Debugger und Kerneldebugger

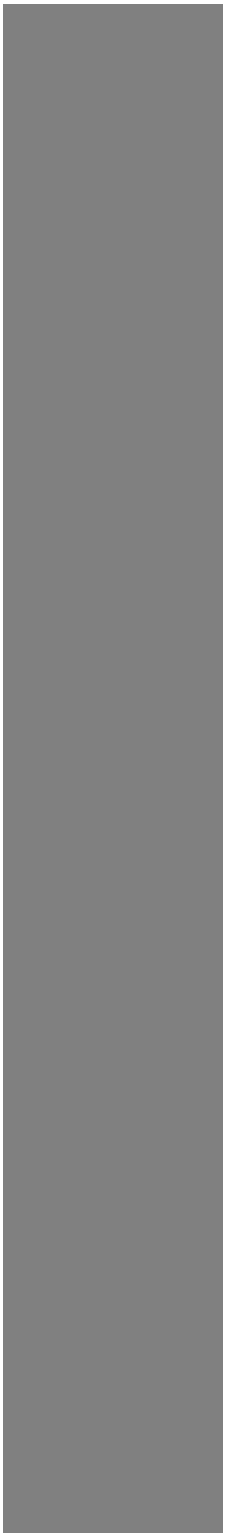
Universität des Saarlandes

Seminar

von
Martin Mehlmann
mehlmann@st.cs.uni-sb.de

Lehrstuhl für Rechnerarchitektur und Parallelrechner
Prof. Dr. Wolfgang J. Paul
FB 14 Informatik
Universität des Saarlandes

Saarbrücken, 05 Oktober 2004



Inhaltsverzeichnis

0pt0.5pt

1 Begriffsklärung Bug

1.1 Der erste Bug

Der erste Bug wurde am 9. September 1945 von der Compilerbauingeneurin Grace Hopper bei der Fehlersuche in einem Mark 2 Rechner gefunden. Hierbei handelte es sich um eine Motte, die sich, angelockt von einem Kathodenstrahler, mit ihrem Flügeln im Kartenleser verfangen hatte und somit das Einlesen der Lochkarte blockierte. Die resultierenden Begriffe für die Informatik waren Bug für Fehler und Debugging für Fehlersuche und Fehlerbehebung. Ein Bug kann einen Fehler in Hardware oder einen Fehler in Software bezeichen.

1.1.1 Fehler in Hardware

Bei Fehlern in Hardware unterscheidet man zwischen einem

- physikalischem Fehler
- Fehler im Entwurf/Design

1.1.2 Fehler in Software

Bei Fehlern in Software unterscheidet man zwischen einem

- Fehler im Quellcode
- Fehler im Programmzustand
- Fehler im Entwurf/Design

Die Bezeichnung Bug als Fehler in Software ist sehr vage und lässt sich weiter klassifizieren. Hierzu betrachtet man den Vorgang vom Starten bis zum Fehlschlagen eines Programms.

- Der Programmierer hinterlässt einen Defekt im Quellcode.
- Wenn dieser defekte Code ausgeführt wird, ist das Resultat eine Infektion des Programmzustandes.
- Die Infektion propagiert sich eventuell über mehrere Variablen und breitet sich aus.
- Die Infektion resultiert in einem vom Benutzer beobachteten Fehlschlagen des Programms.

Hiermit bezeichnet das Fehlschlagen ein vom Benutzer beobachtetes Verhalten des Programms das nicht dem erwarteten Verhalten entspricht. Allerdings ist nicht jedes Verhalten, dass den Erwartungen des Benutzers widerspricht ein Bug in der Software, sondern kann durchaus von den Entwicklern so gewollt worden sein (a bug or a feature?). Um eine Infektion im Programmzustand auszumachen, sollte eine vollständige Spezifikation des Programmzustandes zu jedem Zeitpunkt vorliegen.

1.2 Notwendigkeit von Debugging

Nicht jeder Defekt im Programmzustand muss in einem Fehlschlagen resultieren, denn

- Nicht jeder Defekt resultiert in einer Infektion.
  Wenn ein existierender defekter Code in keinem Testlauf ausgeführt wird, so resultiert er auch in keiner Infektion.
- Nicht jede Infektion resultiert in einem Fehlschlagen.
  Wenn sich eine Infektion nicht bis zum Fehlschlagen propagiert, da sie vorher geheilt wird oder wenn eine Infektion aufgrund eines anderen Defekts sich vorher im Fehlschlagen des Programms äussert.

