6.4. Kernel-Threads

Anders als Tasklets und Timer können und müssen sich Kernel-Threads schlafen legen. Ein Kernel-Thread entspricht einem Thread auf User-Ebene mit dem Unterschied, dass er komplett im Betriebssystemkern abgearbeitet wird. Daher benötigt er weder Code- noch Datensegmente im User-Space. Ansonsten aber ist er wie jeder andere Rechenprozess im System durch eine Task-Struktur repräsentiert und erscheint beim Aufruf des Kommandos »*ps awxu*« in der Prozesstabelle. Der Abarbeitungszeitpunkt des Kernel-Threads wird folglich durch den Scheduler festgelegt.

Während Applikationen im User-Bereich den Systemcall *fork* zum Erzeugen eines neuen Rechenprozesses nutzen, verwendet der im Kernel die Funktion *kernel_thread*. *fork* hilft im Kernel selbst nicht weiter, da nach Beendigung des Systemcalls die Bearbeitung im User-Kontext fortgesetzt wird (anhand des Rückgabewertes wird dann im Code der Applikation der Elternprozess vom Kindprozess unterschieden). Bei der Erzeugung eines Kernel-Threads wird deshalb nur eine Kopie des gerade aktiven Prozesses angefertigt (wie bei *fork*). Die Funktion *kernel_thread* modifiziert dabei die Prozessorregister des neuen Prozesses derart, dass die übergebene Funktion beim Scheduling des neuen Kernel-Threads abgearbeitet wird.

Die Funktion *kernel_thread* bekommt drei Parameter übergeben:

1. »function_ptr«: Die Adresse der Funktion, die bei erstmaliger Aktivierung durch den Scheduler aufgerufen werden soll.

2. »arg«: Ein Argument für die aufzurufende Funktion.

3. »flags«: Ein Bitfeld, welches die Erzeugung des neuen Rechenprozesses steuert.

Der erzeugte Kernel-Thread erbt hierbei zunächst alle Ressourcen und Eigenschaften vom gerade aktiven Prozess, seinem Elternprozess. Wird bei einem Modultreiber beispielsweise das Erzeugen des Threads in der Funktion `init_module` aufgerufen, erbt der Kernel-Thread die Ressourcen der Applikation, die *insmod* aufgerufen hat. Welche das tatsächlich sind, lässt sich über den Aufrufparameter »Flags« steuern. Dieser Parameter wird aus den folgenden Bits zusammengesetzt:

- CLONE_FS

  Filesysteminformationen werden nicht kopiert, sondern gemeinsam genutzt. Zu den Filesysteminformationen gehören beispielsweise das Rootfilesystem, das »current working directory« und die *umask*.

- CLONE_FILES

  Der neue Thread verwendet die gleichen Filedeskriptoren wie der Elternprozess. Es wird keine Kopie dieser Information angelegt.

- CLONE_SIGHAND

Mit dieser Option wird eingestellt, dass der neue Thread die gleichen Signal-Handler wie der Elternprozess verwendet.

Da ein Kernel-Thread weder geöffnete Dateien noch Signal-Handler verwenden kann, wird im Regelfall in das Bitfeld »CLONE_KERNEL« eingetragen. Es ist in `<linux/sched.h>` als Kombination der drei angegebenen Bits definiert.

```
thread_id=kernel_thread(thread_function, NULL, CLONE_KERNEL );
```

Der Rückgabewert der Funktion ist die Prozess-Identifikation (PID) des neuen Rechenprozesses. Sie wird später benötigt, um den Kernel-Thread wieder zu entfernen. Ein Rückgabewert von »0« bedeutet, dass die Erzeugung schief gegangen ist.

Ein neu erzeugter Kernel-Thread muss noch einige Initialisierungen durchführen. Zunächst gibt der Thread über die Funktion *daemonize* alle Ressourcen im User-Bereich, die er bei der Erzeugung standardmäßig bekommen hat, frei. Damit besitzt der Prozess keine residenten Pages mehr.

