Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 51

Kern-Technik

Zum Schutz kritischer Abschnitte haben Mutexe im Linux-Kernel Semaphore verdrängt. Hauptgrund ist die bessere Performance. Das patentierte Lock-Debugging dagegen ist ein Abfallprodukt. Eva-Katharina Kunst, Jürgen Quade

Kritische Abschnitte zu schützen gehört zu den Herausforderungen der Programmierung moderner Systeme. Mehrere Methoden, beispielsweise atomare Integer- und Bitvariablen, Semaphore, Spinlocks, Sequence-Locks, Memory-Barrier [1], RT-Mutexe oder auch Futexe [2], waren bereits Gegenstand der Kern-Technik. Nur Mutexe, die im Linux-Kernel seit einiger Zeit die klassischen Semaphore ablösen, blieben außen vor.

Das scheint zunächst nicht tragisch, handelt es sich bei einem Mutex doch ohnehin um eine Sonderform des Semaphors. Bei ihnen dürfen konfigurierbar viele Instanzen gleichzeitig einen kritischen Abschnitt betreten. Das Mutex – begrifflich von Mutual Exclusion (gegenseitiger Ausschluss) abstammend – fixiert die Instanzenanzahl auf eine. Ein kritischer Abschnitt ist eine Codesequenz, in der die CPU auf gemeinsam genutzte Betriebsmittel zugreift. Arbeiten mehrere Instanzen parallel in diesem Code, kommt es zu unvorhersehbaren Ergebnissen. Semaphore und Mutexe verhindern das, indem sie den Eintritt sequenzialisieren.

Die Begrenzung auf eine Instanz ermöglicht im Vergleich zum universellen Semaphor eine effizientere Implementierung sowohl bezüglich der Laufzeit als auch des Speicherverbrauchs [3]. Und da Entwickler Semaphore ohnehin vorwiegend als Mutexe einsetzen, spricht vieles für deren Verwendung.

Entweder – oder

Jedoch gibt es bei den Kernel-Mutexen im Vergleich zum Semaphor weitere Einschränkungen: Während der Kern Semaphore unabhängig vom so genannten Eigentümer einer Instanz hält und andere Instanzen sie freigeben dürfen, ist dies mit den Linux-Mutexen nicht zu machen. Hier gilt das Eigentümerprinzip: Die Instanz, die ein Mutex erfolgreich reserviert hat, gibt es auch frei. Ein Consumer-Producer-System lässt sich damit also nicht aufbauen. Das stellt aber kein Problem dar, denn Semaphore stehen ja ohnehin weiter zur Verfügung.

Erfreulicherweise ist der Übergang vom klassischen Semaphor oder auch Spinlock zum effizienten Mutex schmerzlos. In den meisten Fällen ersetzt ein Systemaufruf einen anderen. Konkret gilt das für die Definition und Initialisierung des Mutex, für das Halten und Freigeben und für das Prüfen, ob eine Instanz ein Lock hält oder nicht.

Mutexe richtig reservieren

Für die Definition und Initialisierung gibt es statische und dynamische Methoden. Den Torwächter spielt in beiden Fällen eine Variable vom Typ »struct mutex«. Die statische Methode definiert die Mutexvariable über das Makro »DEFINE_MUTEX()«. Das Beispielmodul von Listing 1 zeigt demgegenüber mit Zeile 3 in Kombination mit Zeile 9 die dynamische Variante.

Um das Mutex zu reservieren, ruft der Programmierer im einfachsten Fall die

Mutex versus Spinlock

Zwei Methoden regeln den Zugang zu Codestücken, die exklusiven Zugriff auf Ressourcen benötigen: Ein Mutex legt eine Instanz schlafen, während das Spinlock aktives Warten realisiert – das so genannte Spinning.

Bei einem kurzen kritischen Abschnitt ist aktives Warten effizienter, da kein Kontextwechsel notwendig ist. Außerdem lässt es sich auch im Kontext von Hard-IRQs und Soft-IRQs einsetzen. Dort schützt es kritische Abschnitte, die sich beispielsweise eine Treiberfunktion mit einer Interrupt-Service-Routine teilt. Das Mutex spielt seinen Vorteil demgegenüber bei längeren zu schützenden Codesequenzen aus. Durch das Schlafenlegen der Instanz können andere rechenbereite Prozesse die vorhandene CPU-Zeit nutzen. Außerdem ist das Verfahren auch auf Singlecores einsetzbar.