Dijkstra hat schon 1972 erkannt, dass Testen nur die Anwesenheit von Fehlern zeigen kann, nicht aber deren Abwesenheit. Um dennoch möglichst viele Defekte im Quellcode zu finden, muss eine Testsuite vorhanden sein, die eine möglichst hohe Abdeckung (engl. coverage) des Quellcodes realisiert. Um allerdings die Abwesenheit von Defekten im Quellcode garantieren zu können muss die verwendete Software verifiziert worden sein (siehe [VS]) . Solange nicht alle verwendete Software und die darunterliegende Hardware verifiziert wurde, wird es weiterhin eine Notwendigkeit für das Debugging von Programmen geben.

2 Begriffsklärung Debugging

Jedes Fehlschlagen eines Programms kann auf eine Infektion im Programmzustand zurückgeführt werden. Jede Infektion wiederum kann auf einen Defekt zurückgeführt werden. Hierbei sind als Ausnahme bestimmte physikalische Einflsse, wie z.B. kosmische Strahlung oder ähnliches zu nennen, welche zu einer spontanen Infektion beispielsweise durch einen Bitflip führen können, obwohl kein Defekt vorliegt. Die Wahrscheinlichkeit solcher Phänomene ist allerdings in normaler Büroumgebung so gering, dass deren Einfluss vernachlässigt werden kann.
Debugging bedeutet ein beobachtetes Fehlschlagen einem Defekt zuzuordnen und diesen zu entfernen. Debugging ist somit ein zweidimensionales Suchproblem. Hierbei sucht man

- in der Zeit (temporal)
  Das Ziel ist es, den Beginn der Infektion zu bestimmen, d.h. den Zeitpunkt

während des Programmlaufs, an dem der Zustandsübergang von gesund nach infiziert aufgrund eines Defekts passiert.

- in den Variablen (lokal)
  Das Ziel ist es die Programmvariablen zu einem bestimmten Zeitpunkt während des Programmlaufs in gesunde und infizierte Variablen aufzuteilen.

2.1 Vorgehensweise

Die allgemeine Vorgehensweise beim Debugging umfasst die folgenden Punkte

- Beobachte das Fehlschlagen des Programms
- Reproduziere das Fehlschlagen unter denselben Umständen
  Dieser Vorgang lässt sich häufig automatisieren durch capture and replay Werkzeuge, wie z.B expect und autoexpect (siehe [EA]). Allerdings kann es teilweise sehr schwierig sein alle Umstände zeitlich exakt zu reproduzieren. Manschmal ist es sogar unmöglich ein Fehlschlagen zu reproduzieren, wenn eine Anwendung beispielsweise echte Zufallszahlen, z.B aus `/dev/rand` oder `/dev/srand` verwendet.
- Vereinfache die Umstände
  Dieser Vorgang ist ebenfalls meist automatisierbar. Hierzu kann man z.B Deltadebugging auf einer Fehlerinduzierenden Eingabe durchführen (siehe [DD]).
- Lokalisiere den Defekt
  Die eigentliche Suche im Programmzustand nach dem Defekt lässt sich ebenfalls mithilfe von Deltadebugging auf Speichergraphen (engl. memory graphs) automatisieren (siehe [DD]). Dabei wird ein Programmzustand durch einen gerichteten Graphen repräsentiert (siehe [MG]).
- Beseitige den Defekt
  Die Beseitigung eines Defekts, sollte den eigentlichen Grund und nicht nur die Symptome beheben. Ausserdem ist darauf zu achten, dass ein Beheben des Defekts auf ein bestimmte Art nicht zu Problemen, Inkonsistenzen oder neuen Defekten führt. Dementsprechend sollte die Art der Behebung des Defekts möglichst generell sein (engl. most general fix).

2.2 Unterscheidung

2.2.1 Statisches Debugging

Hierbei handelt es sich um einen deduktiven Prozess. Man versucht vom abstrakten Programmcode auf Ergebnisse konkreter Programmläufe zu schliessen. Es bezeichnet also Debugging anhand des Quelltextes ohne das zu debuggende Programm auszuführen.

2.2.2 Dynamisches Debugging

Hierbei handelt es sich um einen induktiven Prozess. Man versucht von den Ergebnissen aus konkreten Programmläufen auf den abstrakten Quellcode zu verallgemeinern.
Es bezeichnet also Debugging zur Laufzeit durch Beobachten von einzelnen Programmläufen und Debugging nach dem Fehlschlagen eines bestimmten Programmlaufs (Post Mortem Debugging).