Jeder Kernel-Thread ist im System durch einen Namen gekennzeichnet. Dieser Name wird in der Funktion *daemonize* gesetzt. Dazu muss der Funktion – wie bei `sprintf` – ein Formatstring mit zugehörigen Parametern übergeben werden. *daemonize* sorgt dafür, dass der Name des Kernel-Threads nicht mehr als die maximal zulässigen 16 Zeichen umfasst.

Da Kernel-Threads im Kernel-Level ablaufen, bekommen sie die CPU so lange zugeteilt, bis sie sich schlafen legen (*wait_event* oder *wait_event_interruptible*) oder beenden. Ein Kernel-Thread ist beendet, wenn die beim Erzeugen des Threads angegebene Funktion per `return` verlassen wird. Ein Kernel-Thread, der sich weder schlafen legte noch sich rechtzeitig beendete, nähme den anderen Rechenprozessen, insbesondere den Benutzerapplikationen, alle Rechenzeit weg.

Um einen schlafenden Kernel-Thread wieder zu aktivieren, werden die gleichen Mechanismen wie bei den Applikationen verwendet, nämlich zum einen das Aufwecken über *wake_up_interruptible* und zum anderen das Unterbrechen des Wartens durch das Senden eines Signals.

Ein schlafender Kernel-Thread kann aber nur dann durch ein Signal aktiviert werden, wenn der Thread »unterbrechbar« wartet (Taskzustand `TASK_INTERRUPTIBLE`) und das entsprechende Signal nicht blockiert ist.

In Linux-Kernel 2.6 sind 64 Signale definiert, wovon gegenwärtig aber nur etwa 32 genutzt werden. Ein Prozess und damit insbesondere auch ein Kernel-Thread kann jedes einzelne dieser Signale akzeptieren oder blocken.

Implementierungstechnisch sind die 64 Signale in zwei 32-Bit-Variablen repräsentiert, so dass jedes Signal durch ein Bit in einer der beiden 32-Bit-Variablen repräsentiert werden kann. Die Zuordnung zwischen Bits und Position im Bitfeld (entspricht der Nummer des Signals) befindet sich in der Datei `<asm/signal.h>`. Eine »1« an einer Bitposition bedeutet, dass das zugehörige Signal geblockt ist und ankommende Signale verworfen werden. Zur Freigabe eines spezifischen Signals kann der Treiberentwickler die Funktion *allow_signal* verwenden. Diese bekommt als Parameter die Nummer des Signals beziehungsweise die symbolische Repräsentierung des Signals übergeben.

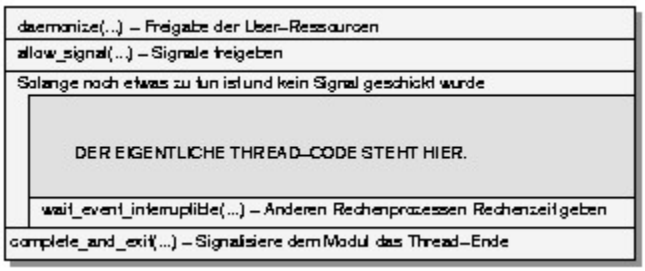
Das Senden eines Signals innerhalb des Kernels wird über die Funktion *kill_proc* realisiert. Dazu muss die PID des Kernel-Threads bekannt sein. Innerhalb des Kernel-Threads selbst kann auf diese PID mittels `current->pid` zugegriffen werden. Ansonsten wird die PID des neu erzeugten Kernel-Threads als Rückgabewert des Aufrufs der Funktion *kernel_thread* zurückgegeben.

