisung wird dann in einem Schritt ausgeführt. Anschließend wird der Haltepunkte wieder eingefügt und die Ausführung bis zum nächsten Signal fortgesetzt. Manche Prozessoren (Pentium Familie von Intel) bieten spezielle Haltepunkte-Register (debug registers), welche Haltepunkte speichern können und so das Verändern des Maschinencodes überflüssig machen.

3. Kernel-Debugging

3.1 Die Problematik

Möchte man Fehler im Kern untersuchen, trifft man grundsätzlich auf einige Probleme:

- Ein Kernel ist normalerweise sehr groß (Im Falle von Linux und Windows Millionen von Quelltextzeilen)
- Ein Kernel ist sehr komplex (Mehrfädigkeit, hardwarespezifische Bereiche, ...)
- Es gibt kein übergeordnetes Programm, welches die Ausführung des Kernels überwacht.

Die Tatsache, dass es keine Ausführungsüberwachungsschicht für den Kernel gibt, ist der zentrale Unterschied zum Debugging von Benutzerprogrammen.

3.2 Anwendbare Methoden

3.2.1 Debug-Ausgaben im Quelltext - *printk*

Die gängigste Debugging-Technik ist die Überwachung des Programmablaufs. In normalen Anwendungsprogrammen kann man dies durch *printf* an passender Stelle erreichen. Ähnliches ist im Kernel-Code möglich, dazu kann man die Funktion *printk* verwenden. Allerdings gibt es entscheidende Unterschiede.
printk bietet die Möglichkeit, Meldungen nach ihrer Wichtigkeit oder Bedeutung in verschiedene *Loglevel* einzuteilen, was normalerweise durch ein **MAKRO** geschieht.

3.2.2 Beobachten des Systemverhaltens - *strace*

Besonders für Gerätetreiber geeignet ist die Methode, Anwenderprogramme (also im User-Space) zu beobachten, welche Dienste des Treibers in Anspruch nehmen.

Mit dem Befehl *strace* lassen sich alle Systemaufrufe anzeigen, die von einem Anwenderprogramm aufgerufen werden. Hierbei wird nicht nur der Aufruf selber, sondern auch alle Argumente und Rückgabewerte in symbolischer Form angezeigt. Im Fehlerfalle also der symbolische Wert des Fehlers (z.B. ENOMEM) und auch der passende String (Hier "Out of memory").

3.2.3 Explizites Auslesen gewünschter Information

Diese Methode bildet eine Mischform von Kernel- und Anwendungs-Debugging. Über gezielte Anfragen mit den üblichen Linux/Unix-System-Werkzeugen lässt sich ein aufgetretener Fehler schnell auf den User-Space Teil eines Programms, oder eben auf einen Kernel /Treiber-Fehler eingrenzen.

So kann uns *ps* dabei helfen, die Fehlerursache genau einzugrenzen (crsh_prg ist der Prozessname des abgestürzten Programmes, genauso ist hier die Angabe der PID möglich - wobei dann -C wegfällt):

>ps -u -C crsh_prg

liefert:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
snafu	1231	0.0	0.1	1200	312	pts/1	**D**	23:23	00:00	./crsh_prg

Interessant ist der Wert von STAT (Prozessstatus Code). Dieser Prozess ist in STATe D, „uninterruptible sleep“ (normalerweise E/A Operationen), was auf einen Fehler im Kernel/Treiber hindeutet, da crsh_prg hier im Kernel-Mode und im Zustand TASK_UNINTERRUPTIBLE ist.

Um den Fehler weiter einzugrenzen kann man folgenden *ps*-Aufruf verwenden:

>ps -o pid,tt,user,fname,wchan -C crsh_prg

Falls für den laufenden Kernel die passende Kernel-Symboltabelle *System.map* vorhanden ist (normalerweise in */boot*, wobei an den Dateinamen noch die Kernelversion angehängt wird, z.B. *System.map-2.4.24*) liefert dieser Aufruf dieses:

```
PID   TT      USER    COMMAND     WCHAN
1231  pts/1   Snafu   crsh_prg    noninterruptible_delay
```

WCHAN zeigt uns die Funktion an, in welcher der untersuchte Prozess blockiert ist. Mit Hilfe einer Volltextsuche über die Kernelquelltexte lässt sich die Datei und schließlich die Funktion finden, in der wir hängen geblieben sind. Diese kann man nochmals einer genauen Überprüfung im Quelltext unterziehen – oder gezielt einen Debugger (=> 3.3) verwenden um das Verhalten genauer zu untersuchen.