3 Begriffsklärung Debugger

Ein Debugger ist generell eine Software, welche das Debugging anderer Software ermöglicht. Eine etwas strengere Definition stellt bestimmte Anforderungen bezüglich der Funktionalität an die Software, welche als Debugger bezeichnet wird. Ein Debugger dient zur kontrollierten Ausführung eines beobachteten Programmes. Hierbei ist es dem Debugger möglich das kontrollierte Programm zu überwachen und dessen Zustand zu manipulieren.

3.1 Unterscheidung

3.1.1 Debugger auf Maschinenebene

Debugger auf Maschinenebene verfolgen das Programmgeschehens auf Maschinensprachenebene. Dies hat den Vorteil, dass Software, deren Quellcode nicht verfügbar ist, oder die nicht mit Debuggingunterstützung übersetzt wurde, untersucht werden kann. Der Nachteil liegt darin, dass die Person, die ein Programm Debugging möchte Kenntnis über die Maschininensprache der Rechnerarchitektur haben muss und sich anhand dieses low level Assemblercode zurechtfinden muss.

3.1.2 Debugger auf Quellsprachenebene

Debugger auf Quellsprachenebene ermöglichen eine Verfolgung des Programmgeschehens auf Quellsprachenebene. Der Vorteil besteht darin, dass Quellcodeblöcke oder Funktionen beim Durchlaufen eines beobachteten Programmes übersprungen werden können. Ausserdem lassen sich Ausdrücken in der Quellsprache durch den Debugger auswerten. Um ein Programm mithilfe eine Debuggers auf Quellsprachenebene Debugging zu können, muss das zu debuggende Programm mithilfe von Debugunterstützung übersetzt worden sein. Dies hat den Nachteil, dass der Quellcode zum Übersetzen mit Debugsymbolen vorliegen muss und dass die resultierende ausführbare Datei erheblich grösser sein kann. Solche Debuginformationen umfassen unter anderem die Symboltabelle, welche alle Variablennamen und Funktionsnamen verwaltet. Zusätzlich beinhalten sie ein Matching zwischen Quellcode und dem korrespondierenden architekturabhängigen Assemblercode.
Ein sehr bekannter und beliebter Quellsprachen Debugger unter Linux vorwiegend für C und C++ ist der GNU Debugger GDB oder sein beliebtestes graphisches Frontend Data Display Debugger (DDD).

3.2 Debugger Grundprinzipien

Jeder Debugger muss mindestens den folgenden Prinzipien genügen

- Das Heisenberg-Prinzip
  Die Verwendung eines Debuggers darf das Verhalten des untersuchten Programms nicht beeinflussen. Das untersuchte Programm beispielsweise ist auf einen Teil des Speichers angewiesen, der aber auch vom Debugger verwendet wird. Dies darf den Programmlauf des untersuchten Programmes nicht beeinflussen, so dass der Debugger dadurch falsche Ergebnisse an den Benutzer liefert.
- Das Wahrheitsprinzip
  Die Informationen, die der Debugger liefert, müssen wahrheitsgemäss ein. Beispielweise könnte eine Variable in ein Register geladen und verändert worden sein, der Debugger aber den alten Wert aus dem Speicher holen, bevor der neue zurückgeschrieben wurde.

3.3 Debugger Funktionalität

3.3.1 Anzeigen des Programmzustands

Im folgenden wird die Kernfunktionalität eines Debuggers vorgestellt. Die einzelnen Funktionen lassen sich entsprechend der Definition von Debugging als zweidimensionales Suchproblem klassifizieren.

Suche in der Zeit

- Haltepunkte
  Haltepunkte (engl. breakpoints) halten das untersuchte Programm zu einem bestimmtem Zeitpunkt während des Programmlaufs an und geben die Kontrolle an den Debugger zurück.
- Einzelschrittausfhrung
  Einzelschrittausführung ist sinnvoll um den genauen Programmlauf auf Anweisungsebene in der Quellsprache oder auf Maschineninstruktionsebene zu verfolgen. Hierbei unterscheidet man bei Quellsprachendebugger zwei Methoden:
  - step
    Die nächste Anweisung des untersuchten Programms wird ausgeführt und die Kontrolle an den Debugger zurückgegeben. Falls es sich bei dieser Anweisung um einen Funktionsaufruf handelt, so wird die Einzelschrittausführung in der aufgerufenen Funktion fortgesetzt.
  - next
    Ähnlich wie step, falls es sich aber bei der nächsten Anweisung des untersuchten Programms um einen Funktionsaufruf handelt, so wird die Funktion ausgeführt und erst dann wird die Kontrolle an den Debugger zurück gegeben

- Programmzähler (engl. instruction pointer)
  Informationen über den genauen Programmfluss liefert auch der Programmzähler. Dieser kann beim Fehlschlagen eines Programms auf die betreffende Quellcodezeile verweisen, um einen ersten Hinweis auf den Fehler geben zu können.