Innerhalb des Kernels, also einer Treiberfunktion respektive dem Kernel-Thread, wird mit Hilfe der Funktion *signal_pending* festgestellt, ob ein Systemaufruf (z.B. *wait_event*) ordnungsgemäß oder aufgrund des Empfangs eines Signals beendet wurde:

```
if( !signal_pending(current) ) {
    ... // ein Signal liegt vor
} else {
    ... // kein Signal
}
```

Spätestens wenn das zugehörige Treibermodul entladen wird, muss sich der Kernel-Thread beenden. Im Regelfall wird ihm dazu ein Signal geschickt, beispielsweise über die Shell mit dem Kommando *kill* oder innerhalb des Kernels durch Aufruf der Funktion *kill_proc*. Ob über die Shell oder innerhalb des Kernels: Der Thread wird dabei über seine PID ausgewählt. Doch noch einmal aufgepasst: Signale können durch den Aufruf von Funktionen, beispielsweise durch Aufruf der Funktion *daemonize*, blockiert sein! Eine der ersten Maßnahmen eines Threads (nach Aufruf der Funktion *daemonize*) ist die Freigabe der Signals mittels *allow_signal*.

Abbildung 6-5. Typischer Aufbau eines Kernel-Threads



Es droht die Gefahr eines ungeschützten kritischen Abschnitts. Wie bei den übrigen Methoden auch muss sichergestellt sein, dass sich der Kernel-Thread beendet hat, bevor das Modul entladen wird. Die Aufforderung sich zu beenden, ist über *kill_proc* einfach realisiert. Da sich jeder Rechenprozess selbst beendet, muss jetzt noch auf das wirkliche Ende gewartet werden. Der Schutz dieses kritischen Abschnittes ist alles andere als trivial, insbesondere wenn man bedenkt, dass bei einem Mehrprozessorsystem der Code zum Entladen des Moduls auf einem anderen Prozessor bearbeitet wird als der Kernel-Thread. Dieser Schutz kann über ein Completion-Objekt realisiert werden (siehe Kapitel *Bis zum »Ende«*).

Das Struktogramm in Abbildung *Typischer Aufbau eines Kernel-Threads* zeigt, welche Elemente ein typischer Kernel-Thread standardmäßig benötigt. In Beispiel *Ein einfacher Kernel-Thread* ist ein vollständiges, funktionstüchtiges Modul gezeigt, welches einen Kernel-Thread startet. Dieser Kernel-Thread macht in einer Schleife eine Ausgabe und legt sich dann schlafen. Nach spätestens zehn Durchläufen wird die Schleife abgebrochen und der Thread beendet sich.

Beispiel 6-9. Ein einfacher Kernel-Thread

```
#include <linux/module.h>
#include <linux/version.h>
```

```
#include <linux/init.h>
#include <linux/completion.h>

MODULE_LICENSE("GPL");

static int thread_id=0;
static wait_queue_head_t wq;
static DECLARE_COMPLETION( on_exit );

static int thread_code( void *data )
{
    unsigned long timeout;
    int i;

    daemonize("MyKThread");
    allow_signal( SIGTERM );
    for( i=0; i<10; i++ ) {
        timeout=HZ; // wait 1 second
        timeout=wait_event_interruptible_timeout(
            wq, (timeout==0), timeout);
        printk("thread_code: woke up ...\n");
        if( timeout==-ERESTARTSYS ) {
            printk("got signal, break\n");
            break;
        }
    }
    thread_id = 0;
    complete_and_exit( &on_exit, 0 );
}

static int __init kthread_init(void)
{
    init_waitqueue_head(&wq);
    thread_id=kernel_thread(thread_code, NULL, CLONE_KERNEL );
    if( thread_id==0 )
        return -EIO;
    return 0;
}

static void __exit kthread_exit(void)
{
    if( thread_id ) kill_proc( thread_id, SIGTERM, 1 );
    wait_for_completion( &on_exit );
}

module_init( kthread_init );
module_exit( kthread_exit );
```

