Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 44

Kern-Technik

Manche Applikationen müssen verhindern, dass die CPU bestimmte Codesequenzen mehrfach betritt. Dazu aktivieren sie Realtime-Mutexe im Kernel. Die sind nicht nur besonders effizient, sondern beherrschen auch die für Echtzeitanwendungen wichtige Prioritätsinversion. Eva-Katharina Kunst, Jürgen Quade

Der Schutz kritischer Abschnitte gehört zu den diffizilsten und unangenehmsten Aufgaben eines Programmierers. Er muss diese Codeteile identifizieren (siehe Kasten „Kritischer Abschnitt“) und dazu aus einer Unzahl von Schutzmöglichkeiten die effizienteste Variante auswählen [1]. Sehr verbreitet sind das bekannte Semaphor und das Mutex. Seit Kernel 2.6 hat Linus Torvalds zusätzlich das so genannte Futex in seinen Kernel aufgenommen [2]. „Ein Futex“, so verspricht die Werbebroschüre, „ist ein Fast-User-Mutex, das deshalb so schnell ist, weil es ohne den Kernel auskommt.“

Kernel kontrolliert Eintritt

Das entspricht allerdings nur der halben Wahrheit. Richtig ist, dass ein Futex schnell ist. Unwahr hingegen, dass es ohne den Kernel auskommt. Auch das Futex benötigt Kernelunterstützung: Befindet sich bereits ein Thread im kritischen Abschnitt, legt Linux nachfolgende Threads schlafen beziehungsweise weckt sie wieder auf. Die Kernelentwickler sprechen in diesem Fall von einem Slow-Path, dem langsamen Weg. Im Fast-Path, dem schnellen Weg also, ist der Kernel tatsächlich nicht beteiligt.

Da im Userspace keine Codesequenz wirklich atomar abläuft, beruhen Futexe ausschließlich auf einzelnen Maschinenbefehlen. Einer davon ist »cmpxchg«, der in einem Schritt eine Speicherzelle verändert und gleichzeitig deren Wert überprüft. Die das Futex repräsentierende Speicherstelle »F« kann dabei einen Wert von 0, 1 oder 2 haben (Tabelle 1). Die Operation »futex_lock()« lässt sich damit auf Applikationsebene wie in Abbildung 1 realisieren [3]: Betritt der Kontrollfluss den kritischen Abschnitt, setzt er Futex »F« auf 1. So zeigt er an, dass der Abschnitt ab jetzt nicht mehr frei ist (Abbildung 1, Zeile 2). Je nach dem bisherigen Wert von »F« treten jetzt drei Fälle auf.

War der bisherige Wert 0, ist der kritische Abschnitt bislang frei. Das ist der Fast-Path und bedeutet, dass die Applikation ohne anzuhalten den kritischen Abschnitt betreten darf. Das Futex benötigt dazu keine Hilfe vom Kernel.

Im zweiten Fall ist der ursprüngliche Wert von »F« 1. Ein anderer Prozess belegt also bereits den kritischen Abschnitt, doch warten keine weiteren Rechenprozesse. Daher setzt der Thread das Futex »F« nun auf den Wert 2 (Zeile 4). Das ist das Zeichen dafür, dass ein Rechenprozess wartet. Das Betriebssystem weckt diesen Schläfer wieder auf, sobald der Abschnitt frei ist.

Prozesse treten einzeln ein

Während andere Prozesse die Zeilen 2 bis 4 abarbeiten, kann der Thread, der sich bis jetzt im kritischen Abschnitt befindet, die Funktion »futex_unlock(F)« aufrufen. Damit gibt er das Lock wieder frei. Auf einer SMP-Maschine tritt so ein Fall häufig auf. Er lässt sich am Rückgabewert von 0 in Zeile 4 leicht erkennen. Der Rechenprozess wartet dann doch nicht (er durchläuft dann die While-Schleife ab Zeile 5 nicht), sondern beendet die Funktion »futex_lock()«.

