

## 3 Einführung in den Kern

*Dijkstra probably hates me.*

LINUS TORVALDS

In diesem Kapitel soll der grundlegende Aufbau des Systemkerns und das Zusammenspiel der wichtigsten Komponenten im Mittelpunkt stehen. Es ist Grundlage für das Verständnis der weiteren Kapitel. Bevor es jedoch so richtig losgeht, noch einige allgemeine Bemerkungen zum LINUX-Kern.

LINUX ist nicht auf dem Reißbrett entstanden, sondern hat sich evolutionär entwickelt und entwickelt sich noch weiter. Jede Funktion des Kerns wurde mehrfach geändert und erweitert, um Fehler zu beheben und neue Features einzubauen. Wer selbst schon an so einem großen Projekt gearbeitet hat, der weiß, wie schnell Programmcode unübersichtlich und fehlerhaft werden kann. LINUS TORVALDS hat es als Koordinator des LINUX-Projektes geschafft, den Kern übersichtlich zu gestalten und immer wieder von alten Überbleibseln zu säubern.

Trotzdem ist der LINUX-Kern sicherlich nicht in allen Punkten ein gutes Beispiel für strukturiertes Programmieren. Es gibt „Magic-Numbers“ im Programmtext statt Konstantendeklarationen in Headerfiles, inline expandierte Funktionen statt Funktionsaufrufen, goto-Anweisungen statt eines einfachen break, Assembleranweisungen statt C-Code und viele andere Unschönheiten mehr. Viele dieser Merkmale unstrukturierten Programmierens wurden jedoch bewusst eingearbeitet. Große Teile des Systemkerns sind zeitkritisch, deswegen wurde der Programmcode auf gutes Laufzeitverhalten und nicht auf gute Lesbarkeit optimiert. Das unterscheidet Linux zum Beispiel von MINIX (siehe [Tan90]), welches als „Lehrbetriebssystem“ geschrieben wurde und nie für den täglichen Einsatz gedacht war. Im Gegensatz dazu ist LINUX jedoch ein „richtiges“ Betriebssystem, und für ein solches ist der Kern bemerkenswert gut strukturiert.

Ziel unseres Buches ist es, die prinzipielle Funktionsweise des LINUX-Kerns zu erläutern. Deswegen stellen die in diesem und in den nächsten Kapiteln vorgestellten Algorithmen einen Kompromiss zwischen den Original-Quelltexten und einem gut lesbaren Programmcode dar, wobei darauf geachtet wurde, dass die Veränderungen leicht nachvollziehbar sind.

**Allgemeine Architektur** Seit den Anfängen von UNIX hat sich die interne Struktur von Betriebssystemen stark geändert. Damals war es revolutionär, dass der größte Teil des Kerns in einer höheren Programmiersprache, C, geschrieben wurde. Heute ist so etwas selbstverständlich. Der aktuelle Trend geht in Richtung einer Mikrokernel-Architektur, wie zum Beispiel dem Mach-Kern (vgl. [Tan86]) oder auch dem Kern von Windows-NT. Auch das Experimental-UNIX MINIX (vgl. [Tan90]) und das sich in Entwicklung befindliche Hurd-System sind Mikrokernel-basiert. Der eigentliche Kern stellt dabei nur das

absolut notwendige Minimum an Funktionalität (Interprozesskommunikation und Speicherverwaltung) zur Verfügung und kann deswegen klein und kompakt implementiert werden. Auf diesen Mikrokern aufbauend, wird die restliche Funktionalität des Betriebssystems in eigenständige Prozesse ausgelagert, die mit dem Mikrokern über eine wohldefinierte Schnittstelle kommunizieren. Der große Vorteil dieser Architekturen ist (neben einer gewissen Eleganz) eine auf den ersten Blick wartungsfreundlichere Struktur des Systems. Einzelne Komponenten arbeiten unabhängig voneinander, können sich nicht ungewollt beeinflussen und sind leichter austauschbar. Die Entwicklung neuer Komponenten wird vereinfacht.

Daraus ergibt sich auch ein Nachteil dieser Architekturen. Mikrokern-Architekturen erzwingen die Einhaltung der definierten Schnittstellen zwischen den einzelnen Komponenten und erschweren damit trickreiche Optimierungen. Außerdem ist die im Mikrokern benötigte Interprozesskommunikation auf heutigen Hardware-Architekturen aufwendiger als einfache Funktionsaufrufe. Das System wird dadurch etwas langsamer als traditionelle monolithische Kerne. Dieser leichte Geschwindigkeitsnachteil wird gern in Kauf genommen, da die heutige Hardware in der Regel schnell genug ist und da der Vorteil der einfacheren Wartbarkeit des Systems die Entwicklungskosten senkt. Erst in den letzten Jahren wurden Mikrokern Systeme gebaut, deren Performance es mit monolithischen Systemen aufnehmen kann. Dies ist aber noch ein Bereich aktiver Grundlagenforschung.

Mikrokern-Architekturen repräsentieren sicherlich die Zukunft der Betriebssystementwicklung. LINUX hingegen entstand auf der „langsamen“ 386-Architektur, der unteren Grenze für ein vernünftiges UNIX-System. Gutes Laufzeitverhalten durch Ausreizen aller Optimierungsmöglichkeiten stand bei der Entwicklung mit im Vordergrund. Das ist ein Grund dafür, warum LINUX in der klassischen monolithischen Kernarchitektur realisiert wurde. Ein weiterer Grund ist sicherlich, dass eine Mikrokern-Architektur ein sorgfältiges Systemdesign bedingt. Da LINUX evolutionär, aus Spaß am Systementwickeln, entstanden ist, war dies einfach nicht möglich.

Trotz des monolithischen Ansatzes ist LINUX keine chaotische Ansammlung von Programmcode. Die meisten Komponenten des Kerns werden nur über sauber definierte Schnittstellen angesprochen. Ein gutes Beispiel hierfür ist das Virtuelle Dateisystem (VFS), welches eine abstrakte Schnittstelle zu allen dateiorientierten Operationen darstellt. Auf das VFS gehen wir im Kapitel 6 näher ein. Das Chaos findet sich eher im Detail. An zeitkritischen Stellen sind Programmteile oftmals in „handoptimiertem“ C-Code oder gar in Assembler geschrieben und damit schwer zu verstehen. Zum Glück sind diese Programmfragmente selten und in der Regel recht gut kommentiert.