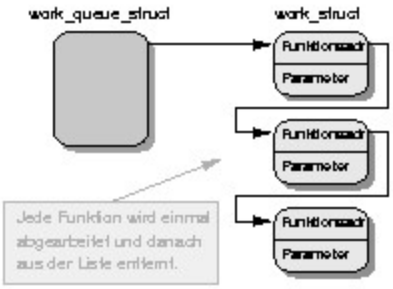
6.4.1. Workqueues

Anhand von Workqueues kann der Treiberentwickler auf einfache Weise einzelne Funktionen abarbeiten lassen. Prinzipiell konkurrieren Workqueues damit mit den Tasklets, allerdings mit dem Unterschied, dass Tasklets vergleichsweise hochprior (direkt nach einem Interrupt) behandelt werden, während Funktionen innerhalb einer Workqueue ähnlich normalen Applikationen verarbeitet werden. Im Gegensatz zu Tasklets ist ihre Abarbeitung unkritisch. Daher ist – wenn möglich – eine Workqueue einem Tasklet immer vorzuziehen.

Eine Workqueue ist nichts weiter als ein Kernel-Thread, der sequentiell Funktionen abarbeitet, deren Adressen zuvor in eine zur Workqueue gehörigen Liste eingehängt wurden. Nur wenn in dieser Liste mindestens eine Funktion eingehängt ist, wird die Workqueue aktiv; ansonsten schläft der zugehörige Kernel-Thread.

Da Workqueues im Prozess-Kontext abgearbeitet werden, könnten innerhalb der Workqueue-Funktionen Warteaufrufe verwendet werden. Das erweist sich in den meisten Fälllen allerdings als wenig sinnvoll: Legt sich nämlich eine der in der Workqueue eingehängten Funktionen schlafen, würde das zum Schlafenlegen des gesamten Workqueue-Threads führen und zugleich alle übrigen, eingehängten Funktionen verzögern.

Abbildung 6-6. Datenstrukturen der Workqueues



Zur Verwendung von Workqueues werden zwei Datenstrukturen benötigt (siehe Abbildung *Datenstrukturen der Workqueues*):

1. `struct work_queue_struct`

   Eine Datenstruktur, die die Workqueue selbst, also einen Kernel-Thread, repräsentiert.

2. `struct work_struct`

   Jeweils eine Datenstruktur pro Funktion, die im Rahmen der Workqueue abgearbeitet werden soll.

Das Erzeugen des Workqueue-Objektes erfolgt durch Aufruf der Funktion *create_workqueue*. Als einziger Parameter ist der Name der Workqueue respektive des Kernel-Threads zu übergeben. Dieser Name darf nicht mehr als 10 Zeichen umfassen! Mit Aufruf dieser Funktion erzeugt der Kernel den zugehörigen Kernel-Thread, der ab sofort auf Funktionen wartet, die er zur Bearbeitung aufrufen kann.

Ein Objekt, das jeweils eine abzuarbeitende Funktion repräsentiert (`struct work_struct`), wird statisch

```
static DECLARE_WORK( work_object, work_queue_function, NULL );
```

oder dynamisch (also zur Laufzeit) definiert:

```
static struct work_struct work_object;
...
static int __init drv_init(void)
{
    ...
    INIT_WORK( &work_object, work_queue_function, NULL);
```

Das so erzeugte Objekt erhält den Namen, der im ersten Parameter angegeben ist – hier also »work_object«. Der zweite Parameter (»work_queue_function«) beinhaltet die Adresse der Funktion, die bei Abarbeitung der Workqueue genau einmal aufgerufen wird. Als dritter Parameter kann schließlich noch ein Datum übergeben werden, das der Funktion »work_queue_function« beim Aufruf übergeben werden würde. Diese Technik macht es möglich, eine Funktion mehrfach zu verwenden.

Die Funktion *queue_work* übernimmt das Einhängen des Work-Objektes in die Workqueue (siehe Beispiel *Einhängen einer Funktion in die Workqueue*). Der erste Parameter spezifiziert den Kernel-Thread – sprich die Workqueue – in dessen Kontext die Funktion abgearbeitet werden soll. Eine so eingehängte Funktion wird bei nächster Gelegenheit vom Kernel-Thread aufgerufen. Es ist durchaus möglich, beliebig viele Funktionen in einer Workqueue unterzubringen. Jede von ihnen wird genau einmal aufgerufen und anschließend wieder aus der Workqueue entfernt. Achtung: Es ist zu vermeiden, dass sich ein Work-Objekt selbst in die Workqueue einhängt. Workqueues sind so implementiert, dass sich selbst einhängende Funktionen gleich wieder aufrufen. Als Folge wäre das System nur noch mit der Abarbeitung dieser einen Funktion beschäftigt.