Damit betritt er direkt den kritischen Abschnitt. Allerdings hat »F« jetzt den Wert 2, obwohl in Wirklichkeit doch gar kein Prozess wartet. Die Konsequenz dieser Konstellation: Signalisiert der Prozess durch Aufruf der Funktion »futex_unlock(F)«, dass er den kritischen Abschnitt verlassen hat, durchläuft er unnötigerweise den Slow-Path und setzt einen Systemaufruf zum Aufwecken eines Rechenprozesses ab.

Tabelle 1: Werte einer Futex-Speicherzelle

Wert	Bedeutung
F = 0	Der kritische Abschnitt ist frei
F = 1	Der kritische Abschnitt ist besetzt und es gibt keinen Rechenprozess, der wartet
F = 2	Der kritische Abschnitt ist besetzt, es gibt mindestens einen wartenden Rechenprozess

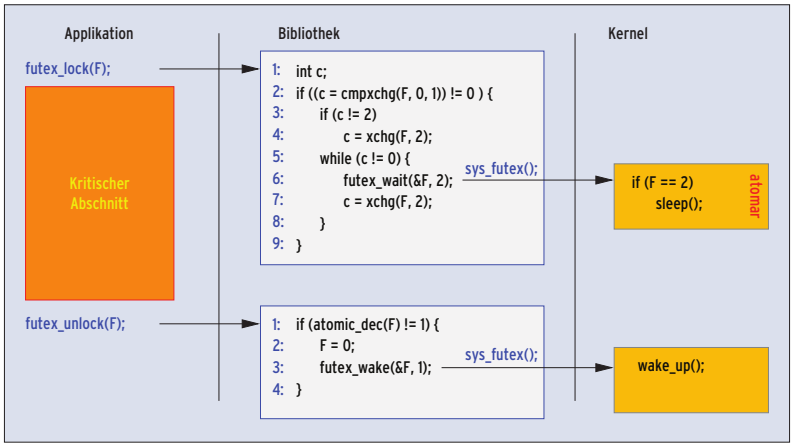


Abbildung 1: Zur Implementierung eines Locks benötigen Entwickler im Userland nur wenige Zeilen Code. Bibliotheken setzen die vom Kernel bereitgestellten Mechanismen um, hier ein Mutex.

Die dritte Alternative bekommt Bedeutung, wenn der ursprüngliche Wert 2 war. Dann war der kritische Abschnitt belegt und es warten bereits weitere Rechenprozesse darauf, ihn betreten zu dürfen. In diesem Fall legt Linux den Prozess direkt mit Hilfe des Systemcalls »sys_futex()« schlafen.

Linux singt das Schlaflied

Passiert dies, kommt der Kernel ins Spiel. Über den Systemcall »sys_futex()« erhält er neben der Adresse von »F« auch noch einen Sollwert: Nur wenn »F« diesen Sollwert besitzt, bettet der Kernel den aufrufenden Prozess wirklich zur Ruhe. Schließlich könnte ja wiederum – beispielsweise auf einem anderen CPU-Kern – der kritische Abschnitt soeben verlassen worden sein. Der Rückgabewert 0 der Funktion »xchg()« in Zeile 4 signalisiert dies, sodass der Thread die While-Schleife ab Zeile 5 nicht durchläuft.

Die Funktion zum Verlassen des kritischen Abschnitts ist glücklicherweise einfacher gestrickt. Sie liest den Wert von »F« aus und dekrementiert ihn gleichzeitig. Dabei sind nur zwei Fälle zu betrachten: War der Wert von »F« vor dem Dekrementieren 2, gibt es Rechenprozesse, die darauf warten, den kritischen Abschnitt betreten zu dürfen. Einen von ihnen weckt Linux mit Hilfe des Systemcalls »sys_futex()« wieder auf. Andernfalls beträgt der Wert vor dem Dekrementieren 1. Da niemand am Eingang des Abschnitts wartet, ist ein Aufwecken also unnötig. Der Fall, dass »F« den Wert 0 hat, tritt bei der Unlock-Operation nicht auf.