Suche in den Variablen

- Werte von Variablen
- Funktionsstapel (engl. call stack)
- Registerinhalte

Suche in Zeit und Variablen

- Datenhaltepunkte (engl. watchpoints)
- Bedingte Haltepunkte

3.3.2 Ändern des Programmzustands

- Werte von Variablen
- Funktionsstapel
- Programmzähler
- Registerinhalte

3.4 Debugger Funktionsweise

Das Betriebsystem führt Programme als Prozesse aus. Hierbei bekommt jeder Prozess seinen virtuellen Adressraum zugeordnet. Das Betriebssystem sorgt dafür, dass die entsprechenden physikalischen Adressräume dieser Prozesse disjunkt sind. Dies führt dazu, dass es keinem Prozess möglich ist auf den Prozessraum eines anderen Prozesses zuzugreifen und lässt sich begründen durch fundamentale Sicherheitsanforderungen. Eine Ausnahme ist sog. shared memory, bei dem es möglich ist einen Speicherraum zwischen mehreren Prozessen aufzuteilen um dessen gemeinsame Nutzung durch Prozesse zu ermöglichen.
Der Debugger und das untersuchte Programm sind separate Benutzerprozesse auf der gleichen oder verschiedenen Maschinen mit seperaten geschützten Addressräumen. Die erforderliche Kommunikation erfolgt über eine vom Kernel des Betriebssystemns zur Verfügung gestellte Debugschnittstelle. Diese dient zur Kontrolle der Ausführung von Prozessen durch andere Prozesse.
Linux stellt zu diesem Zweck den system call `ptrace` zur Verfügung. Jede Interaktion zwischen dem Debugger und dem untersuchtem Programm läuft ber die folgende Schnittstelle:

```
long ptrace(enum request, pid_t pid, void* addr, void* data)
```

Der erste Parameter spezifiziert den Typ der Anfrage:

- Vorbereitung des untersuchten Prozesses
  `PTRACE_TRACEME`
- Zugriff auf Register des untersuchten Prozesses
  `PTRACE_GETREGS`
  `PTRACE_SETREGS`
- Zugriff auf Speicher des untersuchten Prozesses
  `PTRACE_PEEKTEXT, PTRACE_PEEKDATA`,
  `PTRACE_POKETEXT, PTRACE_POKEDATA`
- Fortsetzen und Einzelschrittausfhrung des untersuchten Prozesses
  `PTRACE_SYSCALL`
  `PTRACE_CONT`
  `PTRACE_SINGLESTEP`
- Beenden des untersuchten Prozesses
  `PTRACE_KILL`

Die Funktionweise des Debuggers unter Verwendung des system calls `ptrace` wird im Folgenden erklärt.

- Der Debugger (Vaterprozess) ruft den system call `fork()` auf und erzeugt somit ein Kind, welches im wesentlichen ein Klon des Vaters ist, allerdings eine andere PID (Prozess id) besitzt.
- Danach ruft der Debugger einen wait system call auf,
  wie z.B. `waitpid(child_pid, &ret_val, 0)` und geht in einen Wartezustand.
- Im weiteren wartet er auf das SIGCHLD Signal vom Kernel, dass ihm signalisiert, dass der Kindprozess gestoppt oder beendet wurde. Der Rückgabewert `ret_val` liefert ihm den Grund (das Signal) aufgrund dessen der Kindprozess gestoppt wurde.
- Jetzt kann er `ptrace` verwenden, um den Kindprozess zu kontrollieren.
- Schliesslich geht er nach erneutem Aufruf eines wait system calls wieder in den Wartezustand.

Für den Vaterprozess ist eine Änderung der Semantik der wait system calls ntig. Diese Funktion wird dahingehend modifiziert, dass sie nicht nur zurückkehrt, wenn der Kindprozess beendet wurde, sondern auch, wenn dieser gestoppt wurde.