Adaptives Spinning nutzt Multicores

Zudem haben die Kernelprogrammierer dem Linux-Mutex ein so genanntes adaptives Spinning beigebracht und es damit zum Zwitter gemacht. Wird auf einer Mehrkernmaschine ein zu reservierendes Mutex von einer Instanz auf einer anderen CPU gehalten, entscheidet sich der Kernel für das aktive Warten (Spinning) und führt keinen Kontextwechsel durch.

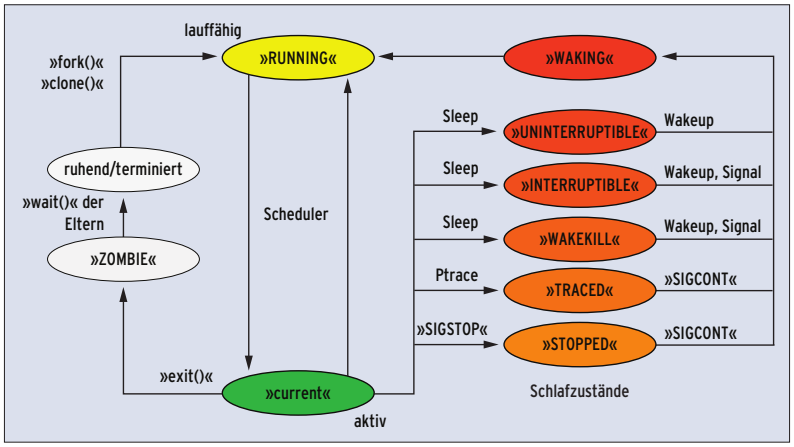


Abbildung 1: Im Zustand »TASK_WAKEKILL« weckt der Kernel seit 2.6.25 schlafende Prozesse auch dann auf, wenn sie das tödliche Signal »SIGKILL« empfangen.

Funktion »mutex_lock()« auf, »mutex_unlock()« gibt es wieder frei:

```
mutex_lock(&mutex_dynamisch);
    /* ... Kritischer Abschnitt */
mutex_unlock(&mutex_dynamisch);
```

Besser reserviert der Entwickler jedoch nach der dynamischen Methode mit den Funktionen »mutex_lock_killable()« oder »mutex_lock_interruptible()«. Diese Funktionen sind zwar komplizierter anzuwenden, weil ihr Rückgabewert auszuwerten ist:

```
if (mutex_lock_killable(&mutex_statisch)
      == -EINTR) {
    return -EIO;
}
    /* ... Kritischer Abschnitt */
mutex_unlock(&mutex_dynamisch);
```

Doch ermöglicht es diese Variante, eine auf das Mutex wartende Instanz per Signal wieder aufzuwecken.

Die Funktionen mit dem Namenszusatz »_killable« sind übrigens vergleichsweise neu im Linux-Kernel, bisher kaum genutzt und wenig bekannt. Das möchten die Kernelhacker unbedingt ändern. Die Funktionen »mutex_lock_interruptible()« und »mutex_lock_killable()« haben gemeinsam, eine schlafende Treiberinstanz auch dann aufzuwecken, wenn sie ein Signal erhält.

Im Schlaf töten

Während die unterbrechbare Variante aber auf jedes Signal reagiert, muss es bei der Killable-Variante schon ein Signal sein, das den Prozess sicher beendet. Das Signal »SIGKILL« bricht den Prozess in jedem Fall ab, bei den übrigen Signalen hängt es aber davon ab, welche Signale die Instanz abfängt. Sie unterbrechen den mit »mutex_lock_killable()« eingeleiteten Schlaf nicht.