3.2.4 "Oops"-Auswertung

Eine Kernel-Oops-Meldung zeigt den Prozessor Status zum Zeitpunkt des Fehlers an. Dazu gehören der Inhalt der CPU-Register, die Lage der Seitendeskriptoren-Tabellen und andere, auf den ersten Blick unverständliche Informationen. Die Meldung wird durch *printk*-Anweisungen ausgegeben und, wie weiter vorn in Abschnitt 3.2.1 beschrieben, weitergeleitet.

Sämtliche Informationen werden als hexadezimale Werte ausgegeben. Um größeren Nutzen aus diesen Informationen zu ziehen, müssen sie in Symbole aufgelöst werden. Dafür gibt es Hilfsprogramme: *klogd* und *ksymoops*. Während *klogd* die Symbole automatisch decodiert (natürlich nur wenn *klogd* aktiviert ist), muss *ksysmoops* vom Benutzer explizit aufgerufen werden.

Beide Programme liefern dann Informationen, welche Funktion aufgerufen wurde bzw. in welchem Modul bei welchem Offset es zum Crash kam.

3.3 Kernel-Debugger – Patches und Ähnliches - kurz vorgestellt

3.3.1 gdb

Prinzipiell ist es möglich einen laufenden Kernel mit dem normalen GNU Debugger *gdb* zu untersuchen. Dazu muss gdb so aufgerufen werden, als wäre der Kernel ein Anwendungsprogramm. Um überhaupt arbeiten zu können benötigt man eine unkomprimierte Version des Kernel-Images, wobei der Kernel sinnvollerweise mit *-g* übersetzt worden sein sollte, um Debug-Informationen einzubinden.

Ein typischer Aufruf des gdb sieht also so aus:

>gdb /usr/src/linux/vmlinux /proc/kcore

Das erste Argument ist der volle Pfad des unkomprimierten Kernels, das zweite der Namen einer Core-Datei. In unserem Falle (laufender Kernel) ist diese Core-Datei das Kernel-Core-Image /proc/kcore.

kcore repräsentiert die *ausführbare Kernel-Datei*. Im *gdb* kann man nun die Kernel-Variablen mit den normalen *gdb*-Befehlen ansehen. Z.B. lässt sich mit dem Kommando *p jiffies* die Anzahl der System-Ticks seit dem Systemstart anzeigen.(Weitere Beschreibung der *gdb* Kommandos in 3.4 KGDB)
Es ist nicht möglich die Kernel-Daten zu ändern (es sind keine Schreibzugriffe in den Speicher erlaubt), Haltepunkte oder Beobachtungspunkte zu setzen oder Kernel-Funktionen im Einzelschrittmodus laufen zu lassen.

3.3.2 kdb – Patch

kdb ist ein integrierter Kernel-Debugger, der als nicht-offizieller Patch von [KDB] heruntergeladen werden kann. Um *kdb* verwenden zu können, muss der Kernel mit der passenden *kdb* -Version gepatcht, neu übersetzt und installiert werden.

Läuft der Kernel mit *kdb*, gibt es mehrere Möglichkeiten, den Debugger zu starten (Drücken der Pause/Break Taste, Haltepunkt wird erreicht, Kernel-Oops tritt auf)

Die Syntax von *kdb* ist weitgehend mit der des gdb identisch. *kdb* verfügt über umfangreiche Debugging-Fähigkeiten. So ist es möglich Daten zu manipulieren, Einzelschrittausführung zu nutzen (*kdb* arbeitet auf Maschinencodeebene), Haltepunkte bei Datenzugriff zu setzen, Code zu Disassemblieren und vieles mehr. Näheres in der *kdb* Dokumentation.