Wenn man sich die Codegrößen der einzelnen Komponenten des LINUX-Kerns ansieht, stellt man fest, dass der überwiegende Teil auf Gerätetreiber und Ähnliches entfällt. Die zentralen Routinen zur Prozess- und Speicherverwaltung (also der eigentliche Kern im Sinne einer Mikrokern-Architektur) sind dagegen mit jeweils 13.000 Zeilen C-Code relativ klein und überschaubar.

Inzwischen ist es möglich, einen Großteil der Treiber aus dem Kern auszulagern. Sie können als eigenständige, unabhängige Module (siehe Kapitel 9) bei Bedarf zur Laufzeit

nachgeladen werden. LINUX versucht damit erfolgreich, die Vorteile einer Mikrokernarchitektur zu nutzen, ohne den monolithischen Entwurf aufzugeben.

**Prozesse und Tasks** Aus der Sicht eines unter LINUX ablaufenden Prozesses stellt sich der Kern als Anbieter von Dienstleistungen dar. Einzelne Prozesse existieren unabhängig nebeneinander und können sich nicht direkt beeinflussen. Der eigene Speicherbereich ist vor dem Zugriff fremder Prozesse geschützt.

Die interne Sicht auf ein laufendes LINUX-System ist etwas anders. Auf dem Rechner läuft nur ein Programm — das Betriebssystem —, welches auf alle Ressourcen zugreifen kann. Die verschiedenen Tasks werden durch Coroutinen realisiert, d.h., jede Task entscheidet selbst, ob und wann sie die Steuerung an eine andere Task abgibt.<sup>1</sup> Eine Konsequenz daraus ist, dass ein Fehler in der Kernprogrammierung das ganze System blockieren kann. Jede Task kann auf alle Ressourcen anderer Tasks zugreifen und diese modifizieren.

Bestimmte Teile einer Task laufen in einem weniger privilegierten Nutzermodus des Prozessors ab. Diese Teile der Task erscheinen nach außen hin (in der externen Sicht auf den Kern) als Prozesse. Aus Sicht dieser Prozesse findet ein echtes Multitasking statt. Die Abbildung 3.1 soll das verdeutlichen.

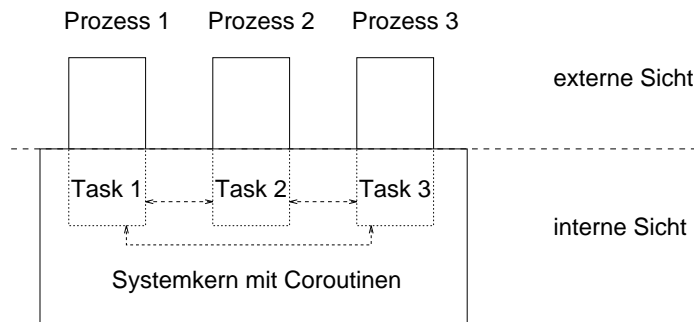


Abbildung 3.1: Verhältnis von interner und externer Sicht auf die Prozesse

Im Folgenden wollen wir allerdings auf eine exakte Unterscheidung der Begriffe *Task* und *Prozess* verzichten und diese Begriffe synonym gebrauchen. Dieses einfache Prozessmodell wird allerdings insofern erweitert, dass es auch Threads geben kann, die nur im Kernelmodus existieren. Wenn sich eine Task im privilegierten Systemmodus befindet, kann sie verschiedene Zustände annehmen. Abbildung 3.2 zeigt die wichtigsten dieser Zustände. Die Pfeile geben die möglichen Zustandsübergänge in diesem Diagramm an.

Die folgenden Zustände sind möglich:

**In Ausführung** Die Task ist aktiv und befindet sich im nichtprivilegierten Nutzermodus. In dem Fall arbeitet der Prozess ganz normal das Programm ab. Dieser Zustand kann nur durch einen Interrupt oder einen Systemruf verlassen werden. In Abschnitt

<sup>1</sup> Dies wird auch als kooperatives Multitasking bezeichnet.

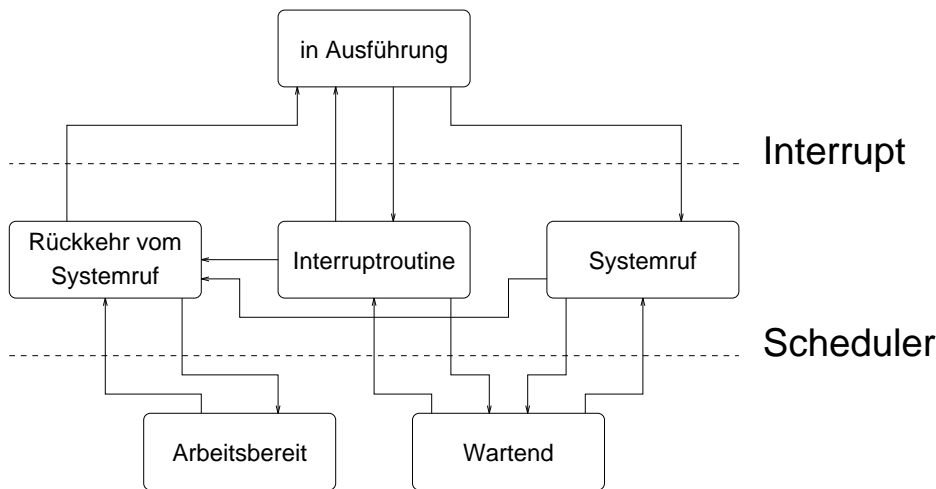


Abbildung 3.2: Zustandsgraph eines Prozesses

3.3 werden wir sehen, dass Systemrufe auch nur Spezialfälle von Interrupts sind. In jedem Fall wird der Prozessor in den privilegierten Systemmodus geschaltet und die entsprechende Interruptroutine aktiviert.