Kernel 2.6.25 führte dazu den Prozesszustand »TASK_WAKEKILL« ein, um Applikationsprogrammieren entgegenzukommen (siehe Abbildung 1). Die meisten Anwendungen werten erfahrungsgemäß die Rückgabewerte von Systemaufrufen wie beispielsweise »read()« und »write()« nicht hinreichend aus. Amateurprogrammierer gehen oft fälschlicherweise davon aus, dass Signale die Funktionen nicht abbeziehungsweise unterbrechen. Daraus zogen die Kernelhacker die Konsequenz, die nicht unterbrechbaren Varianten den an sich geschickteren unterbrechbaren vorzuziehen: Lieber ein paar ewig schlafende Jobs in der Prozesstabelle als undefiniert arbeitende Applikationen. Die neuen »_killable«-Funktionen lösen diese Gewissenskonflikte [4].

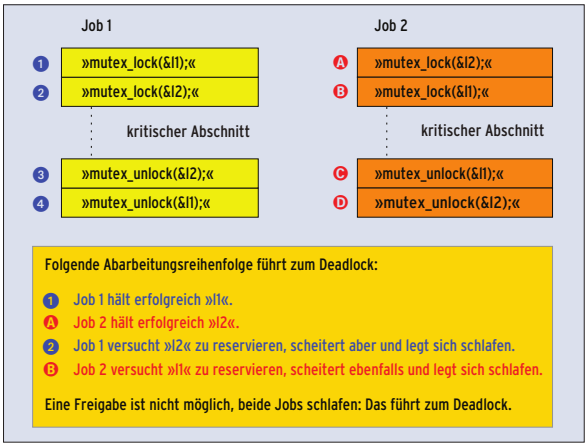
Außerdem gibt es für Mutexe die von Semaphoren und Spinlocks abgeleitete Funktion »mutex_trylock()«. Sie verhindert das Schlafenlegen der aufrufenden Instanz, wenn eine andere Instanz das Mutex bereits hält. In diesem Fall liefert die Funktion den Wert 0 zurück. War die Reservierungsanfrage hingegen erfolgreich, erhält der Aufrufer eine 1 als Rückgabe. Das entspricht dem Verhalten von »spin_lock_trylock()«. Umgekehrte Logik bei »down_trylock()«: Die Funktion gibt in der gleichen Situation 0 zurück. Wer nur den Zustand, ob frei oder gehalten, des Mutex prüfen will, wählt die Funktion »mutex_is_locked()«.

Atomare Variablen

Die Funktion »atomic_dec_and_mutex_lock()« korrespondiert mit der von Spinlocks bekannten Funktion »atomic_dec_and_lock()«. Sie dekrementiert die übergebene atomare Variable und schnappt sich das Lock in einem Rutsch. Falls das Ergebnis 0 ist, sperrt der Kern das Mutex per »mutex_lock()«, reserviert es so und zeigt dies durch den Rückgabewert 1 an. Ist die Variable jedoch ungleich 0, gibt

Listing 1: Bewusst fehlerhaftes Modul »mutextest.c«

```
01 #include <linux/module.h>
02
03 static struct mutex kern_technik;
04
05 static int __init mod_init(void)
06 {
07   /* Mutex-Debugging reaktivieren */
08   debug_locks = 1;
09   mutex_init(&kern_technik);
10   if (mutex_lock_killable(&kern_technik)
11             == -EINTR) {
12     pr_info("Durch Signal unterbrochen!\n");
13     return -EIO;
14   }
15   // FEHLER: Mutexe dürfen nicht zweimal
16   // (rekursiv) reserviert werden
17   if (mutex_lock_interruptible(&kern_technik)
18             == -EINTR) {
19     pr_info("Durch Signal unterbrochen!\n");
20     mutex_unlock(&kern_technik);
21     return -EIO;
22   }
23   // Hier steht eigentlich der Code des per
24   // Mutex geschützten kritischen Abschnitts.
25   mutex_unlock(&kern_technik);
26   return -EIO;
27 }
28
29 static void __exit mod_exit(void)
30 {
31   ;
32 }
33
34 module_init(mod_init);
35 module_exit(mod_exit);
36 MODULE_LICENSE("GPL");
```



◀ Abbildung 2: Im Zusammenspiel mit mehreren Locks kommt es schnell zu Verklemmungen.

die Routine 0 zurück und versucht erst gar nicht das Mutex zu halten.