- Der Kindprozess meldet sich durch den Aufruf `ptrace(PTRACE_TRACEME, 0, 0, 0)` beim Kernel für Tracing an. Der Kernel erlaubt dadurch dem Vaterprozess die Kontrolle über den Kindprozess.
- Daraufhin wird das Speicherbild des Kindes mithilfe eines exec system calls, wie z.B `execve(debuggee, NULL)` durch das Speicherbild des untersuchten Prozesses (debuggee) ersetzt.
- Der Kindprozess stoppt bei jedem empfangenen Signal, ausser bei bestimmten Signalen, die ihn fortsetzen, nachdem er gestoppt wurde. Nach dem Stoppen wird das empfangene Signal vom Kindprozess standardmäsig ignoriert.
- Der Vaterprozess wird durch den Kernel durch das SIGCHLD Signal über das Stoppen des Kindes informiert.

Für den Kindprozess ist eine Änderung der Signalbehandlung notwendig, so dass beim Empfang eines Signals das Kind gestoppt wird und der Vater durch Rückkehr des modifizierten wait system calls informiert wird. Ausserdem sind erweiterte Prozess-Kontroll Mechanismen erforderlich, die das stoppen und dispatchen des Debuggers und des getraceten Prozesses ermöglichen.
Im Folgenden wird die Verwendung von `ptrace` durch den Debugger zur Realisierung eines Haltepunkts erläutert.

3.4.1 Setzen eines Haltepunkts

Angenommen, der Debugger möchte einen Haltepunkt an Adresse 0xBEAF setzen. Falls Debugregister für Haltepunkte vorhanden sind, so wird eines dieser Register verwendet und die Adresse 0xBEAF darin abgelegt (Hardwarehaltepunkt). Hierbei handelt es sich um spezielle vom Prozessor zur Verfügung gestellte Register, die nicht bei jedem Prozessor vorhanden sein müssen.
Falls kein Hardwarehaltepunkt verfügbar ist, wird die Instruktion, die unter der Adresse 0xBEAF zu finden ist, gesichert und durch eine sog. trap instruction ersetzt (Softwarehaltepunkt).

```
int rc;
if (Haltepunkt-Register frei) {
  rc = ptrace (PTRACE_SETREGS, 0, 0xBEAF)
} else {
  saved_instruction = ptrace (PTRACE_PEEKTEXT, 0xBEAF, 0);
  rc = ptrace (PTRACE_POKETEXT, 0xBEAF, break_instruction );
}
```

3.4.2 Erreichen eines Haltepunkts

Beim Erreichen eines Haltepunkts löst das Debugregister oder die trap instruction eine Trap (Softwareinterrupt) aus, welche vom Kernel verarbeitet wird. Der Kernel

sendet das SIGTRAP Signal an den Kindprozess und lässt diesen an Adresse 0xBEAF stoppen. Daraufhin wird der Debugger vom Kernel durch das SIGCHLD Signal über das Stoppen des Kindes informiert. Der Debugger versichert sich, dass das untersuchte Programm aufgrund eines Haltepunktes gestoppt wurde und stellt die gesicherte Instruktion wieder her. Nun kann der Debugger unter Verwendung von `ptrace` die Werte aus dem Speicher oder aus Registern des angehaltenen Kindprozesses holen um den Benutzer über den Zustand des Programms zu informieren. Ebenso kann der Werte von Variablen oder Register ändern, um den Zustand des Programms zu manipulieren.

```
int rc;
if ( !(Aufruf von Register) ) {
  rc = ptrace (PTRACE_POKETEXT, 0xBEAF, saved_instruction);
}
```

3.4.3 Fortsetzen des untersuchten Programms

Falls ein Debugregister für den Haltepunkt verwendet wurde, wird das untersuchte Programm sofort fortgesetzt. Ansonsten wird zuerst eine einzelne Anweisung ausgeführt, um anschliessend die trap instruction wiederherzustellen, damit beim nächsten Erreichen dieser Adresse das Programm wieder gestoppt wird. Danach wird das beobachtete Programm fortgesetzt.

```
int rc;
if ( Unterbrechung wurde von Haltepunkt-Register ausgelst ) {
  rc = ptrace (PTRACE_CONT, 0, 0);
} else {
  rc = ptrace (PTRACE_SINGLESTEP, 0, 0);
  rc = ptrace (PTRACE_POKETEXT, 0xBEAF, break_instruction);
  rc = ptrace (PTRACE_CONT, 0, 0);
}
```

3.4.4 Einzelschrittausfhrung

Falls Debugregister für Einzelschrittausführung verfügbar sind, so wird eines dieses Register verwendet. Das trap flag im Debug Register wird gesetzt, welches eine Trap nach jeder Instruktion auslst. Der Kernel sendet das SIGTRAP Signal an den Prozess, lässt diesen stoppen und informiert den Debugger durch das SIGCHLD Signal darüber. Falls keine Hardwareunterstützung verfügbar ist, wird ein Haltepunkt nach jeder Instruktion gesetzt.