3.3.3 IKD Der Integrated Kernel Debugger – Patch [IKD]
Ist eigentlich eine Sammlung verschiedenster Werkzeuge um integriertes Kernel-Debugging zu ermöglichen- ebenso ein inoffizieller Kernel-Patch. Zur Nutzung muss der Kernel mit dem Patch (zur Kernelversion passend!) von [IKD] gepatcht, neu übersetzt und installiert werden.
Enthalten sind u.a. ein "Kernel Stack Debugger", "Stack Meter", "Soft Lockup"-Detektor und eine – im Normalfall – leicht veraltete Version des *kdb*.
Eine vollständige Beschreibung aller Features findet sich in der Dokumentation zu IKD.

3.3.4 Der kgdb – Patch [KGDB]
Der *kgdb*-Patch ermöglicht die uneingeschränkte Verwendung von *gdb* zur Untersuchung des laufenden Kernels. Dazu werden zwei Rechner über eine serielle Leitung verbunden, wobei auf dem einen (test/target-System) der zu debuggende Kern und auf dem anderen (development-System) *gdb* läuft.
Auf kgdb wird in 3.4 noch genauer eingegangen, da es zu den mächtigsten Werkzeugen im Bereich Kernel-Debugging gehört und auf den weitverbreiteten *gdb* aufsetzt.

3.3.5 Dynamic Probes – Patch [DProbes]
Dynamic Probes erlaubt das Einsetzen einer “Sonde” an fast jeder Stelle im System, sowohl im User als auch im Kernel-Space. Diese Sonden können Informationen an den User-Space zurückliefern, Register ändern oder eine Reihe anderer Dinge tun.
DProbes wurde von IBM entwickelt und unter die GPL gestellt. Es läuft mittlerweile auf einer Vielzahl von Plattformen.
Das LTT (LinuxTraceToolkit [LTT]), ein Werkzeugsatz der das Verfolgen aller Vorgänge im Kernel erlaubt und diese dann grafisch aufbereitet anzeigen kann, arbeitet mit DProbes zusammen.

3.3.6 Kernel Crash Dump Analyzer [LKCD/Crash]
Crash Dump Analyzer ermöglichen es dem System, seinen Zustand aufzuzeichnen, wenn ein Oops auftritt, so dass der Zustand hinterher mit den dazugehörigen Hilfsprogrammen nach Belieben analysiert werden kann.

3.3.7 Die User-Modus-Linux-Portierung (Patch) [UML]
Es handelt sich dabei um eine virtuelle Maschine, die unter Verwendung der Linux-Systemaufrufe implementiert ist. User-Modus-Linux erlaubt es dem Linux-Kernel als separater Prozess im User-Modus auf einem Linux-System zu laufen. Für Debugging-Zwecke ist die Tatsache wichtig, dass es sich nur um eine virtuelle Maschine handelt, so können selbst schwerwiegende Fehler im Kernel keine Beschädigung des echten Systems bewirken. Dass der User-Mode-Kernel einfach mit einem „normalen“ Debugger (gdb) untersucht werden kann – es handelt sich ja nur um einen normalen Prozess -, ist jedoch von größter Bedeutung.

Weitere und genauere Informationen auf der Website des Projekts [UML].

3.4 KGDB – ein Kernel-Debugger

Wie in 3.3.4 bereits angedeutet wurde ist der *kdgb* ein spezieller Kernel-Patch, der ein interaktives Debugging des Kernels mit gdb ermöglicht. Dazu werden zwei Rechner benötigt, die über eine serielle Verbindung verbunden werden.
Der Rechner, auf dem der mit *kgdb* gepatchte Kernel läuft, wird dabei als test-Maschine bezeichnet, während derjenige von dem aus die test-Maschine mit dem gdb untersucht wird, development-Maschine genannt wird.