Trotz der scheinbar einfachen Semantik der Mutex-Funktionen ist ihr Einsatz alles andere als trivial. Ein Fehler wie etwa die falsche Reihenfolge beim Reservieren führt leicht zu einem sporadisch auftretenden Deadlock, wie das Beispiel in Abbildung 2 zeigt. Hinzu kommen noch generell einzuhaltende Randbedingungen beim Einsatz (siehe Kasten „Verwendungsrichtlinien für Mutexe“).

Beinahe alle Einschränkungen kann der Torvalds'sche Betriebssystemkern seit einiger Zeit überprüfen: das Eigentümerprinzip, das Verbot des rekursiven Reservierens, die Gebote, nur über vorgegebene Funktionen zu initialisieren, den Job aufrechtzuerhalten, so lange eine Instanz das Mutex noch hält, und währenddessen seinen Speicher nicht freizugeben sowie den Bann innerhalb von Interruptkontexten.

Ingo Molnar hat dem Kernel mit seinem Lockdep-Validator das dazu notwendige Rüstzeug mitgegeben, das zugleich den korrekten Einsatz von Semaphoren und Spinlocks überwacht. Da die Überwachung jedoch mit Speicherplatz- und Performance-Einbußen verbunden ist, bleibt der Validator standardmäßig deaktiviert.

Bevor ein Entwickler diese Funktionalität aktivieren darf, steht ihm jedoch ein Neubau des Kernels mit der Konfigurationsoption »DEBUG_MUTEXES« ins Haus. Der Kasten „Kernel 2.6.33 unter Ubuntu 9.10 selbst übersetzen“ weist den Weg, da sich die Tools zur Kernelgenerierung zurzeit nicht mit dem Quellcode des aktuellen Kernels vertragen: Am Ende des oft länglichen Kompiliervorgangs beschwert sich ein Werkzeug über eine fehlerhaft erzeugte Headerdatei. Mit einem Patch – wie in Abbildung 3 zu sehen – gelingt das Übersetzen trotzdem. Aber vielleicht ist das bereits Geschichte, wenn der Artikel erscheint.

Zu viele Fehlermeldungen

Ein zweites Patch benötigt der Kernel selbst. Sobald die Überwachungsmimik einen möglichen Fehler in der Logik der Sperren erkennt, reicht er dazu Debug-Informationen an den Syslog-Daemon weiter. Danach schaltet das Subsystem aber jedes weitere Mutex-Debugging ab. Zu leicht würde der Entwickler ansonsten mit einer Kaskade von Fehlermeldungen überhäuft.

Molnars System meldete den platzierten Fehler im Testmodul zunächst nicht: Stattdessen schlug auf dem Testsystem der Autoren bereits beim Booten eines ansonsten unveränderten Kernels die Überwachung mit der Meldung »Possible circular locking dependency detected« im Modul »pulseaudio« an – ein mögliches Problem, das an anderer Stelle zu untersuchen ist.

Es gibt keine Option, ein deaktiviertes Mutex-Debugging wieder einzuschalten. Das Deaktiveren des Mutex-Debuggers steuert die Kernelvariable »debug_locks«, die in der Datei »/usr/src/linux-2.6.33/lib/debug_locks.c« definiert ist. Hat sie den Wert 1, meldet der Kernel ein potenzielles Lockproblem, bei 0 schweigt er. Um das Debug-Subsystem zu kontrollieren, benötigen Entwickler also Zugriff auf die Variable und fügen die Zeile

```
EXPORT_SYMBOL_GPL(debug_locks);
```

hinter deren Definition ein. Alternativ geht das auch am Ende der Quelldatei »debug_locks.c«.

