

# Kernel-Debugging

Bernhard Walle  
bernhard@bwalle.de

## 1 Kernel-Konsole

Auch im Zeitalter von ausgefeilten Softwarewerkzeugen sind Logausgaben auf die Konsole immer noch die einfachste und schnellste Debuggingstrategie, von der im Linux-Kernel und seinen Gerätetreibern auch ausgiebig Gebrauch gemacht wird.

### 1.1 Logmeldungen ausgeben

Im Kernel steht für Logausgaben die Funktion `printk()` zur Verfügung. Sie funktioniert genauso wie `printf()` im Userspace, mit dem Unterschied, dass der Formatstring mit einer Priorität beginnt. Die folgende Tabelle zeigt die verfügbaren Loglevel:

| Konstante    | Wert  | Bedeutung                                     |
|--------------|-------|---|
| KERN_EMERG   | "<0>" | Unbenutzbares System                          |
| KERN_ALERT   | "<1>" | Benutzerintervention unmittelbar erforderlich |
| KERN_CRIT    | "<2>" | Kritischer Zustand                            |
| KERN_ERR     | "<3>" | Fehler  |
| KERN_WARNING | "<4>" | Warnung                                       |
| KERN_NOTICE  | "<5>" | Wichtige Meldung, aber kein Fehler            |
| KERN_INFO    | "<6>" | Informationsmeldung                           |
| KERN_DEBUG   | "<7>" | Debugmeldung                                  |

Ein `printk()`-Aufruf könnte also folgendermaßen aussehen:

```
printk(KERN_INFO "%d devices found\n", nr_devices);
```

In Gerätetreibern wird man `printk()` vornehmlich nicht direkt verwenden. Stattdessen gibt es die Funktion `dev_printk()`, die als zusätzlichen Parameter einen `struct device`-Zeiger erhält. Dadurch können die Logausgaben einem Gerät, für das der Treiber verantwortlich ist, zugeordnet werden. In beiden Fällen existieren für jeden Loglevel Abkürzungsmakros, zum Beispiel `pr_info()` und `dev_info()`.

Zudem gibt es Zusicherungsmakros (*Assertions*): Während `BUG_ON()` bei nicht erfüllter Bedingung eine Kernelpanic auslöst, wird bei `WARN_ON()` lediglich ein Backtrace ins Kernellog geschrieben und das System läuft unverändert weiter.

## 1.2 Logmeldungen anzeigen

Im Kernel sind die Logmeldungen als Ringpuffer organisiert. Im Userland gibt es zwei Schnittstellen, um auf den Inhalt des Ringpuffers zuzugreifen:

- Die Datei `/proc/kmsg` wird vom `klogd` gelesen, der die Meldungen über den Syslog-Mechanismus an den `syslogd` weitergibt. Dieser sorgt dann seinerseits dafür, dass die Meldungen dauerhaft im Systemlog gespeichert bleiben.
- Zusätzlich kann jederzeit über die Funktion `klogctl()` auf den kompletten Inhalt des Ringpuffers zugegriffen werden. Das Programm `dmesg` macht davon Gebrauch und gibt den aktuellen Inhalt des Kernellogs aus.

Beide Mechanismen helfen allerdings nicht bei Systemabstürzen, da dann im Userspace keine Daten mehr verarbeitet werden können. Hier ist man auf die Kernelkonsole angewiesen. Standardmäßig ist dies die aktuelle Konsole, bei manchen Distributionen werden aber alle Kernelmeldungen in der Vorkonfiguration auf eine dedizierte Konsole umgeleitet. Diese Einstellung kann mit `setlogcons` bzw. `klogconsole` angepasst werden.

Der Loglevel der Kernelkonsole kann mit `sysctl kernel.printk` geändert werden. Der Wert besteht aus vier Zahlen, wovon die erste die gerade aktive Logpriorität widerspiegelt. Mit der Tastenkombination `SysRQ-Loglevel` ist eine Anpassung sogar noch dann möglich, wenn ansonsten keine Kommandos mehr ausgeführt werden können.

Eine serielle Konsole empfiehlt sich während der Kernelentwicklung aus mehreren Gründen: Sie ist zuverlässiger, kann protokolliert werden und man ist nicht auf eine Kamera angewiesen, um Ausgaben etwa per E-Mail zu versenden. Allerdings ist ein zusätzlicher PC erforderlich, der mit einem Nullmodemkabel verbunden wird. Auf diesem läuft ein Terminalprogramm wie `screen` oder `picocom`. Auf dem Zielrechner wird an die Bootkommandozeile `console=ttyS0,115200` angehängt. Soll weiterhin als Systemkonsole der Bildschirm verwendet werden, so ergänzt man diese Kommandozeile durch `console=tty0`.

Falls der Zielrechner keine serielle Schnittstelle besitzt, so gibt es mehrere Alternativen: Bei Arbeitsplatzrechnern dürfte die einfachste Möglichkeit eine zusätzliche PCI-Karte sein. Handelt es sich um einen Server, hat man gute Chancen, dass das System ohnehin schon eine virtuelle serielle Konsole besitzt: Neben dem standardisierten IPMI gibt es auch proprietäre Implementierungen wie *iLO* von HP oder *AMT* von Intel.