3.4.5 Datenhaltepunkte

Falls Debugregister für Datenhaltepunkte verfügbar sind, so wird eines dieser Register verwendet. Das trap flag im Debug Register wird gesetzt, welches eine Trap auslöst,

sobald sich der Wert an der registrierten Adresse ändert. Der Kernel sendet das SIGTRAP Signal an den Prozess und lässt diesen stoppen. Der Debugger wird wiederum durch das SIGCHLD Signal vom Kernel informiert. Falls keine Debugregister für Datenhaltepunkte vorhanden sind oder der beobachtete Ausdruck ausgewertet werden muss, wird Einzelschrittausführung verwendet. In diesem Fall wird nach jeder Instruktion auf eine Änderung des Wertes getestet, was den Nachteil hat, dass der Programmlauf des Kindprozesses stark verlangsamt wird. Als Richtwert kann ein Faktor von 1000 angenommen werden.

4 Debugging Werkzeuge

4.1 strace

4.1.1 Funktionalität von strace

strace dient zum Beobachten eines Prozesses. Es ermöglicht die Verfolgung aller system calls, die vom beobachteten Prozess aufgerufen werden und protokolliert für jeden Aufruf den Name, die Argumente und den Rückgabewert des system calls. Ausserdem werden alle vom Prozess empfangenen Signale protokolliert. Somit erhält man Informationen über die Interaktion des Programms mit dem Kernel. Das untersuchte Programm muss keinen besonderen Anforderungen wie z.B. Debugsymbolen genügen. Der Nachteil für die Verwendung als Debugging Werkzeug besteht darin, dass nur die Schnittstelle zwischen `libc`, welche unter Linux die system calls zur Verfügung stellt und dem Kernel untersucht werden kann.

4.1.2 Funktionsweise von strace

strace wird unter Verwendung von ptrace realisiert und funktioniert wie folgt: Zuerst weist strace das nach `fork()` entstandene Kind an, ihm durch `PTRACE_TRACEME` das Recht zur Kontrolle zu geben. Der Kernel sendet das SIGSTOP Signal an Kindprozess um diesen anzuhalten und das SIGCHLD Signale an strace um es über das Stoppen des Kindes zu informieren. Jedes vom untersuchten Prozess empfangene Signal lässt diesen stoppen und strace wird das vom Kind empfangene Signal weitergeleitet. Es braucht dieses also nur zu protokollieren. Nach jedem emfangenen Signal oder aufgerufenen system call kann strace das beendetet Kind fortsetzen. Hierzu wird insbesondere die Anfrage `PTRACE_SYSCALL` verwendet, welche den Kernel anweisst das Kind nach jedem aufgerufenen system call zu stoppen und es strace ermöglicht den Systemcall zu protokollieren.

5 Kernel Debugging

5.1 Problematik

Die Problematik beim Kernel Debugging liegt darin, dass sich der Kernel nicht wie ein beliebiger Prozess kontrollieren lässt. Es gibt keine Instanz, welche die Ausführung des Kernels kontrolliert, oder in der Lage ist, den Kernel anzuhalten, ausser dem Kernel selbst. Der Kernel läuft in eigenem speziell geschützten Adressraum, so dass es keinem Prozess möglich ist auf den Addressraum des Kernels zuzugreifen. Der Grund dafür liegt in elementaren Sicherheitsanforderungen. Die hat allerdings den Nachteil, dass die gängige Debuggerfunktionalität schwieriger zu erreichen ist und nicht wie oben beschrieben realisiert werden kann.