**Interruptroutine** Die Interruptroutinen werden aktiv, wenn die Hardware eine Ausnahmebedingung signalisiert, sei es, dass neue Zeichen an der Tastatur anliegen oder dass der Zeitgeberbaustein alle 10 Millisekunden ein Signal sendet. Weitere Informationen zu Interruptroutinen sind in Abschnitt 3.2.2 enthalten.

**Systemruf** Systemrufe werden durch softwaremäßig ausgelöste Interrupts eingeleitet. Nähere Informationen dazu finden Sie in Abschnitt 3.3. Ein Systemruf hat die Möglichkeit, seine Arbeit explizit zu unterbrechen, um auf ein Ereignis zu warten.

**Wartend** Der Prozess wartet auf ein externes Ereignis. Erst nachdem dieses eingetreten ist, setzt der Prozess seine Arbeit fort.

**Rückkehr vom Systemruf** Dieser Zustand wird automatisch nach jedem Systemaufruf und nach einigen Interrupts erreicht. Hier wird geprüft, ob der Scheduler aufgerufen werden muss und ob Signale zu behandeln sind. Der Scheduler kann den Prozess in den Zustand „arbeitsbereit“ überführen und einen anderen Prozess aktivieren.

**Arbeitsbereit** Der Prozess bewirbt sich um den Prozessor, dieser ist aber im Moment von einem anderen Prozess belegt.

**Prozesse und Threads** In vielen modernen Betriebssystemen gibt es die Unterscheidung von Prozessen und Threads. Ein Thread ist dabei ein unabhängiger „Faden“ im Ablauf eines Programmes, der parallel zu anderen Threads abgearbeitet werden kann. Im Unterschied zu Prozessen arbeiten die Threads über demselben Hauptspeicher und können sich so gegenseitig beeinflussen.

LINUX macht diese Unterscheidung nicht. Im Kern gibt es nur den Begriff der *Task*, diese kann sich mit anderen *Tasks* Ressourcen (wie zum Beispiel denselben Speicher) teilen. Damit ist die *Task* eine Verallgemeinerung des sonst üblichen Thread-Konzeptes. Näheres dazu finden Sie im Abschnitt 3.3.3.

**Multiprozessorsysteme** Seit der Version 2.0 unterstützt Linux SMP (*Symmetric Multi Processing*). Während in der Version 2.0 die Implementation anfänglich noch trivial war — es konnte zu einem Zeitpunkt nur ein Prozessor Kernelcode abarbeiten —, ist sie mittlerweile ziemlich komplex geworden. Mehrere Prozessoren können jetzt gleichzeitig Kernelcode abarbeiten. Deswegen muss der Zugriff auf alle globalen Datenstrukturen des Kernels synchronisiert werden. Die dadurch auftretenden Probleme haben wir in diesem Kapitel größtenteils ignoriert, um die Beschreibung verständlich zu halten.

## 3.1 Wichtige Datenstrukturen

In diesem Kapitel werden wichtige Datenstrukturen des LINUX-Kerns beschrieben. Das Verständnis dieser Strukturen und ihres Zusammenspiels ist Voraussetzung für das Verständnis der weiteren Kapitel.

### 3.1.1 Die Taskstruktur

Einer der wichtigsten Begriffe in einem Multitaskingsystem wie LINUX ist die *Task*. Die Datenstrukturen und Algorithmen zur Prozessverwaltung sind der zentrale Kern von LINUX.

Die Beschreibung der Eigenschaften eines Prozesses erfolgt in der Struktur `task_struct`, welche im Folgenden erläutert wird. Auf die ersten Komponenten der Struktur wird auch aus Assemblerrouninen heraus zugegriffen. Dieser Zugriff erfolgt nicht wie in C üblich über die Namen der Komponenten, sondern über ihren Offset relativ zum Anfang der Struktur. Deswegen darf man den Anfang der Taskstruktur auch nicht modifizieren, ohne dass vorher alle Assemblerrouninen überprüft und gegebenenfalls angepasst werden.

```
struct task_struct
{
    volatile long state;
```

`state` enthält eine Codierung für den aktuellen Zustand des Prozesses. Wenn der Prozess auf die Zuteilung der CPU wartet oder gerade läuft, hat `state` den Wert `TASK_RUNNING`. Wartet der Prozess dagegen auf bestimmte externe Ereignisse und ist deswegen im Moment stillgelegt, hat `state` den Wert `TASK_INTERRUPTIBLE` oder `TASK_UNINTERRUPTIBLE`. Der Unterschied zwischen diesen Werten besteht darin, dass im Zustand `TASK_INTERRUPTIBLE` ein Prozess durch Signale wieder aktiviert werden kann, während er im Zustand `TASK_UNINTERRUPTIBLE` in der Regel direkt oder indirekt auf eine Hardwarebedingung wartet und damit keine Signale akzeptiert. `TASK_STOPPED` beschreibt einen Prozess, dessen Ausführung angehalten worden ist.

Dies ist entweder nach dem Empfang eines entsprechenden Signales (SIGSTOP, SIGSTP, SIGTTIN oder SIGTTOU) der Fall, oder wenn der Prozess von einem anderen Prozess durch den Systemruf *ptrace* überwacht wird und die Steuerung an den überwachenden Prozess übergeben hat. TASK\_ZOMBIE beschreibt einen Prozess, der beendet wurde, dessen Taskstruktur sich aber noch in der Prozesstabelle befinden muss (vgl. Systemrufe *\_exit* und *wait* in Abschnitt 3.3.3). Das Schlüsselwort *volatile* deutet an, dass diese Komponente auch asynchron aus Interruptroutinen heraus geändert werden kann.

```
unsigned long flags;
```

*flags* enthält die Kombination der Statusflags PF\_ALIGNWARN, PF\_STARTING, PF\_EXITING, PF\_FORKNOEXEC, PF\_SUPERPRIV, PF\_DUMPCORE, PF\_SIGNALED, PF\_MEMALLOC, PF\_VFORK und PF\_USEDFFU.

Diese Flags werden im Wesentlichen für das Accounting von Prozessen benutzt und haben auf die Arbeitsweise des Systems keinen weiteren Einfluss.

