Anleitung zur

Fehlersuche mit Hilfe eines core-Dumps

Berufsakademie Stuttgart, Fachrichtung Nachrichtentechnik,
Labor "Rechnerorganisation" im 6. Studienhalbjahr

Stand 28. Februar 2007

`http://home.ntz.de/phleps/programming/coredump.pdf`

1 Was ist ein core-Dump?

Bei Programmabstürzen oder -abbrüchen entstehen oft Dateien mit dem Namen *core*. Manchen Entwicklern entlockt so ein *"core-Dump"* eine hilflose Geste. Sie betrachten ihn als Ergebnis eines besonders schwer zu findenden Fehlers, da er meist zusammen mit der Fehlermeldung *"Segmentation fault (core dumped)"* oder *"Bus error (coredump)"* auftritt.
In Wirklichkeit ist eine *core*-Datei eine große Hilfe bei der Fehlersuche: Sie ist ein Speicherabzug – engl. "dump" – des Programms zu dem Zeitpunkt, an dem der Fehler bemerkt wurde. Der Speicherabzug ist eine Momentaufnahme mit dem Inhalt aller Register sowie der Segmente für Code, Daten und Stack. Wenn die Quelldateien des Programms zur Verfügung stehen und das Programm mit der Debug-Option übersetzt wurde, kann der Speicherabzug mit Hilfe eines Debuggers untersucht werden.

2 Wann entsteht ein core-Dump?

Die meisten UNIX-Systeme legen eine core-Datei an, wenn ein Programm wegen bestimmter Signale bzw. wegen einer Exception abgebrochen wird.
Bei manchen UNIX-Systemen (z. B. bei Linux) wird die core-Datei jedoch nur angelegt, wenn ihre Größe einen eingestellten Grenzwert nicht überschreitet. Unter Linux kann die maximal erlaubte Größe einer core-Datei mit dem Befehl **`ulimit -c`** angezeigt und mit dem Befehl **`ulimit -c`** *Größe in KB* verändert werden.

Unter anderem bei folgenden Signalen wird – als vordefinierte Standard-Reaktion – das Programm abgebrochen und ein core-Dump geschrieben:

SIGABRT	abort	Abbruch durch Aufruf der Funktion **`abort()`**
SIGBUS	bus error	Zugriff mit unzulässiger Speicherausrichtung (unzulässiges Alignment)
SIGFPE	arithmetic exception	Fehler bei arithmetischer Operation, z. B. Division durch 0
SIGILL	illegal instruction	Unzulässiger CPU-Befehl
SIGSEGV	invalid memory reference	Zugriff auf unzulässige Speicheradresse
SIGSYS	invalid system call	Unzulässiger Aufruf einer Systemfunktion, z. B. mit falschem Argument

Das Funktionsmakro **`assert()`** ermöglicht, die getroffenen Annahmen über unzulässige Programmzustände oder unzulässige Funktionsparameter zu überprüfen. Bei einem entdeckten Fehler bricht das Makro **`assert()`** das Programm durch die Funktion **`abort()`** ab, wodurch ein core-Dump erstellt wird.

3 Anleitung zum Untersuchen eines core-Dumps

Bei der Untersuchung eines core-Dumps ist in der Regel folgende Vorgehensweise sinnvoll:

1. Untersuchung des Programmzustands zum Zeitpunkt des Abbruchs:

   In diesem Schritt werden die Funktionen der Aufruf-Hierarchie statisch untersucht um herauszufinden, in welcher Funktion der tatsächliche Programmzustand vom erwarteten abweicht.

   (a) Debugger für die core-Datei starten, bzw. core-Datei in den Debugger laden.
   gdb-Befehl: **`target core core`**
   Der Debugger ist danach in dem selben Zustand, wie wenn das Programm schon beim abgebrochenen Programmlauf mit dem Debugger untersucht und direkt vor dem Abbruch angehalten worden wäre.

   (b) Aufruf-Hierarchie (d. h. "Kette") der aufgerufenen Funktionen und Unterfunktionen zum Zeitpunkt des Abbruchs feststellen:
   gdb-Befehl: **`backtrace`** bzw. **`bt`**
   In welcher Funktion erfolgte der Abbruch?
   Von welcher Funktion wurde die abbrechende Funktion gerade aufgerufen?
   Notieren Sie diese Kette der Funktionsaufrufe bis zum Hauptprogramm.
   Die Informationen dafür liefert der Inhalt des Stack zum Zeitpunkt des Abbruchs.