5.2 Möglichkeiten

5.2.1 Kernel-Oops Meldungen

Kernel-Oops Meldungen sind die Fehlermeldungen des Kernels und werden durch eine Ausnahme (engl. exception) ausgelöst. Sie enthalten kodierte Informationen zum Zustand von CPU Registern und Kernelstapel zu einem bestimmten Zeitpunkt. Oops Meldungen können zu einer Konsole oder zum Kernel Ring Puffer gesendet werden. `klogd` entnimmt diese Meldungen aus den Ring Puffer und sendet sie weiter an `syslogd`. Mithilfe von `ksymoops` lassen sich Kernel-Oops Meldungen dekodieren und die dekodierten Informationen anzeigen. Diese Informationen umfassen unter anderem

- den Grund für die im Kernel geworfene Ausnahme
- den Oops Counter
- den Wert des Programmzählers (EIP)
- Informationen zu den EFLAGS und Inhalt der CPU Register
- dem Kernelstapel inklusive call tree der Kernelfunktionen

5.2.2 Linux Trace Toolkit

Bei LTT handelt es sich um einen Kernel Patch inklusive eines Hilfprogramms, welches im Benutzer Adressraum läuft. Nach Anwenden des Patches auf den Kernel lassen sich Ereignisse im Kernel besser verfolgen. Der Kernel kann Ereignisse protokollieren und gibt diese an ein bestimmtes Kernel Modul weiter, welches diese in einem Puffer speichert. Das Hilfsprogramm hat Zugriff auf diesen Puffer und liest ihn in regelmässigen Zeitabständen aus. Die gelesenen Informationen werden aufbereitet und dem Benutzer in lesbarer Form dargestellt. Die Ergebnisse erhalten insbesondere Timinginformationen, welche Aufschluss geben, was zu welchem Zeitpunkt im Kernel passiert ist. Die gewonnenen Informationen können zum Einkreisen von Performanzproblemen und somit zum Finden von Bottlenecks im Kernel dienen.

5.2.3 Kernel Debugging mit GDB

Der laufender Kernel kann mit GNU Debugger GDB untersucht werden. Hierzu wird der GDB mit dem Aufruf `gdb /usr/src/vmlinux /proc/kcore` gestartet. Dabei ist `/usr/src/vmlinux` die unkomprimierte Version des Kernels mit Debugsymbolen und `/proc/kcore` der Name der Coredatei, welche den momentan laufenden Speicher des Kernels enthält. `/proc/kcore` wird wie alle Dateien im /proc/ Verzeichnis erst beim Auslesen erzeugt. Der Kernel wird beim Debugging mit dem GDB nicht angehalten. Vom Kernel eingelesene Daten werden von GDB intern zwischengespeichert. Eine wiederholte Anfrage nach den selben Daten liefert die gespeicherten Daten aus dem Zwischenspeicher. Somit muss zum Erhalten von aktuellen Werten jedesmal der Zwischenspeicher reinitialisiert werden. Dies geschieht mithilfe des GDB Kommandos `core-file /proc/kcore`.
Die Nachteile dieser Methode den Kernel zu Debugging bestehen darin, dass keine Änderung des Kernelzustands durch den GDB möglich ist. Desweiteren lässt sich keine kontrollierte Ausführung des Kernels durch den GDB realisieren, da der GDB nicht schreibend und nur über einen Zwischenspeicher lesend auf den Kernelspeicher zugreifen kann.
Der Vorteil liegt darin, dass diese Methode keine weiteren Anforderungen hat, d.h. es muss ausschlieslich der GDB und ein gängiger Linux Kernel vorhanden sein.

5.2.4 Kernel Debugging mit User Mode Linux

Eine bessere Methode den Kernel zu Debugging ist mithilfe einer virtuellen Maschine, wie z.B. VMware oder UML. Bei UML handelt es sich um die Portierung eines Linux Kernels als virtuelle Maschine. Hierbei wird die Hardwareschicht der virtuellen Maschine durch system calls an das Hostsystem emuliert. Das virtuelle Linux inklusive virtuellem Kernel läuft als separater Prozess im Benutzer Adressraum auf einem Host System und wird daher vom Kernel des Hostsystems wie ein normaler Prozess behandelt. Seit Kernel 2.5 ist UML Teil der Kerneldistribution. Im folgenden wird kurz umrissen, was passiert, wenn die virtuelle Maschine einen system call aufruft:

- Ein virtueller Prozess, der in der virtuellen Maschine läuft führt eine system call instruktion aus.
- Mithilfe von ptrace wird der tracende Prozess des Hostsystems aufgeweckt. Der tracende Prozess annuliert den Funktionsaufruf im Namen des virtuellen Prozesses und weist den Kernel des Hostsystems an, den system call auszuführen.
- Der system call wird ausgeführt und nach dessen Beendigung wird der tracende Prozess wieder aufgeweckt.
- Der tracende Prozess manipuliert den Zustand des virtuellen Prozesses mithilfe von `ptrace` so, dass dieser denkt, er habe den system call beendet.
- Der virtuelle Prozess wird fortgesetzt.