Der Kernel hat sich zum Futex »F« eine Liste angelegt, in der er die schlafenden Rechenprozesse verwaltet. Zum Aufwecken wird der erste aus der Liste entfernt und wachgerüttelt. Aber es gibt noch ein Problem: Was passiert, wenn eine Applikation in einem kritischen Abschnitt abstürzt? Anders als bei Filedeskriptoren, die der Kernel verwaltet und im Fall eines Programmabsturzes freigibt, ist das bei den Futexen zunächst nicht möglich. Aus Sicht des Kernels ist ein Futex nichts weiter als eine Speicherstelle.

Um bei Abstürzen Deadlocks zu verhindern, haben Torvalds und seine Mannen das „robuste Futex“ entwickelt [5]. Hinter ihm verbirgt sich der Systemaufruf »sys_set_robust_list()«. Er macht das Futex – genauer eine Reihe von Futexen – vor der Benutzung dem Kernel als solches bekannt. Stürzt ein Programm jetzt ab, prüft der Kernel dessen Liste der verwendeten Futexe. Findet er aktive, informiert er die wartenden Prozesse über den Absturz.

Parameter verwirren Programmierer

Entwickler verwenden also den Systemcall »sys_futex()« dazu, einen Prozess schlafen zu legen oder aufzuwecken. Seine Parameter lassen sich sehr universell einsetzen und sind damit komplex und in der Praxis fehlerträchtig. Kein Wunder also, dass die Standardbibliothek keine Funktion bereithält, die die Systemcalls »sys_futex()« und »sys_set_robust_list()« explizit unterstützt. Allerdings verwenden die unter Linux üblicherweise eingesetzten Synchronisationsfunktionen »pthread_mutex_lock()« und »pthread_mutex_unlock()« intern den Syscall »sys_futex()«. Auf diese Weise setzten Programmierer unbemerkt bereits reichlich Fast-User-Mutexe ein. Die Pthread-Funktionen unterstützen übrigens insbesondere das robuste Futex.

Für Echtzeitprogrammierer wird es allerdings erst richtig spannend, wenn sie

Kritischer Abschnitt

Entwickler bezeichnen Codesequenzen, in denen sie auf gemeinsam genutze Betriebsmittel zugreifen, als kritischen Abschnitt. Das gemeinsam genutzte Betriebsmittel ist zumeist eine Datenstruktur, etwa im Kernel die Liste aller Rechenprozesse. Damit es beim Zugriff auf das Betriebsmittel zu keinen Inkonsistenzen kommt, dürfen keine parallelen Zugriffe stattfinden. Die notwendige Sequenzialisierung realisieren die Programmierer über ein Lock (Semaphor, Mutex, Spinlock). Dazu rufen sie zu Beginn der Codesequenz eine Funktion »lock()« oder »P()« auf und zeigen das Ende des Abschnitts mittels »unlock()« oder »V()« an (Abbildung 2).

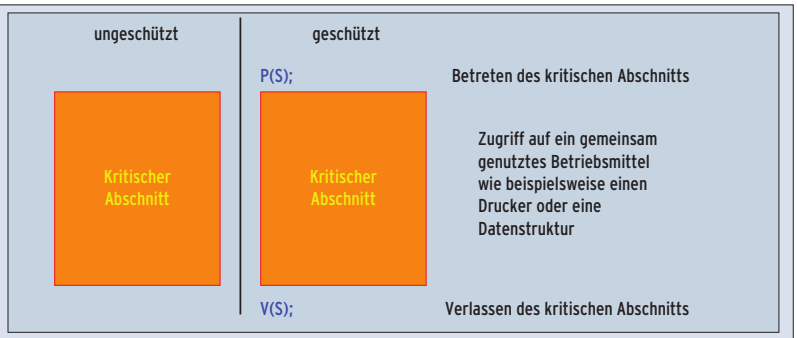
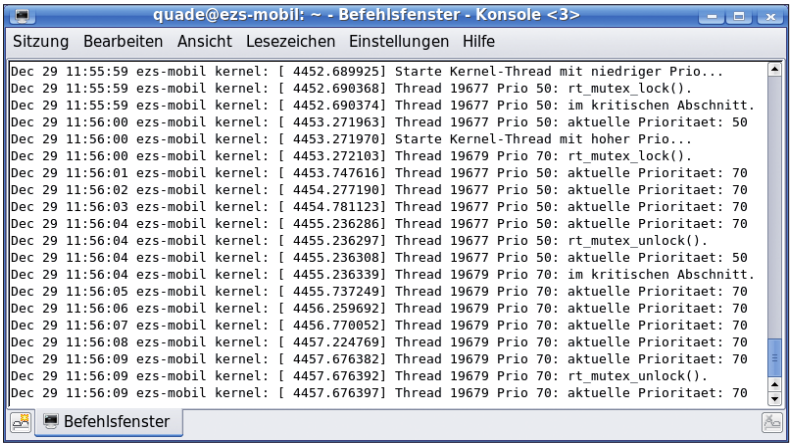