   (c) Untersuchen des Inhalts der Variablen und Funktionsargumente zum Zeitpunkt des Abbruchs:
   Welche Variable enthält einen Inhalt, der den Abbruch verursacht hat?
   Welche Variable enthält Daten, die nicht plausibel sind, z. B. einen Wert außerhalb des erwarteten Wertebereichs, oder einen im aktuellen Programmzustand unerwarteten Wert?
   Ist das Programm in einem Programmzweig, der bei den aktuellen Randbedingungen eigentlich nicht durchlaufen werden sollte?
   Diese Untersuchung beginnt am besten bei der letzten aufgerufenen "eigenen" Funktion: Wenn der Abbruch z. B. in der Bibliotheksfunktion **`printf()`** erfolgte, beginnt die Untersuchung bei den Variablen und Argumenten der Funktion, die **`printf()`** aufgerufen hat.

Wenn die Untersuchung ergibt, daß ein Funktionsaufruf mit fehlerhaften Argumenten zum Absturz geführt hat, wird die Untersuchung in der aufrufenden Funktion fortgesetzt. Die Aufruf-Hierarchie wird so weit verfolgt, bis die Funktion gefunden ist, in der die (direkte oder indirekte) Fehlerursache zum erstenmal auftritt.

gdb-Befehle zum Bewegen innerhalb der Aufruf-Hierarchie des Stack: **`up`** und **`down`**

Außerdem können Sie im *gdb*-Debugger in eine in der **`backtrace`**-Liste aufgeführte Funktion springen, indem sie den Eintrag dieser Funktion mit der **mittleren** Maustaste anklicken.

Damit steht jetzt fest, *wo* und *wie* das Programm vom erwarteten Verhalten abweicht. Oft ist aber bei dieser statischen Analyse, die der core-Dump ermöglicht, noch nicht erkennbar, *warum* es zu der Abweichung kommt.

2. Rekonstruktion des Ablaufs, der zum fehlerhaften Zustand führte:

   Aus der Untersuchung des core-Dumps sind die Randbedingungen bekannt, bei denen der Fehler zuerst auftritt: Funktionsname, Funktionsargumente und Variableninhalte.

   In diesem Schritt wird der fehlerhafte Programmablauf nachgestellt und mit Hilfe des Debuggers untersucht:

   (a) Setzen Sie einen Breakpoint auf die oben bestimmte Funktion, in der die (direkte oder indirekte) Fehlerursache zum erstenmal auftritt.

   gdb-Befehl: **`break`** *Funktionsname* bzw. **`b`** *Funktionsname*

   (b) Starten Sie das Programm im Debugger mit den Argumenten (bzw. geben Sie bei Eingaben wieder die Daten ein), mit denen das Programm in den fehlerhaften Zustand kommt[1].

   gdb-Befehl: **`run`** bzw. **`run`** *Argument1 Argument2 ...*

   (c) Das Programm bleibt nun bei jedem Aufruf der Funktion, bei der der Breakpoint gesetzt wurde, stehen. Möglicherweise erfolgte der Absturz nicht beim ersten Aufruf dieser Funktion, sondern erst bei einem späteren.

   Wenn aus dem aktuellen Programmzustand sicher erkennbar ist, daß der fehlerhafte Zustand beim aktuellen Aufruf der Funktion nicht eintreten wird, kann das Programm bis zum nächsten Breakpoint fortgesetzt werden.

   gdb-Befehl: **`continue`** bzw. **`c`**

   Wenn nicht offensichtlich ist, daß die Funktion beim aktuellen Aufruf fehlerfrei durchlaufen wird, muß die Funktion Schritt für Schritt abgearbeitet und beobachtet werden, um dabei die Fehlerursache zu finden.

   gdb-Befehle: **`next`** bzw. **`n`**
   oder **`step`** bzw. **`s`**

[1] Je nach den Zusammenhängen innerhalb des Programms kann es zum Nachstellen des Absturzes notwendig sein, das Programm wieder mit genau den Daten in genau der Eingabe-Reihenfolge zu betreiben, wie dies beim abgestürzten Programmlauf der Fall war.
Möglicherweise ergibt aber die vorher durchgeführte statische Analyse des core-Dump, welche Dateneingabe zum Absturz führt. In diesem Fall kann der gewünschte fehlerhafte Programmzustand evtl. direkt durch Eingabe dieser Daten reproduziert werden.
Eine weitere Möglichkeit, den fehlerhaften Programmzustand zu reproduzieren, ist das gezielte Setzen von Variablen-Inhalten auf die Werte, die zum Fehler führen.