Listing 1 demonstriert nicht nur den Einsatz von Mutexen in eigenem Kernelcode, sondern auch das Mutex-Debugging. Sobald der Entwickler das Modul mit Hilfe des Makefiles aus Listing 2 und »make« übersetzt hat, lädt er es anschließend

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m    := mutextest.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09         $(MAKE) -C $(KDIR) M=$(PWD) modules
10 endif
```

Verwendungsrichtlinien für Mutexe

Die Quellcodedatei »linux/mutex.h« listet eine Reihe von Einschränkungen beim Einsatz eines Mutex auf. Das beginnt bei der Initialisierung, schließt das mehrfache Halten ein und endet bei Vorgaben zum Freigeben der Datenstruktur. So dürfen Programmierer nur über die erwähnten Funktionen beziehungsweise Makros Mutexe initialisieren. Das bedeutet konkret, dass es nicht erlaubt ist, den Speicherbereich eines Mutex-Objekts einfach zu kopieren, um auf diese Weise ein zweites Mutex zu erhalten. Ebenfalls verboten ist die mehrfache Initialisierung eines Linux-Mutex, insbesondere ist dies nicht erlaubt, wenn das Mutex bereits von einer Instanz gehalten wird.

Da ein Mutex die aufrufende Instanz unter Umständen schlafen legt, darf der Programmierer die Funktion nur im Prozess- oder im Kernelkontext aufrufen. Die Verwendung innerhalb einer Interrupt-Service-Routine, eines Soft-IRQ, eines Tasklet oder eines Timers ist also ausgeschlossen. Normale Treiberfunktionen wie »driver_ope n()«, »driver_read()«, »driver_write()« oder »driver_close()« können Mutexe aber problemlos verwenden. Ähnlich steht es um Kernelthreads oder um den Code von Systemcalls. Das Eigentümerprinzip besagt, dass nur die Instanz, die das Mutex reserviert, es auch wieder freigeben darf.

Zu ergänzen ist noch, dass Entwickler Mutexe weder mehrfach reservieren (rekursives Locken) noch mehrfach freigeben dürfen. Und schließlich darf eine Instanz, die ein Mutex hält – beispielsweise ein Kernelthread –, sich nicht beenden, ohne das Mutex vorher freizugeben. Entwickler dürfen den Speicher, in dem sie das Mutex-Objekt ablegen, nicht freigeben, so lange ihr Code das Mutex noch hält.

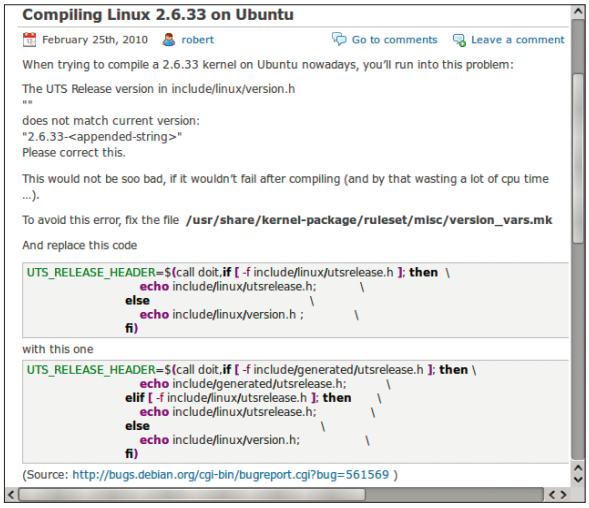


Abbildung 3: Auf einem nicht ganz aktuellen Ubuntu 9.10 müssen Entwickler vor dem Übersetzen noch die Datei »/usr/share/kernel-package/ruleset/misc/version_vars.mk« patchen, damit die Skripte eine Headerdatei korrekt generieren.

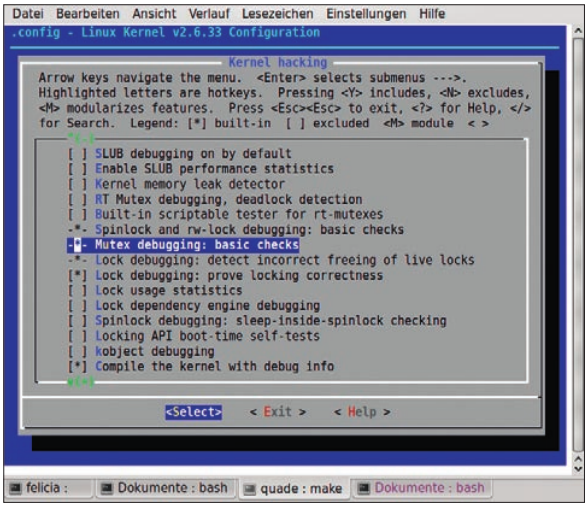


Abbildung 4: Wer im Kernel die Option »DEBUG_MUTEXES« einschaltet, ist in der Lage, eine fehlerhafte Verwendung der Synchronisationselemente aufzuspüren. Die Voreinstellung deaktiviert den Code, da er sich primär zum Debuggen eignet.