Abbildung 2: Anwender schützen kritische Abschnitte in ihren Programmen, damit diese sich nicht inkonsistent verhalten. Dazu umgürten sie betroffene Codeteile mit den Funktionsaufrufen »P()« und »V()«.



◀ **Abbildung 3: Der hochpriorisierte Thread 19679 vererbt dem niedrig priorisierten Thread 19677 seine Vorrangstellung, weil dieser ein gemeinsam genutztes Lock hält. Nun geht es für beide schneller voran.**

sich im Kernel die für ein Futex verwendete Technik ansehen: Futexe bauen auf den Realtime-Mutexen »rt_mutex« auf [4]. Darüber hinaus kommen sie auch mit den in Echtzeitanwendungen wichtigen Problemen zurecht, die sich durch unterschiedliche Prioritäten ergeben (siehe Kasten „Prioritätsinversion und Prioritätsvererbung"). Zusätzlich interessant: Realtime-Mutexe stehen sämtlichen Kernelfunktionen und Treibern zur Verfügung. Damit komplettieren sie das umfangreiche Arsenal von Schutzmechanismen, das bei Spinlocks beginnt und über Sequence-Locks zu Memory-Barrieren führt [1].

Pünktlich und priorisiert

Die Anwendung der Echtzeit-Mutexe ist einfach: Im Kernel gibt es die Datenstruktur »struct rt_mutex« und die darauf aufbauenden Funktionen »rt_mutex_lock()«, »rt_mutex_lock_interruptible()«, »rt_mutex_timed_lock()« und »rt_mutex_unlock()«. Wie in Listing 1 zu sehen, ruft der Entwickler die Funktion »rt_mutex_lock()« (im Code in der Variante »rt_mutex_lock_interrupti-