Bisher werden allerdings für die test-Maschine nur x86 Architekturen unterstüzt (Unterstützung für PowerPC, x86_64 und s390 sind in der Entwicklung). *kgdb*-Patches sind ab Kernelversion 2.4.2.2 für einige 2.4er Kernel, sowie den 2.6.0 und 2.6.1 verfügbar. (Siehe [KGDB])

3.4.1 Wie funktioniert KGDB
Wenn ein Kernel mit *kgdb*-Unterstüzung gepatcht wird, werden einige native Kernel-Quelltexte verändert.
Hierbei werden in die IDT eine Exception (Interrupt Gate) und spezielle Traps (Trap Gate) eingefügt.
Die „neue“ Exception dient dazu bei einem INT 3 (Interruptvector 3, unmaskierbares Signal von der Seriellen Schnittstelle) den Kern anzuhalten, alle Register zu sichern und in den Debuggermodus zu wechseln. Ist diese interaktive Sitzung (im folgenden *remote-gdb*) wieder beendet, werden die Registerinhalte zurückkopiert und die Ausführung dort fortgesetzt wo sie unterbrochen wurde. (Das ist der Fall wenn man auf der Development-Maschine die Tastenkombination STRG-C drückt.)

Während einer remote-gdb-Sitzung ist es möglich den „erstarrten“ Kernel mit den Werkzeugen von *gdb* zu untersuchen. Nach dem „continue“ Kommando werden die gespeicherten Register wieder zurückkopiert und die Ausführung des Kernels wird dort fortgesetzt wo sie unterbrochen wurde.

Einzelschrittausführung:
... wird realisiert, indem an der Stelle nach der momentanen Quellcode-Anweisung (Der Kernel wurde ja mit -g kompiliert, also liegt die passende Information vor) eine trap-Anweisung eingefügt.
In dieser passiert folgendes:

- Registerinhalte werden auf den Stack gesichert

- eine weitere trap-Anweisung wird sofort wieder an die Stelle nach der nächsten Kernel-Quellcode-Anweisung einfügt
- die trap-Anweisung von der alten Position wird entfernt
- *remote-gdb wird* gestartet

Ist die interaktive Sitzung beendet, werden die (möglicherweise veränderten) Registerinhalte wieder zurückkopiert und die Ausführung des Kernels wird dort fortgesetzt wo sie unterbrochen wurde

Haltepunkte:
Analog zur Einzelschrittausführung wird hier an einer frei wählbaren Stelle eine trap-Anweisung eingefügt.

4. Literaturverzeichnis:

[UML]: **User-Mode-Linux**
http://usermodelinux.org/

[KDB]: **KDB (Built-in Kernel Debugger) - SGI - Developer Central Open Source**
http://oss.sgi.com/projects/kdb/

[IKD]: **Integrated Kernel Debugger**
http://www.kernel.org/pub/linux/kernel/people/andrea/ikd/
http://www.kernel.org/pub/linux/kernel/people/andrea/ikd/README

[KGDB]: **kgdb: linux kernel source level debugger**
http://kgdb.sourceforge.net/

[DProbes]: **Dynamic Probes – IBM Linux Technology Center**
http://www-124.ibm.com/developerworks/oss/linux/projects/dprobes/

[LTT]: **The Linux Trace Toolkit – OPERSYS Inc.**
http://www.opersys.com/ltt/index.html

[LKCD]: **Linux Kernel Crash Dumps**
http://lkcd.sourceforge.net/

[Crash]: **In Memory Core Dump – Mission Critical Linux**
http://oss.mclx.com/projects/mcore/

"kdb vs. kgdb" – "*which kernel debugger is 'best'?*"
http://kerneltrap.org/node/view/112

KGDB Internals – "A summary of how KGDB works"
http://www.superhlinux.com/kgdb/internals.php3

Kernel debugging, part 1,2,3: "Understanding what's gone wrong"
Daniel P. Bovet, Marco Cesati, Cosimo Comella –
http://buffer.antifork.org/linux/Kernel_Debugging_1.txt
http://buffer.antifork.org/linux/Kernel_Debugging_2.txt
http://buffer.antifork.org/linux/Kernel_Debugging_3.txt

Bovet D.P.; Ceasti M.: **Understanding the Linux Kernel** – 2nd Edition (2002) - O'Reilly & Associates

Zeller A., Krinke J.: - **Programmierwerkzeuge** - dpunkt Verlag - Juni 2000

Rubini A. & Corbet J.: **Linux-Gerätetreiber** - 2. Auflage - O'Reilly & Associates - April 2002