mittels »insmod mutextest.ko«. In einem zweiten Terminal lassen sich per »tail -f /var/log/kern.log« die Ausgaben des Syslog-Daemon verfolgen.

Dosierter Regelverstoß

Um das Beispiel in Listing 1 nicht unnötig aufzublasen, reserviert der Code das Mutex direkt in der Funktion »mod_init()«, die der Kernel beim Laden des Moduls per »insmod« aufruft. Um den Mutex-Debugger zu triggern, reserviert das Modul das Mutex fehlerhafterweise zweimal, nämlich in den Zeilen 9 und 17. Das verstößt gegen die Verwendungsrichtlinien. Der durch »debug_locks = 1;« wieder aktivierte Lockdep-Validator erkennt das Problem und gibt es im Syslog aus (Abbildung 5). Er meldet nicht nur die genaue Ursache mit »possible recursive locking detected«, sondern spannenderweise auch den im Programmcode verwendeten Namen des Lock, in diesem Fall »kern_technik«.

Weniger für Mutexe, sehr wohl aber für Spinlocks ist der in geschweiften Klammern stehende Geheimcode interessant, der dem Mutex-Namen in der Ausgabe folgt: »{ + . + ...}«. Eine Hilfe zur Auflösung dieses Rätsels findet sich in der Datei »/usr/src/linux-2.6.33/Documentation/lockdep-design.txt« und in Abbildung 6.

Grundsätzlich registriert der Lock-Validator, in welchem Kontext (Hard-IRQ, Soft-IRQ, Reclaim-FS) ein Kernelthread, eine ISR, ein Tasklet oder ein Timer die jeweilige Sperre hält. Mit Hard-IRQ bezeichnen die Kernelentwickler eine Interrupt-Umgebung, in der Interrupts typischerweise gesperrt sind. Das betrifft hauptsächlich Interrupt-Service-Routinen. Sie sind ihrerseits in der Lage, Code wie Tasklets oder Timer im Soft-IRQ-Kontext zu unterbrechen. Die dritte Umgebung schließlich kennzeichnet, dass sie Zugriffe auf Dateisysteme zulässt.

Der Validator differenziert darüber hinaus zwischen normalen Locks und Lese-Schreib-Locks. Letztere gewähren nur

Kernel 2.6.33 unter Ubuntu 9.10 selbst übersetzen

Der Standardkernel eines Ubuntu-Systems unterstützt das Debuggen von Locks (Mutex, Spinlocks, Semaphore) nicht im Auslieferungszustand. Entwickler müssen sich ihren eigenen Kernel bauen. Das aber ist – wie im Artikel ausgeführt – mit dem noch aktuellen Ubuntu 9.10 und dem aktuellen Kernel 2.6.33 nicht ohne Weiteres möglich. Bis die Distribution ihre Tools anpasst, hilft folgendes Rezept:

- Pakete »kernel-package«, »build-essential« und »libncurses5-dev« zum Generieren von Kernelpaketen installieren.
- Pakete patchen: Abbildung 3 zeigt, wie Entwickler in der Datei »/usr/share/kernel-package/ruleset/misc/version_vars.mk« ein für die Generierung von Kernel 2.6.33 ungeeignetes Codestück gegen die funktionierende Variante austauschen [5].
- Quellcode runterladen: »wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.tar.bz2«.
- Quellcode auspacken: »tar xjvf linux-2.6.33.tar.bz2 -C /usr/src«.
- Kernel konfigurieren:

```
cd /usr/src/linux-2.6.33
cp /boot/config-$(uname -r) ➥
/usr/src/linux-2.6.33/.config
make menuconfig
```

Die Option »DEBUG_MUTEXES« im Menüpunkt »Mutex debugging: basic checks« – wie in Abbildung 4 erkennbar – einschalten.

- Kernel patchen: Für die im Artikel beschriebenen Tests ist es notwendig, die Variable »debug_locks« für den Modulzugriff freizugeben. Dazu fügt der Entwickler die Zeile »EXPORT_SYMBOL_GPL(debug_locks);« ans Ende der Datei »/usr/src/linux-2.6.33/lib/debug_locks.c« an.
- Image- und Headerpakete kompilieren und Reste vorheriger Kernel-Bauten aufräumen:

```
make-kpkg clean
CONCURRENCY_LEVEL=4 ➥
make-kpkg --initrd --rootcmd=fakeroot ➥
--revision=20100228 kernel_image ➥
kernel-headers
```

- Die entstandenen Image- und Headerpakete »linux-image-2.6.33_20100228_amd64.deb« und »linux-headers-2.6.33_20100228_amd64.deb« mit »dpkg« installieren (hier für das 64-Bit-System).
- Den neuen Kernel booten.

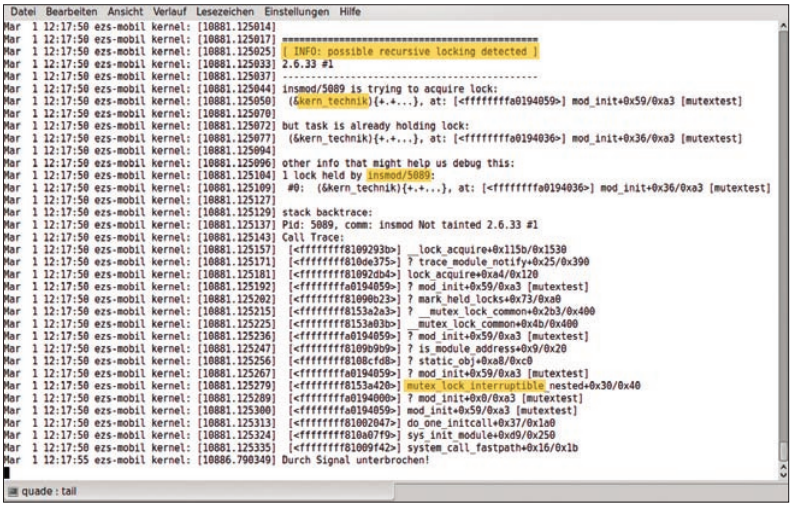


Abbildung 5: Die Ausgabe des Lockdep-Validators zeigt im Klartext den vermeintlichen Fehler.

jenen Prozessen exklusiven Zugriff auf den kritischen Abschnitt, die dort Daten verändern wollen. Die hier erwähnten Mutexe gibt es allerdings nur als normale Locks. Jede der sechs Varianten kann einen von vier Zuständen annehmen. Abbildung 6 verdeutlicht das Prinzip. Im Beispiel halten sowohl ein normaler Hard-IRQ als auch ein normaler Soft-IRQ das Lock »kern_technik« bei freigegebenen Interrupts. Für die Fehlersuche ist das hilfreich: Der Programmierer kann daraus schließen, ob er eine Sperre in einem unerlaubten Kontext reserviert hat oder nicht.

Der Lockdep-Validator organisiert Locks in Klassen, wobei jeweils gleichartige Locks eine Klasse bilden. Gleichartig bedeutet, das Lock sichert eine bestimmte Datenstruktur, beispielsweise »struct inode«. Der Lockdep-Validator registriert, in welchem Kontext und von welcher Instanz ein Lock reserviert wird. Reservierte und nicht freigegebene Locks protokolliert er in durchaus umfangreichen Tabellen inklusive der Zeitstempel mit.

Lockdep findet Deadlocks