Ist auch dies keine Option, dann gibt es noch die Netconsole und als letzte Alternative einen USB-Adapter. Benötigt man die Logausgaben schon früh beim Booten, so sind fest einkompilierte USB- bzw. Netzwerktreiber die Bedingung. Allerdings ist für `earlyprintk` eine „echte“ oder eine emulierte serielle Schnittstelle unerlässlich.

## 2 Kernel-Crashdumps

Ein *Crashdump* bezeichnet den Speicherinhalt des Rechners unmittelbar vor dem Absturz. Dieser stellt den Zustand des Systems dar, wodurch man durch dessen Analyse auf die Fehlerursache schließen kann.

Während die „Debugging mit `printk()`“-Methode gut beim Entwickeln ist, kann man sie nur schlecht zur Fehlersuche beim Kunden einsetzen. Somit ist es wenig verwunderlich, dass die Entwicklung von Crashdumps bei den Enterprise-Distributionen begonnen hat. Während SUSE auf *LKCD* setzte, hatten ältere Red-Hat-Versionen *Netdump* und *Diskdump* integriert. All diese Vorgänger spielen heute keine Rolle mehr, wenngleich sie natürlich die Entwicklung von *Kdump* beeinflusst haben.

## 2.1 Kexec und Kdump

Bei *Kexec* handelt es sich um einen Mechanismus, um Linux aus Linux heraus zu booten; das BIOS wird dabei umgangen. Vorteile sind kürzere Startzeiten und das Vermeiden einer teilweise komplizierten Bootloaderkonfiguration. Problematisch wirken sich fehlerhafte Treiber aus, die von einem frisch initialisierten Gerät ausgehen.

Als einzige Voraussetzung für *Kexec* muss der Kernel mit der Option `CONFIG_KEXEC=y` kompiliert worden sein und man muss die *kexec-tools* installieren.

Die Idee von *Kdump* ist es, *Kexec* so zu erweitern, dass im Falle eines Systemabsturzes ein sog. *Capture-Kernel* gebootet wird. In dieser speziellen Umgebung ist es möglich, den Speicher der kompromittierten Umgebung in Form eines Crashdumps zu sichern. Da das Sichern in einer „sauberen“ Umgebung erfolgt, ist das Verfahren relativ zuverlässig.

Um *Kdump* zu nutzen, müssen im laufenden Kernel die Optionen `CONFIG_KEXEC=y` und `CONFIG_DEBUG_INFO=y` gesetzt sein. Im *Capture-Kernel* wird `CONFIG_CRASH_DUMP=y`, `CONFIG_PROC_VMCORE=y` und `CONFIG_RELOCATABLE=y` benötigt. Man kann diese Optionen auch kombinieren und somit den normalen Kernel auch zum Schreiben des Dumps verwenden. Folgende Schritte sind notwendig, um ein Speicherabbild zu erhalten:

### 1. Speicherreservierung

Der *Panic-Kernel* wird – im Gegensatz zu einem normalen, mit *Kexec* geladenen Kernel – in einen speziellen Bereich geladen, von wo er auch direkt ausgeführt wird. Dieser Bereich beinhaltet auch den Speicher, den man im Userspace zum Sichern des Dumps zur Verfügung hat. Da der alte Bereich nicht überschrieben werden kann bevor er gesichert wurde, braucht man also Speicher, der im normalen Betrieb vor dem Absturz nicht angefasst wird.

Das Reservieren des Speichers erfolgt mit der Option `crashkernel=MENGE` auf der Kernelkommandozeile. Eine Gute Wahl ist 64M bei „normalen“ und 128M bei größeren Systemen.

### 2. Panic-Kernel laden

Das folgende Beispiel zeigt, wie man mit *Kexec* den *Panic-Kernel* lädt:

```
% kexec -p /boot/vmlinuz --initrd=/boot/initrd-kdump --args="\
root=... maxcpus=1 irqpoll elevator=deadline reset_devices"
```

Die zusätzlichen Kerneloptionen erhöhen die Fehlertoleranz des Linux-Kernels, so dass man auch unter ungünstigen Voraussetzungen in dieser speziellen Situation einen Dump erhält.

### 3. Systemcrash

Der Systemcrash kann entweder eine Kernelpanic sein oder mit SysRq-C manuell ausgelöst werden. Um bei jedem Kerneloops eine Panic auszulösen, setzt man `sysctl kernel.panic_on_oops=1`, und um eine Panic bei einem NMI auszulösen verwendet man `sysctl kernel.unknown_nmi_panic=1`.

### 4. Dump sichern

Das Wegkopieren des Dumps erfolgt in der Regel in der Initrd, damit unter bestimmten Umständen gar kein Zugriff auf das Rootdateisystem notwendig ist. Trotzdem hat man hier ein normales Linux-System zur Verfügung, wenn auch mit sehr wenig Speicher. Der alte Speicher, der gesichert werden soll, steht als `/dev/oldmem` zur Verfügung. Als Dump verwendet man aber `/proc/vmcore`, da hier bereits die ELF-Header enthalten sind.