Das in älteren Kernen vorhandene Statusflag PF\_TRACED wurde in die Komponente

```
unsigned long ptrace;
```

ausgelagert. Die Werte PF\_PTRACED und PF\_TRACESYS zeigen hier an, dass der Prozess von einem anderen Prozess mit Hilfe des Systemrufes *ptrace* überwacht wird. Nähere Informationen zu diesem Systemruf findet der interessierte Leser in Abschnitt 5.4 und Anhang A.

```
int sigpending;
```

Das Flag *sigpending* ist gesetzt, wenn an diesen Prozess Signale ausgeliefert werden müssen. Näheres dazu findet sich im Abschnitt 3.2.1.

```
mm_segment_t addr_limit;
```

Im Gegensatz zu älteren Kernen kann es seit Version 2.4 auch Tasks (Threads) innerhalb des Kernels geben. Diese dürfen selbstverständlich auf einen größeren Adressraum zugreifen als Tasks im Userspace. *addr\_limit* beschreibt den Adressraum, auf dem der Kernel der Tasks Zugriff ermöglicht.

```
struct exec_domain *exec_domain;
```

Linux kann Programme anderer UNIX-Systeme auf i386-Basis, die dem iBCS2-Standard entsprechen, abarbeiten. Da sich die verschiedenen iBCS2-Systeme leicht unterscheiden, wird für jeden Prozess in der Komponente *exec\_domain* eine Beschreibung mitgeführt, welches UNIX für diesen Prozess emuliert werden soll.

```
long need_resched;
```

*need\_resched* ist ein Flag, welches anzeigt, dass eine Scheduling durchgeführt werden muss. In der Kernelversion 2.0 war dies noch eine globale Variable, aus Effizienzgründen ist diese jetzt in der Taskstruktur der aktuellen Task abgelegt.

Für den Betrieb auf Multiprozessorsystemen muss die ganze Struktur vor dem gleichzeitigen Zugriff geschützt werden. Das Locking wird über die Komponente

```
int lock_depth;
```

realisiert.

Damit endet der hartcodierte Teil der Taskstruktur. Die folgenden Komponenten der Taskstruktur sind der Übersicht halber zu Gruppen zusammengefasst.

```
long counter;
long nice;
unsigned long policy; /* SCHED_FIFO, SCHED_RR, *
                      * SCHED_OTHER          */
unsigned long rt_priority;
```

`counter` enthält die Zeit in „Ticks“ (siehe Abschnitt 3.2.5), die der Prozess noch laufen kann, ehe zwangsweise ein Scheduling durchgeführt wird. Der Scheduler benutzt den Wert in `counter`, um den nächsten Prozess auszuwählen. Damit stellt `counter` so etwas wie die dynamische Priorität eines Prozesses dar. `nice` enthält die statische Priorität des Prozesses. Bis zur Kernelversion 2.2 hatte diese Komponente den Namen `priority`. Der Schedulingalgorithmus (siehe Abschnitt 3.2.6) verwendet `nice`, um im Bedarfsfall einen neuen Wert für `counter` zu ermitteln.

LINUX unterstützt inzwischen mehrere Scheduling-Algorithmen. Neben dem klassischen Scheduling (`SCHED_OTHER`) gibt es jetzt auch noch zwei in POSIX.4 beschriebene Real-Time-Scheduling-Algorithmen (`SCHED_RR` und `SCHED_FIFO`). Jeder Prozess kann in eine dieser Schedulingklassen eingeordnet werden. Diese wird, zusammen mit der Real-Time-Priorität, in den Komponenten `policy` und `rt_priority` vermerkt. Näheres dazu im Abschnitt 3.2.6.

### Signale

```
/* int sigpending */
sigset_t blocked;
struct signal_struct *sig;

struct sigpending pending;
```

`sigpending` enthält, wie oben bereits beschrieben, eine Bitmaske der für den Prozess eingetroffenen Signale und `blocked` eine Bitmaske aller Signale, die der Prozess erst später bearbeiten möchte, d.h. deren Bearbeitung im Moment blockiert ist. Die Komponente `sig` enthält Verweise auf die entsprechenden Signalbehandlungsroutinen.

LINUX unterstützt sogenannte „verlässliche Signale“ oder auch „Realtime-Signale“ nach POSIX.4. Diese können nicht wie normale UNIX-Signale in einem Bitfeld verwaltet werden; der Kernel muss dafür sorgen, dass ein mehrfach gesendetes Signal den Empfänger auch mehrfach erreicht. LINUX versucht dieses Verhalten auch für traditionelle Signale zu implementieren. Jedes an den Prozess gesendete Signal wird deswegen in einer Liste `pending` vermerkt. Die Auswertung dieser Signalinformation ist im Abschnitt 3.2.1 beschrieben.

**Prozessrelationen** Alle Prozesse sind mit Hilfe der folgenden beiden Komponenten in eine doppelt verkettete Liste eingetragen.

```
struct task_struct *next_task;
struct task_struct *prev_task;
```

Den Anfang und das Ende dieser Liste enthält die globale Variable `init_task`.

Prozesse existieren in einem UNIX-System nicht unabhängig voneinander. Jeder Prozess (außer dem Prozess `init_task`) hat einen Elternprozess, der ihn mit Hilfe des Systemrufs `fork()` (siehe Abschnitt 3.3.3 und Anhang A) erzeugt hat. Daraus ergeben sich Verwandtschaftsbeziehungen zwischen den Prozessen, die durch die folgenden Komponenten repräsentiert werden:

```
struct task_struct *p_opptr; /* original parent */
struct task_struct *p_pptr; /* parent */
struct task_struct *p_cptra; /* youngest child */
struct task_struct *p_ysptr; /* younger sibling */
struct task_struct *p_osptr; /* older sibling */
```

`p_pptr` ist ein Zeiger auf die Taskstruktur des Elternprozesses. Damit ein Prozess auf alle seine Kinderprozesse zugreifen kann, enthält die Taskstruktur den Eintrag für den zuletzt erzeugten Kindprozess – das „jüngste“ Kind (*youngest child*). Die Kindprozesse desselben Elternprozesses sind untereinander wiederum durch `p_ysptr` (*younger sibling* = nächstjüngeres Kind) und `p_osptr` (*older sibling* = nächstälteres Kind) als doppelt verkettete Liste verbunden. Die Abbildung 3.3 versucht, die Verwandtschaftsbeziehungen zwischen Prozessen etwas zu verdeutlichen. Der Scheduler benutzt eine Liste aller Prozesse, die sich um den Prozessor bewerben. Diese wird über die Komponente

```
struct list_head run_list;
```

realisiert.