Listing 1: Zwei Threads in »prioinv.c« vererben ihre Priorität

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/sched.h>
05 #include <linux/completion.h>
06 #include <linux/kthread.h>
07
08 static wait_queue_head_t wq;
09 static DECLARE_COMPLETION(on_exit);
10 static struct rt_mutex tmut;
11 static struct task_struct *plow, *phigh;
12
13 static int process(void *data) {
15   unsigned long to, i;
16   struct sched_param schedpar;
18
19   schedpar.sched_priority = (long)data;
20   sched_setscheduler(current, SCHED_FIFO, &schedpar);
22   printk("Thread %d Prio %d: rt_mutex_lock().\n",
24          current->pid, current->rt_priority);
26
27   // Begin des kritischen Abschnitts
28   if (rt_mutex_lock_interruptible(&tmut, 0) < 0) {
30     printk("Interrupt waehrend rt_mutex_lock_interruptible\n");
31   } else {
32     printk("Thread %d, Prio %d: im kritischen Abschnitt.\n",
34            current->pid, current->rt_priority);
35     for (i = 0; i < 5; i++) {
36       to = HZ; // eine Sekunde warten
37       to = wait_event_interruptible_timeout(wq, (to == 0), to);
39       printk("Thread %d Prio %d: aktuelle Prioritaet: %d\n",
40              current->pid, current->rt_priority, 99 - current->prio);
43     }
44   }
45   printk("Thread %d Prio %d: rt_mutex_unlock().\n",
47          current->pid, current->rt_priority);
49   rt_mutex_unlock(&tmut);
50   // Ende des kritischen Abschnitts
51
52   printk("Thread %d Prio %d: aktuelle Prioritaet: %d\n",
54          current->pid, current->rt_priority, 99 - current->prio);
56   complete_and_exit(&on_exit, 0);
57 }
58
59 static int __init kthread_init(void) {
61   unsigned long to; // Timeout
62
63   init_waitqueue_head(&wq);
64   rt_mutex_init(&tmut);
65
66   printk("Thread mit niedriger Prio...\n");
67   plow = kthread_create(process, (void *)50, "prio-low");
69   if (IS_ERR(plow)) return -EFAULT;
70   wake_up_process(plow);
71
72   to = HZ; // eine Sekunde warten
73   to = wait_event_interruptible_timeout(wq, (to == 0), to);
74
75   printk("Thread mit hoher Prio ...\n");
76   phigh = kthread_create(process, (void *)70, "prio-high");
78   if (IS_ERR(phigh)) return -EFAULT;
79   wake_up_process(phigh);
81   return 0;
82 }
83
84 static void __exit kthread_exit(void) {
86   printk("kthread_exit()\n");
87   wait_for_completion(&on_exit); wait_for_completion(&on_exit);
89 }
90
91 module_init(kthread_init); module_exit(kthread_exit);
94 MODULE_LICENSE("GPL");
```

ble()«) auf, bevor sein Code den kritischen Abschnitt betritt. Gleiches gilt für »rt_mutex_unlock()«, um das Ende des kritischen Abschnitts zu signalisieren.
Die Funktionen »rt_mutex_lock()« und »rt_mutex_lock_interruptible()« unterscheiden sich nur dadurch, dass sich letztere durch ein Signal unterbrechen lässt, erstere nicht. Die Funktion »rt_mutex_timed_lock()« sorgt dafür, dass ihr Thread nicht unendlich lange auf die Freigabe des kritischen Abschnitts wartet.

Prozess rechts überholt

Wer Listing 1 kompiliert, erhält Einblick in die Prioritätsvererbung. Lädt er das Modul, erzeugt es zwei Threads. Einer erhält eine niedrige Priorität von 50, der andere eine hohe von 70. Das Beispiel räumt dem Thread mit niedriger Priorität genug Zeit ein, um den über das Realtime-Mutex geschützten kritischen Abschnitt zu betreten. Danach aktiviert es den zweiten Thread, der ebenfalls »rt_mutex_lock()« aufruft. Der bevorzugte Prozess wartet auf das RT-Mutex des niedriger eingestuften. Daher vererbt der Kernel die hohe Priorität auf den nachrangigen Prozess (Abbildung 3).