Der Lockdep-Validator hängt sich in alle fürs Locking relevanten Systemcalls (zum Beispiel »exit()«) beziehungsweise Kernelfunktionen (etwa »mutex_lock()«) ein. Er hat eine Reihe von Regeln eingebaut, mit denen er das Reservieren und Freigeben von Locks (Spinlocks, Semaphore, Mutexe) während des normalen Betriebs überwacht. Beim Aufruf von »exit()« zum Beispiel sieht der Validator in den Tabellen nach, ob der zugehörige Job noch ein Lock hält. Wenn ja, kommt es zu einer Meldung im Syslog und zum Abschalten des Überwachungscode.

Weitere Regeln identifizieren beispielsweise mehrfaches Reservieren, mehrfaches Freigeben, Reservieren und Freigeben in unerlaubtem Kontext, verschachteltes Halten von Locks oder sogar das Erkennen von Deadlock-Situationen. Der Validator funktioniert sowohl auf Ein- als auch auf Mehrprozessormaschinen. Schade, dass das Subsystem so viel Rechenzeit verschlingt [6].

Der Lock-Validator kann also weitaus mehr als nur rekursiv gehaltene Mutexe melden [7]. Dank seiner ausgeklügelten Mechanismen hat er dabei geholfen, viele Probleme im Kernel zu identifizieren und zu beseitigen. Ende 2008 beantragte sein Entwickler Ingo Molnar beim amerikanischen Patentamt USPTO die Methode zu patentieren [8]. *(mg)* ■

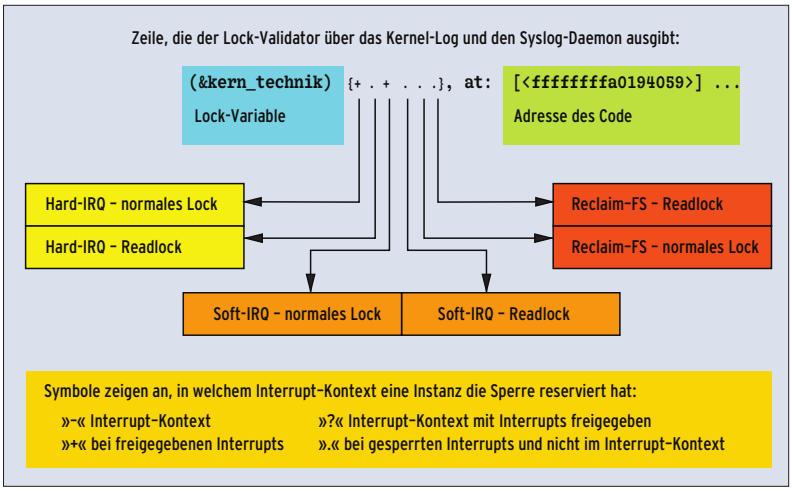


Abbildung 6: Die seltsame Zeichenfolge im Syslog entpuppt sich mit etwas Nachhilfe tatsächlich als sinnvolle Information: Jedes Symbol steht für einen eigenen Lock-Kontext.

Infos

[1] Jürgen Quade, Eva-Katharina Kunst, „Schutz kritischer Abschnitte“: Kern-Technik 5, Linux-Magazin 12/03, S. 82
[2] Jürgen Quade, Eva-Katharina Kunst, „Realtime-Mutexe“: Kern-Technik 44, Linux-Magazin 03/09, S. 88
[3] Ingo Molnar et al., „Generic Mutex Subsystem“, Kerneldokumentation zu Sperren: »Documentation/mutex-design.txt«
[4] Jonathan Corbet, „TASK_KILLABLE“: [http://lwn.net/Articles/288056/]
[5] Nerdy Room, „Compiling Linux 2.6.33 on Ubuntu“: [http://www.nrtm.de/index.php/2010/02/25/compiling-linux-2-6-33-on-ubuntu/]
[6] Ingo Molnar, „Runtime locking correctness validator“: [http://www.mjmwired.net/kernel/Documentation/lockdep-design.txt]
[7] Jonathan Corbet, „The Kernel Lock Validator“: [http://lwn.net/Articles/185666/]
[8] Ingo Molnar, „Method and system for a kernel lock validator“, Patent: [http://www.faqs.org/patents/app/20080294892]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Zurzeit arbeiten Sie an der dritten Auflage ihres Buches „Linux Treiber entwickeln“.