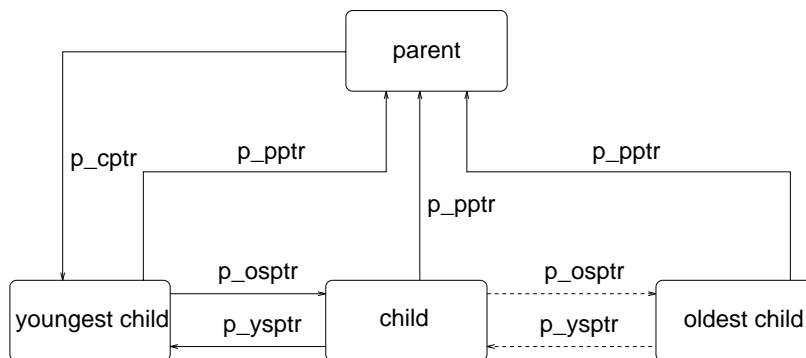


Abbildung 3.3: Verwandtschaftsbeziehungen von Prozessen

**Speicherverwaltung** Die für jeden Prozess notwendigen Daten zur Speicherverwaltung werden aus Gründen der Übersichtlichkeit in einer eigenen Unterstruktur

```
struct mm_struct *mm;
```

zusammengefasst. Diese hat die Komponenten:

```
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk,
unsigned long start_stack, start_mmap;
unsigned long arg_start, arg_end, env_start, env_end;
...
```

welche den Beginn und die Größe der Code- und Datensegmente für das aktuell laufende Programm beschreiben. Weitergehende Informationen sind in Kapitel 4 enthalten.

**Prozessidentifikation** Jeder Prozess besitzt seine eindeutige Prozessidentifikationsnummer `pid`, ist einer Prozessgruppe `pgrp` und einer Sitzung `session` zugeordnet. Jede Sitzung hat einen führenden Prozess (`leader`). Da unter LINUX auch Threads durch eine eigene Task realisiert werden, wurde mit `tgid` die Threadgroup-Identifikationsnummer eingeführt. Es ist dies in der Regel die `pid` des Prozesses, aus dem heraus neue Threads gestartet werden. Im klassischen Sinne ist dies also die wahre PID.

```
pid_t pid, pgrp, session, tgid;
int leader;
```

Zur Realisierung von Zugriffsrechten besitzt jeder Prozess eine Nutzeridentifikation (*User Identification*) `uid` und die Gruppenidentifikation (*Group Identification*) `gid`. Diese werden beim Erzeugen eines neuen Prozesses durch den Systemruf `fork` (siehe Abschnitt 3.3.3 und Anhang A) vom Elternprozess an den Kindprozess vererbt. Für die eigentliche Zugriffskontrolle werden allerdings die effektive Nutzeridentifikation `euid` und die effektive Gruppenidentifikation `egid` benutzt. Eine Neuerung in LINUX ist die Komponente `fsuid`. Diese wird bei allen Identifikationen gegenüber dem Dateisystem benutzt. Normalerweise gilt `(uid == euid) && (gid == egid)` und `(fsuid == euid) && (fsgid == egid)`.

Ausnahmen ergeben sich bei sogenannten Set-UID-Programmen, bei denen die Werte `euid` und `fsuid` bzw. `egid` und `fsgid` auf die Nutzer-ID bzw. die Gruppen-ID des Eigentümers der ausführbaren Datei gesetzt wird. Dadurch ist eine kontrollierte Vergabe von Privilegien möglich.

Normalerweise hat `fsuid` immer den Wert von `euid`, und in anderen UNIX-Systemen oder in älteren LINUX-Versionen wurde anstelle von `fsuid` auch immer die effektive Nutzeridentifikation `euid` benutzt. LINUX erlaubt hingegen durch den Systemruf `setfsuid` das Ändern der `fsuid` ohne Änderung der `euid`. Damit können Dämonen mit `setfsuid` ihre Rechte in Bezug auf Dateisystemzugriffe einschränken (auf die Rechte des Nutzers, für den Dienstleistungen erbracht werden), behalten aber ihre Privilegien bei. Analoges gilt für die Komponente `fsgid` und den Systemruf `setfsgid`.

```
uid_t uid, euid, suid, fsuid;
gid_t gid, egid, sgid, fgid;
```

LINUX erlaubt, wie die meisten modernen UNIX-Derivate, die gleichzeitige Zuordnung eines Prozesses zu mehreren Nutzergruppen. Diese Gruppen werden bei der Kontrolle der Zugriffsrechte auf Dateien berücksichtigt. Jeder Prozess kann maximal `NGROUPS` Gruppen angehören, die in der Komponente `groups` der Taskstruktur abgespeichert werden.

```
gid_t groups[NGROUPS];
int ngroups;
```

Traditionell sind in einem UNIX-System viele Aktionen dem Superuser vorbehalten. Dieser wird daran erkannt, dass seine effektive UID 0 ist. Nur mit dieser EUID kann ein Prozess zum Beispiel eine Netzwerkverbindung auf einem privilegierten Port eröffnen, ein Signal an einen fremden Prozess senden oder das System rebooten. Dieses Konzept der Privilegien ist relativ grob und führte in der Vergangenheit zu vielen Sicherheitsproblemen in UNIX-Systemen. LINUX hat zusätzlich zum Konzept „Superuser“ noch das Konzept der „Capabilities“ eingeführt. Damit ist es möglich, einem Prozess zum Beispiel gezielt das Recht einzuräumen, einen reservierten Netzwerkport zu eröffnen, ohne ihm gleich Superuserprivilegien zu gewähren. Diese Rechte werden in den Komponenten

```
kernel_cap_t cap_effective;
kernel_cap_t cap_inheritable;
kernel_cap_t cap_permitted;

int keep_capabilities:1;
```

verwaltet und im Kern bereits Stellen benutzt. Eine Liste der möglichen Werte für diese Komponenten ist in `include/linux/capability.h` zu finden. Obwohl der LINUX-Kern inzwischen durchgängig auf Capabilities umgestellt ist, werden diese von den Verwaltungsprogrammen und den üblichen Dateisystemen noch nicht unterstützt. Für die Zukunft zeichnet sich hier aber eine interessante Entwicklung zu sichereren Systemen ab.