Beispiel 6-10. Einhängen einer Funktion in die Workqueue

```
if( queue_work( wq, &work_object ) )
    ... // Objekt wird demnächst abgearbeitet.
else
    ... // Objekt ist bereits eingehängt gewesen.
```

Am Rückgabewert der Funktion *queue_work* ist erkenntlich, ob die Funktion eingehängt werden konnte oder nicht. Ist es nicht möglich gewesen, liegt der Grund zumeist darin, dass sich dieselbe Funktion bereits in der Workqueue zur Abarbeitung befindet. Denn ein Work-Objekt kann nur genau einmal eingehängt werden; das gilt insbesondere auch, wenn ein Work-Objekt in unterschiedliche Workqueues eingehängt werden soll.

Es ist auch möglich, eine Funktion erst zu einem späteren Zeitpunkt, also zeitversetzt, abarbeiten zu lassen. Dazu steht dem Treiberentwickler die Funktion *queue_delayed_work* zur Verfügung. Sie enthält als zusätzlichen Parameter einen Timeout-Wert in Jiffies:

```
...
    if( queue_delayed_work( wq, &work, 10*HZ ) ) { // Start nach 10 Sek.
...
```

Um eine Workqueue wieder zu entfernen, existieren die Funktionen *flush_workqueue* und *destroy_workqueue*.

flush_workqueue erzwingt die Abarbeitung der in der Workqueue eingehängten Funktionen. Ist eine Funktion dabei, die zeitversetzt gestartet wird, wartet *flush_workqueue*, bis die Funktion abgearbeitet wurde. *destroy_workqueue* ruft selbst *flush_workqueue* auf und entfernt anschließend die Workqueue komplett.

Vorsicht ist geboten, wenn eine Instanz des Treibers die Workqueue immer wieder mit Funktionen (insbesondere wenn diese zeitversetzt sind) versorgt. In diesem Falle wird die Funktion *flush_workqueue* nicht beendet und wartet ewig. Ein funktionstüchtiges Codebeispiel findet sich in *Codebeispiel für das Anlegen einer Workqueue*.

Beispiel 6-11. Codebeispiel für das Anlegen einer Workqueue

```
#include <linux/version.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>

// Metainformation
MODULE_LICENSE("GPL");

static struct workqueue_struct *wq;

static void work_queue_func( void *data )                    ❶
{
    pr_debug( "work_queue_func...\n" );
    return;
}

static DECLARE_WORK( work_obj, work_queue_func, NULL );      ❷

static int __init drv_init(void)
{
    wq = create_workqueue( "DrvrSmpl" );                     ❸
    if( queue_work( wq, &work_obj ) ) {                      ❹
        pr_debug( "queue_work successful ...\n");
    } else {
        pr_debug( "queue_work not successful ...\n");
    }
    return 0;
}

static void __exit drv_exit(void)
{
    pr_debug("drv_exit called\n");
    if( wq ) {
        destroy_workqueue( wq );                             ❺
        pr_debug("workqueue destroyed\n");
    }
}

module_init( drv_init );
module_exit( drv_exit );
```

❶ Diese Funktion wird abgearbeitet, sobald der Workqueue-Kernel-Thread das nächste Mal gescheduled wird. Die Funktion sollte sich nicht schlafen legen. Zudem können Signale nicht verwendet werden – sie werden vom zugehörigen Kernel-Thread geblockt.