Die Verwendung von UML zum Kerneldebugging hat signifikante Vorteile:

- Ein defekter Kernel kann nicht das Hostsystem zerstören.
- Verschiedene Konfigurationen können leicht auf dem gleichen Rechner getestet werden.
- Der virtuelle Kernel kann durch den GDB ohne Einschränkungen manipuliert werden.

Ein Nachteil besteht darin, dass die Korrektheit des Hostsystems gegeben sein muss, so dass das Debugging des virtuellen Kernels durch Fehler im Kernel des Hostsystems erschwert oder verfälscht werden kann. Auerdem sind für die virtuelle und die Host Maschine zusätzliche Bibliotheken nötig, die die Kommunikation über die system calls realisieren. Diese dürfen die Ergebnisse nicht verfälschen (siehe Heisenberg Prinzip).

6 Kernel Debugger

6.1 KGDB

KGDB ist ein Quellspachen Debugger für den Linux Kernel. Er wird zusammen mit GDB verwendet, um den Kernel zu debuggen. Kernel Entwickler können einen Kernel fast wie einen normalen Prozess durch die Verwendung von KGDB debuggen. Es ist möglich Haltepunkte im Kernelcode zu setzen, per Einzelschrittausführung durch den Code zu navigieren und Variablen zu beobachten. Es werden 2 Rechner benötigt, um KGDB verwenden zu können. Eine dieser Maschinen ist die Entwicklungsmaschine, die andere die Testmaschine. Diese sind durch eine serielle Verbindung verbunden, einem null-modem Kabel, was sie über ihre seriellen Ports verbindet. Der zu debuggende Kernel läuft auf der Testmaschine, während der GDB auf der Entwicklungsmaschine läuft. Die serielle Verbindung wird vom GDB verwendet um mit dem zu debuggenden Kernel zu kommunizieren. KGDB ist ein Kernel Patch. Er muss auf den Kernel angewendet werden, um Kernel Debugging zu ermöglichen. KGDB fügt die folgenden Komponenten zum Kernel hinzu:

- GDB stub
  Der GDB stub is das Herz des Debuggers. Er ist der Teil, der ankommende Anfragen vom GDB auf der Entwicklungsmaschine behandelt. Er kontrolliert alle Prozessoren auf der Zielmaschine wenn der Kernel der darauf läuft unter Kontrolle des Debuggers ist.
- Modifikationen an der Fehlerbehandlung
  Der Kernel gibt die Kontrolle an den Debugger ab, wenn ein unerwarteter Fehler passiert. Ein Kernel, der keinen KGDB Patch enthält, würde mit Kernel Panik bei unerwarteten Fehler reagieren. Änderungen an den Fehlerhandlern erlauben es Kernel Entwicklern unerwartete Fehler zu analysieren.

- Serielle Kommunikation
  Diese Komponente verwendet einen seriellen Treiber im Kernel und stellt eine Schnittstelle für den GDB zum GDB stub im Kernel zur Verfügung. Sie ist verantwortlich für das Senden und Empfangen der Daten über die serielle Schnittstelle. Diese Komponente ist ebenso verantwortlich Anfragen vom GDB zu behandeln, die den Kontrollfluss betreffen.

7 Verweise

- **[EA]** - expect und autoexpect, siehe http://expect.nist.gov/
- **[DD]** - Delta Debugging, siehe http://www.st.cs.uni-sb.de/dd/
- **[MG]** - Speichergraphen (engl. memory graphs), siehe http://www.st.cs.uni-sb.de/memgraphs/
- **[VS]** - Verisoft, siehe http://www.verisoft.de/

8 Glossar

- **GDB** - GNU Debugger GDB, siehe http://www.gnu.org/gdb/gdb.html
- **DDD** - Data Display Debugger, siehe http://www.gnu.org/ddd/
- **KGDB** - Kernel GNU Debugger KGDB, siehe http://kgdb.linsyssoft.com/
- **LTT** - Linux Trace Toolkit, siehe ???
- **LKCA** - Linux Kernel Crash Analyzer, siehe ???
- **VMware** - VMware, siehe http://www.vmware.com/
- **UML** - User Mode Linux, siehe http://user-mode-linux.sf.net