**Files** Die dateisystemspezifischen Informationen sind in der Unterstruktur

```
struct fs_struct *fs;
```

abgelegt. Diese enthält unter anderem die Komponenten

```
atomic_t count;
int umask;
struct dentry * root, * pwd;
```

Ein Prozess kann über den Systemruf `umask` den Zugriffsmodus von neu zu erzeugenden Dateien beeinflussen. Die mit dem Systemruf `umask` gesetzten Werte werden dazu in der Komponente `umask` abgelegt. Unter UNIX besitzt jeder Prozess ein aktuelles Verzeichnis `pwd`<sup>2</sup>, welches bei der Auflösung relativer Pfadnamen benötigt wird und mit dem Systemruf `chdir` geändert werden kann. Jeder Prozess verfügt weiterhin über ein

<sup>2</sup> Die Abkürzung `pwd` steht hier höchstwahrscheinlich in Anlehnung an das UNIX-Kommando `pwd` (*print working directory*), welches den Namen des aktuellen Verzeichnisses ausgibt.

eigenes Wurzelverzeichnis `root`, das zum Auflösen absoluter Pfadnamen benutzt wird. Dieses Root-Verzeichnis kann nur vom Superuser geändert werden (Systemruf `chroot`). Da dies nur in wenigen Fällen genutzt wird (z. B. anonymous FTP), ist diese Tatsache weniger bekannt. `count` ist ein Referenzzähler, da diese Struktur von mehreren Tasks benutzt werden kann.

Ein Prozess, der eine Datei mit `open()` oder `creat()` eröffnet, erhält vom Kern einen Dateideskriptor für die weitere Referenzierung dieser Datei. Dateideskriptoren sind kleine Integerzahlen. Die Zuordnung der Dateideskriptoren zu den Dateien erfolgt unter LINUX über das Feld `fd` in der Unterstruktur

```
struct files_struct *files;
```

Sie hat unter anderem folgende Komponenten

```
atomic_t count;
int max_fds;
struct file ** fd;
fd_set *close_on_exec;
fd_set *open_fds;
```

Dateideskriptoren werden als Index im Feld `fd` benutzt. Man findet auf diese Weise den dem Dateideskriptor zugeordneten Filepointer, mit dessen Hilfe man dann auf die Datei zugreifen kann. `open_fds` ist eine Bitmaske aller benutzten Filedeskriptoren.

Die Komponente `close_on_exec` in der Unterstruktur `files` enthält eine Bitmaske aller benutzten Filedeskriptoren, die beim Systemruf `exec` geschlossen werden sollen. `count` dient wieder als Referenzzähler; `max_fds` ist die Maximalanzahl von offenen Filedescriptoren für den Prozess.

**Zeitmessung** Für jeden Prozess werden mehrere unterschiedliche Zeiten gemessen. Die Zeitmessung wird unter LINUX grundsätzlich in Ticks vorgenommen. Diese Ticks werden von einem Zeitgeberbaustein der Hardware alle 10 Millisekunden erzeugt und vom Timerinterrupt gezählt. In den Abschnitten 3.1.6 und 3.2.5 gehen wir genauer auf die Zeitmessung unter LINUX ein.

`per_cpu_utime[]` und `per_cpu_stime[]` enthalten die Zeit, die der Prozess im Nutzermodus bzw. im Systemmodus verbraucht hat. Diese Daten werden in einem Multiprozessorsystem für jede CPU einzeln erhoben. Die Summe dieser Werte über alle CPUs sowie die Summe der entsprechenden Zeiten aller Kindprozesse sind in der Komponente `times` abgelegt. Die Werte können mit Hilfe des Systemrufs `times` abgefragt werden.

```
long per_cpu_utime[NR_OF_CPUS];
long per_cpu_stime[NR_OF_CPUS];
struct tms times;
unsigned long start_time;
```

`start_time` enthält den Zeitpunkt, zu dem der aktuelle Prozess erzeugt wurde.

UNIX unterstützt verschiedene prozessspezifische Timer. Zum einen gibt es den Systemruf *alarm*, der dafür sorgt, dass dem Prozess nach einer bestimmten Zeit das Signal SIGALRM gesandt wird. Neuere UNIX-Systeme unterstützen zusätzlich Intervalltimer (siehe Systemrufe *setitimer* und *getitimer* auf Seite 331).

```
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
```

Die Komponenten *it\_real\_value*, *it\_prof\_value* und *it\_virt\_value* enthalten die Zeitspanne in Ticks, nach der der Timer abgelaufen ist. In den Komponenten *it\_real\_incr*, *it\_prof\_incr* und *it\_virt\_incr* befinden sich die notwendigen Werte, um den Timer nach Ablauf wieder zu initialisieren. *real\_timer* dient der Realisierung des Realzeit-Intervalltimers. Näheres dazu findet sich auch bei der Beschreibung des Timerinterrupts in Abschnitt 3.2.5.

**Interprozesskommunikation** Im LINUX-Kern ist eine zum SYSTEM V kompatible Interprozesskommunikation implementiert, die unter anderem Semaphore zur Verfügung stellt. Ein Prozess kann einen Semaphore belegen und damit sperren. Wenn andere Prozesse diesen Semaphore auch belegen wollen, werden sie so lange gestoppt, bis der Semaphore freigegeben wird. Dazu dient die Komponente

```
struct sem_queue *semsleeping;
```

Wenn der Prozess beendet wird, muss das Betriebssystem alle vom Prozess belegten Semaphore freigeben. Die Komponente

```
struct sem_undo *semundo;
```

enthält die dazu notwendigen Informationen.