❷ In eine Workqueue können mehrere Funktionen eingehängt werden. Jede Funktion wird über eine Datenstruktur (`struct work_struct`) repräsentiert. Diese Datenstruktur enthält die Adresse der abzuarbeitenden Funktion und einen Pointer auf die der Funktion zu übergebenen Daten. Zur Initialisierung dieser Datenstruktur stehen die Makros *INIT_WORK* und *DECLARE_WORK* zur Verfügung. Ersteres ist für die Initialisierung zur Laufzeit (also innerhalb einer Funktion) vorgesehen, Letzteres zur Initialisierung durch den Compiler (statische Initialisierung).

❸

Diese Funktion erzeugt eine Workqueue mit dem Namen »DrvrSmpl« auf Basis von Kernel-Threads. Der Name der Workqueue darf nicht mehr als 10 Zeichen beinhalten. Über den hier angegebenen Namen lässt sich der Kernel-Thread in der Prozesstabelle identifizieren.

❹ Die in der Datenstruktur »work« angegebene Funktion wird zur Abarbeitung innerhalb der Workqueue »wq« eingehängt. Das ist in den seltensten Fällen innerhalb der Treiberinitialisierungsfunktion durchzuführen, sondern eher innerhalb einer Interrupt-Service-Routine.

❺ Bevor das Modul entladen wird, muss die Workqueue wieder aus dem System entfernt werden. *destroy_workqueue* wartet, bis alle in der Workqueue eingehängten Funktionen abgearbeitet wurden. Der Entwickler muss daher vor Aufruf der Funktion dafür sorgen, dass keine seiner Treiberfunktionen die Workqueue endlos bedient, indem sie immer neue Funktionen in die Queue einhängt.

Die Workqueue-Funktionen stehen nur Treibern zur Verfügung, die einer GNU-Public License unterliegen!

6.4.2. Event-Workqueue

Der Kernel legt standardmäßig für jede CPU eine so genannte Event-Workqueue an (genau eine für ein Einprozessorsystem). Sie ist an die Stelle des aus früheren Versionen bekannten `keventd` getreten und hat für Treiberentwickler den Vorteil, dass sämtliche Initialisierungsarbeiten entfallen. Stattdessen lassen sich mit Hilfe von *schedule_work* Funktionen direkt zur Abarbeitung an den Kernel übergeben. *schedule_work* erhält als einzigen Parameter die Adresse des Work-Objektes. Mit der Funktion *schedule_delayed_work* kann der Treiberentwickler hingegen Funktionen einhängen, die erst zu einem späteren (angegebenen) Zeitpunkt abgearbeitet werden.

Die Aufräumarbeiten übernimmt die Funktion *flush_scheduled_work*. Sie stößt die Abarbeitung der Event-Workqueue an und wartet auf deren Ende (siehe Beispiel *Codebeispiel zur Verwendung der Event-Workqueue*).

Beispiel 6-12. Codebeispiel zur Verwendung der Event-Workqueue

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>

MODULE_LICENSE("GPL");

static void work_queue_function( void *data )
{
    pr_debug( "work_queue_function( %p, jiffies: %ld ) %d\n", data, jiffies,
            current->pid );
    return;
}

static DECLARE_WORK(work, work_queue_function, NULL );

static int __init mod_init(void)
{
    if( schedule_work( &work )==0 ) {
```

```
        pr_debug( "schedule_work not successful ...\n");
    } else {
        pr_debug( "schedule_work successful ...\n");
    }
    return 0;
}

static void __exit mod_exit(void)
{
    pr_debug("mod_exit called\n");
    flush_scheduled_work();
}

module_init( mod_init );
module_exit( mod_exit );
```

Im Gegensatz zu einer selbst erzeugten Workqueue, die nur von dem eigenen Treiber verwendet wird, haben sämtliche Kernelkomponenten Zugriff auf die Event-Workqueue. Das ist insbesondere dann kritisch, wenn eine hier eingehängte Funktion (entgegen der Konvention) den zur Event-Workqueue gehörigen Kernel-Thread in den Wartezustand versetzt. Dann nämlich müsste auch die eigene Funktion mit warten.

Zurück
Softirqs

Zum Anfang
Nach oben

Weiter
Kritische Abschnitte sichern

Lizenz