Die aktuelle Priorität des Thread speichert Linux in dem Element »prio« des Prozess-Kontrollblocks »struct task_struct«. Da für den Kernel – entgegengesetzt der Definition am Programmier-Interface – jedoch 0 eine hohe und 99 eine niedrige Echtzeitpriorität bedeutet, rechnen die Zeilen 35 und 43 die Werte wieder in Posix-Darstellung zurück.
Die Kernelentwickler haben Prioritätsvererbung durchaus kontrovers diskutiert: So schrieb Linus Torvalds noch im Dezember 2005: „Friends don't let friends use priority inheritance." Dass er die Prioritätsvererbung letztlich doch in den Kernel eingebaut hat, zeigt, dass er durchaus in der Lage ist, seine Meinung zu revidieren. *(mg)*

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Linux-Treiber entwickeln": Dpunkt-Verlag, 2. Auflage, Juni 2006
[2] Willi Nüßer, „Kampf um die Ressourcen": Linux-Magazin 10/05, S. 84; [http://www.linux-magazin.de/heft_abo/ausgaben/2005/10/kampf_um_die_ressourcen]
[3] Ulrich Drepper, „Tricky Futexes": [http://people.redhat.com/drepper/futex.pdf]
[4] E.-K. Kunst, J. Quade: „Realtime-Preemption", Linux-Magazin 07/07, S. 102
[5] Ingo Molnar, „Robust Futexes", Kerneldokumentation: [http://lxr.linux.no/linux/Documentation/robust-futexes.txt]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Prioritätsinversion und Prioritätsvererbung

Wenn ein Job mit mittlerer Priorität einen Job höherer Priorität aufhält, nennt man das „Prioritätsinversion" (Abbildung 4). Typischerweise sind daran mindestens drei Jobs beteiligt: Einer mit hoher (A), einer mit mittlerer (B) und eben einer mit niedriger Priorität (C). Außerdem teilen sich Job A und C ein Betriebsmittel. Sie haben also einen kritischen Abschnitt gemeinsam, den ein Lock (Semaphor, Mutex, Spinlock oder Futex) schützt.
Zu einer Prioritätsinversion kommt es, wenn C den kritischen Abschnitt betreten hat, in den A eintreten will. In diesem Fall blockiert A so lange, bis C die Unlock-Funktion aufruft. Wird in dieser Situation Job B lauffähig, verzögert er die Weiterverarbeitung von C, denn der hat ja niedrigere Priorität. Indirekt bremst er so aber auch A, obwohl die Verarbeitung von A aufgrund der Priorität dringlicher wäre.
Zur Entschärfung der Prioritätsinversion setzen Entwickler auf die so genannte Prioritätsvererbung. Sobald der hochpriorisierte Job A die Lock-Funktion beim Betreten des kritischen Abschnitts aufruft und das Semaphor oder das Mutex durch den niedrig priorisierten Rechenprozess C belegt ist, erbt C die hohe Priorität von A (Abbildung 5).
So ausgestattet lässt C sich nicht mehr durch den mittelpriorisierte Job B verdrängen. Gibt er den Lock frei, nimmt er wieder seine ursprüngliche Priorität an und der hochpriorisierte Job betritt – mit signifikant kürzerer Latenzzeit – den kritischen Abschnitt.

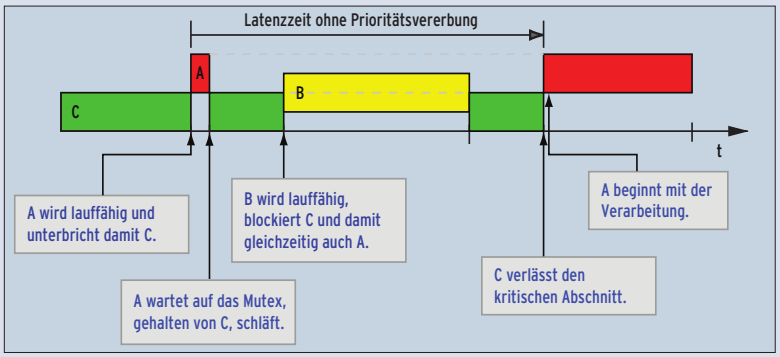


Abbildung 4: Eine Prioritätsinversion mit drei Jobs – A (rot, hohe Priorität), B (gelb, mittlere Priorität) und C (grün, niedrige Priorität) – entsteht, wenn der hochrangige Job A auf zwei nachrangige Jobs wartet.

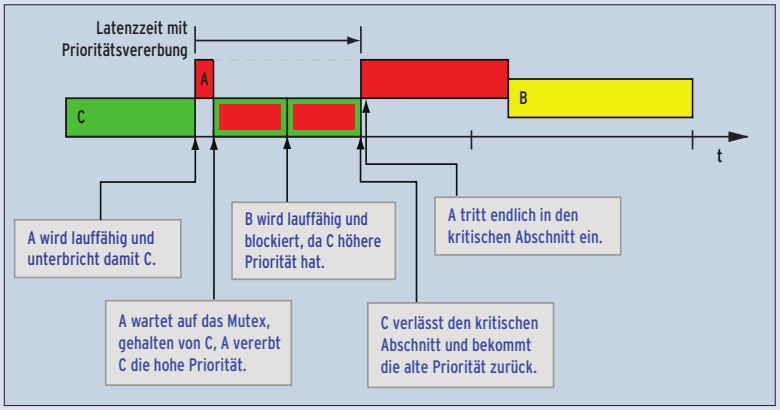


Abbildung 5: Prioritätsvererbung löst die Situation: Der langsame Job C erbt die hohe Priorität von A, solange dieser auf Einlass wartet. Job B wartet. Die Latenz von A ist insgesamt deutlich geringer.