**Verschiedenes** Die folgenden Komponenten lassen sich nicht in die obigen Gruppen einordnen.

```
wait_queue_head_t wait_chldexit;
```

Ein Prozess, der den Systemruf *wait4* ausführt, soll bis zur Beendigung eines Kindprozesses unterbrochen werden. Dazu trägt er sich in die Warteschlange *wait\_chldexit* seiner eigenen Taskstruktur ein, setzt sein Statusflag auf den Wert *TASK\_INTERRUPTIBLE* und gibt die Steuerung an den Scheduler ab. Wenn ein Prozess beendet wird, signalisiert er dies seinem Elternprozess über diese Warteschlange. Näheres findet sich im Abschnitt über Warteschlangen (Abschnitt 3.1.5), im Abschnitt zu den Systemrufen *\_exit* und *wait* (Abschnitt 3.3.3) sowie in den Quelltexten zur Kernelfunktion *sys\_wait4()* (*kernel/exit.c*).

```
struct semaphore *vfork_sem;
```

Die Semantik des Systemaufrufes *vfork* verlangt, dass der Elternprozess mit der Weiterarbeit solange warten muss, bis der Kindprozess entweder beendet wurde oder über *exec*

ein anderes Programm geladen hat. Dieses Warten wird über die Semaphore `vfork_sem` realisiert. Näheres findet sich im Abschnitt 3.3.3

```
struct rlimit rlim[RLIM_NLIMITS];
```

Jeder Prozess kann mit Hilfe der Systemrufe `setrlimit` und `getrlimit` (siehe Seite 332) seine Limits für die Verwendung von Ressourcen kontrollieren. Sie werden in der Struktur `rlim` abgespeichert.

```
int exit_code, exit_signal;
```

Dies sind der Return-Code des Programms und das Signal, mit dem das Programm abgebrochen wurde. Diese Informationen kann ein Elternprozess nach dem Ende eines Kindprozesses abfragen.

```
char comm[16];
```

Der Name des vom Prozess ausgeführten Programms ist in der Komponente `comm` gespeichert. Dieser Name wird für das Debugging benutzt.

```
unsigned long personality;
```

Wie schon erwähnt, unterstützt LINUX über das iBCS-Interface das Abarbeiten von Programmen anderer UNIX-Systeme. Zusammen mit der oben beschriebenen Komponente `exec_domain` dient `personality` der genauen Beschreibung der Eigenarten dieser UNIX-Version. Für normale LINUX-Programme hat `personality` den Wert `PER_LINUX` (definiert in `<linux/personality.h>` als 0)

```
int dumpable:1;
int did_exec:1;
```

Das Flag `dumpable` gibt an, ob vom laufenden Prozess beim Eintreffen bestimmter Signale ein Speicherabzug erzeugt werden soll.

Eine ziemlich obskure Semantik im POSIX-Standard erfordert beim Systemruf `setpgid` die Unterscheidung, ob ein Prozess noch das ursprüngliche Programm ausführt oder schon mit dem Systemruf `execve` ein neues Programm geladen hat. Diese Information wird im Flag `did_exec` verwaltet.

Eine weitere wichtige Komponente in der Taskstruktur ist `binfmt`. Sie beschreibt die Funktionen, die für das Laden des Programms zuständig sind.

```
struct linux_binfmt *binfmt;
struct thread_struct thread;
```

Die `thread_struct`-Struktur enthält alle Informationen über den aktuellen Prozessorstatus zum Zeitpunkt des letzten Übergangs vom Nutzermodus in den Systemmodus. Hier sind alle Prozessorregister gerettet, damit sie bei der Rückkehr in den Nutzermodus wieder restauriert werden können.

### 3.1.2 Die Prozesstabelle

Jeder Prozess belegt genau einen Eintrag in der Prozesstabelle. Sie war bis LINUX 2.2 statisch angelegt und in der Größe auf NR\_TASKS (512) Tasks beschränkt. In der aktuellen Version ist die Prozesstabelle nur noch eine Abstraktion. Die einzelnen Tasks sind über die in der Struktur `task_struct` vorhandenen Verkettungen `next_task` und `prev_task` erreichbar.

Das Makro `init_task` zeigt auf die erste Task im System. Sie wird beim Starten des Systems (in Abschnitt 3.2.4 beschrieben) mit Hilfe des Makros `INIT_TASK` initialisiert. Diese ist nach dem Booten des Systems nur noch für den Verbrauch nichtbenötigter Systemzeit verantwortlich (Idle-Prozess). Sie fällt deswegen etwas aus dem Rahmen und sollte nicht als normale Task angesehen werden.

Obwohl die Prozesstabelle eine rein dynamische Struktur hat, wird die Anzahl der Tasks im System auf `max_threads` begrenzt.

```
int max_threads;
```

Dieser Wert kann jedoch zur Laufzeit über das `sysctl` Interface geändert werden.

Viele Algorithmen im Kern müssen jede einzelne Task berücksichtigen. Um dies zu erleichtern, wurde das Makro `for_each_task()` wie folgt definiert:

```
#define for_each_task(p) \
for( p = &init_task ; ( p = p->next_task) != &init_task ; )
```

Wie man sieht, wird die `init_task` übergangen. Der Eintrag für die aktuell laufende Task war in Version 1 noch über die globale Variable

```
struct task_struct *current;
```

zu erreichen. Da seit der Version 2.0 Multiprocessing (SMP) unterstützt wird, musste dies erweitert werden — es gibt jetzt für jeden Prozessor eine aktuelle Task.

```
#define current get_current()
inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("andl %%esp,%; ":"=r" (current) : "0" (~8191UL));
    return current;
}
```

Die Funktion `get_current()` ist etwas magisch, die Task-Struktur ist aus Effizienzgründen im Stacksegment der aktuellen Task untergebracht.