Das Sichern kann durch einfaches Kopieren von `/proc/vmcore` erfolgen. Bei `cp` sollte man die Option `--sparse=always` verwenden. Speziell bei großen Systemen ist es allerdings wünschenswert, den Dump nach speziellen Kriterien zu filtern und seitenweise zu komprimieren. Beides leistet das Programm *makedumpfile*.

Dieser ganze Prozess kann bei den gängigen Linux-Distributionen mehr oder weniger auf Knopfdruck automatisch konfiguriert werden. Leider gibt es keine einheitlichen Konfigurationsmechanismen, so dass ein Blick in die Dokumentation unerlässlich ist.

## 2.2 Dump analysieren

Das Tool der Wahl zum Analysieren von Kerneldumps ist *crash*. Der Start erfolgt mit

```
% crash [Mapfile] Namelist Dumpfile
```

Hierbei bezeichnet die *Namelist* den Kernel im ELF-Format (normalerweise `vmlinux`), der das *Dumpfile* erzeugt hat. Das *Mapfile* (`System.map`) ist optional und wird benötigt, falls zwar Version und Konfiguration, nicht aber das Binary von *Namelist* mit dem eigentlich gecrashten Kernel übereinstimmen.

Bei *crash* selbst handelt es sich um einen GDB-ähnlichen Debugger mit vielen zusätzlichen Kommandos speziell für Kerneldumps. Neben dem „Crash Whitepaper“ [6] ist auch die eingebaute Hilfe lesenswert. Sie kann über `help KOMMANDO` abgerufen werden.

Beispielsweise kann man sich mit `dmesg` den Inhalt der Kernelkonsole vor dem Crash anschauen, mit `bt` einen Backtrace des aktuellen Tasks – das ist beim Start der Prozess, der zum Absturz geführt hat – betrachten. `sys` und `mach` zeigen allgemeine bzw. maschinen-spezifische Systeminformationen an. Mit der SIAL-Erweiterung lassen sich auch eigene Skripte in einer C-ähnlichen Syntax schnell implementieren.

### 3 Interaktive Kerneldebugger

Während interaktive Debugger im Userspace unter Linux praktisch von Beginn an existierten, gab es im Kernel lange keinen offiziellen Debugger. Der Grund war nicht technischer sondern politischer Natur: LINUS TORVALDS lehnte Kerneldebugger ab, da sie seiner Meinung nach zu schlechterem Code führen.

Beim *KDB* handelt es sich um einen von SGI externen gepflegten Kernelpatch. Er ermöglicht das Debuggen auf demselben Rechner, auf dem der zu untersuchende Kernel läuft. Natürlich kann aber auch ein zweiter Rechner mit einer seriellen Konsole verwendet werden. Mit der PAUSE-Taste springt man in den Debugger und hält das System an. Im Gegensatz zum KGDB wird aber kein Debugging auf Quellcodeebene unterstützt.

Zur Verwendung des *KGDB* benötigt man zwei Rechner, die über ein Nullmodemkabel verbunden sind, analog zur seriellen Konsole. Beim Kernel müssen die Optionen `CONFIG_KGDB=y`, `CONFIG_KGDB_SERIAL_CONSOLE=(y|m)`, `CONFIG_FRAME_POINTER=y`, `CONFIG_DEBUG_RODATA=n` (für Breakpoints) und `CONFIG_DEBUG_INFO=y` gesetzt sein.

Falls man `kgdboc` als Modul kompiliert hat, lädt man es mit `modprobe` und setzt die Schnittstellenparameter als Modulparameter, z. B. `kgdboc=ttyS0,115200`. Mit `Sysrq-G` startet man den KGDB. Dies bedeutet, dass das System anhält und man sich auf dem anderen Rechner mit dem GDB verbindet. Beim einkompilierten KGDB fügt man hingegen den `kgdboc`-Parameter an die Bootkommandozeile an. Soll der KGDB gleich beim Systemstart genutzt werden hilft `kgdbwait`. Das manuelle Starten mit `SysRq` entfällt dann.

Auf dem Entwicklungsrechner verbindet man sich dann mit dem GDB:

```
% gdb vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
```

Der GDB kann dann normal verwendet werden. Die Ausführung des Kernels wird mit `continue` fortgesetzt. Breakpoints können mit `bt` gesetzt, Variablen mit `print` ausgelesen und der Code kann Schritt für Schritt mit `next` bzw. `step` ausgeführt werden.

#### Literatur

- [1] KROAH-HARTMAN: *Linux Kernel in a Nutshell*, O'Reilly-Verlag.
- [2] QUADE, KUNST: *Linux-Treiber entwickeln*, dpunkt-Verlag.
- [3] TURNER, KOMARINSK: *Remote Serial Console HOWTO*.
- [4] GOYAL, BIEDERMAN, NELLITHEERTHA: *Kdump, A Kexec-based Kernel Crash Dumping Mechanism*.
- [5] GOYAL, HORMAN, OHMACHI, SONI, GARG: *Kdump: Smarter, Easier, Trustier*.
- [6] ANDERSON: *Red Hat Crash Utility*, <http://people.redhat.com/anderson>.