### 3.1.3 Files und Inodes

UNIX-Systeme unterscheiden traditionell zwischen einer File-Struktur und einer Inode-Struktur. Die Inode-Struktur beschreibt eine Datei. Dabei ist der Begriff *Inode* mehrfach

belegt. Sowohl die Datenstruktur im Kern als auch die Datenstrukturen auf der Festplatte beschreiben (jede aus ihrer Sicht) eine Datei und werden deswegen Inodes genannt. Wir reden im Folgenden immer von der im Hauptspeicher liegenden Datenstruktur. Inodes enthalten Informationen, wie etwa den Eigentümer und die Zugriffsrechte der Datei. Zu jeder im System benutzten Datei gibt es *genau einen* Inode-Eintrag im Kern.

File-Strukturen (die Datenstrukturen vom Typ `struct file`) enthalten dagegen die Sicht eines Prozesses auf diese (durch eine Inode repräsentierte) Datei. Zu dieser Sicht auf die Datei gehören Attribute, wie etwa der Modus, in dem die Datei benutzt werden kann (read, write, read+write), oder die aktuelle Position der nächsten I/O-Operation.

**File-Struktur** Die Struktur `file` ist in `include/linux/fs.h` definiert.

```
struct file
{
    mode_t f_mode;
    loff_t f_pos;

    atomic_t f_count;
    unsigned int f_flags;

    struct dentry *fs_dentry;
    struct file_operations * f_op;

    ...
};
```

`f_mode` beschreibt den Zugriffsmodus, in dem die Datei eröffnet wurde (nur Lesen, Lesen und Schreiben oder nur Schreiben). `f_pos` enthält die Position des Schreib-Lese-Zeigers, an der die nächste I/O-Operation vorgenommen wird. Dieser Wert wird durch jede I/O-Operation sowie durch die Systemrufe `lseek` und `llseek` aktualisiert. Man beachte, dass der Offset im Kern als 64-Bit-Wert vom Typ `loff_t` gespeichert wird. Damit ist LINUX in der Lage, Dateien größer als 2 Gigabyte ( $2^{31}$  Byte) korrekt zu behandeln.

`f_flags` enthält zusätzliche Flags, die den Zugriff auf diese Datei steuern. Diese können entweder beim Eröffnen einer Datei mit dem Systemruf `open` gesetzt und später mit dem Systemruf `fcntl` gelesen und modifiziert werden. `f_count` ist ein einfacher Referenzzähler. Mehrere Dateideskriptoren können auf dieselbe File-Struktur verweisen. Da diese durch den Systemruf `fork` vererbt werden, kann auch aus verschiedenen Prozessen auf dieselbe File-Struktur verwiesen werden. Beim Eröffnen einer Datei wird `f_count` mit 1 initialisiert. Jedes Kopieren des Dateideskriptors (durch die Systemrufe `dup`, `dup2` oder `fork`) erhöht den Referenzzähler um 1, während er bei jedem Schließen einer Datei (durch die Systemrufe `close`, `_exit` oder `exec`) um 1 dekrementiert wird. Die File-Struktur wird erst freigegeben, wenn kein Prozess mehr auf sie verweist.

`f_dentry` ist ein Verweis auf einen Eintrag im Verzeichniscache, welcher alle offenen Dateien enthält. Über diesen Eintrag kann auch auf die Inode (die eigentliche Beschreibung der Datei) zugegriffen werden. `f_op` verweist auf eine Struktur von Funktions-

pointern, die alle File-Operationen referenzieren. LINUX unterstützt im Vergleich zu anderen UNIX-Systemen sehr viele Dateisystemtypen. Jedes Dateisystem realisiert die Zugriffe auf eine andere Art. Deswegen ist in LINUX ein „Virtuelles Dateisystem“ (VFS) realisiert worden. Die Idee besteht darin, dass die auf dem Dateisystem operierenden Funktionen nicht direkt, sondern über eine `datei(system)`spezifische Funktion aufgerufen werden. Diese dateisystemspezifischen Operationen sind Teil der Struktur `file` bzw. `inode`. Das entspricht dem Prinzip virtueller Funktionen in objektorientierten Programmiersprachen. Ausführlichere Informationen zum VFS finden sich in Abschnitt 6.2.

### Inodes Die Inode-Struktur

```
struct inode
{
```

ist ebenfalls in `include/linux/fs.h` definiert. Viele Komponenten dieser Struktur können über den Systemruf `stat` abgefragt werden.

```
    kdev_t i_dev;
    unsigned long i_ino;
```

`i_dev` ist eine Beschreibung des Geräts (die Plattenpartition), auf der sich die Datei befindet. `i_ino`<sup>3</sup> identifiziert die Datei innerhalb des Geräts. Das Paar (`i_dev`, `i_ino`) beschreibt die Datei damit systemweit eindeutig.

```
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_mtime;
    time_t i_atime;
    time_t i_ctime;
```

Diese Komponenten beschreiben die Zugriffsrechte der Datei, ihre Eigentümer (Nutzer und Gruppe), die Größe `i_size` in Byte, die Zeiten der letzten Änderung `i_mtime`, des letzten Zugriffs `i_atime` sowie der letzten Änderung der Inode `i_ctime`.

```
    struct inode_operations * i_op;
    ...
```

Wie die File-Struktur besitzt auch die Inode einen Verweis auf eine Struktur, die Zeiger auf Funktionen enthält, die auf Inodes anwendbar sind (siehe Abschnitt 6.2.7). Weitere Informationen zur Inode befinden sich in Abschnitt 6.2.

## 3.1.4 Dynamische Speicherverwaltung

Unter LINUX wird der Speicher seitenweise verwaltet. Eine Seite umfasst  $2^{12}$  Bytes. Grundoperationen zum Anfordern neuer Seiten sind die Funktionen

<sup>3</sup> `i_ino` steht hier auch für die Inode. Damit ist in diesem Fall die Blocknummer der Datenstruktur auf der Festplatte gemeint, welche die Datei auf dem externen Speicher beschreibt.

```
struct page * __alloc_pages(int gfp_mask, unsigned long order);
unsigned long __get_free_pages(int gfp_mask
                               unsigned long order);
```

welche in der Datei `mm/page_alloc.c` definiert sind. `gfp_mask` gibt Auskunft darüber, wer zu welchem Zweck die Seiten anfordert und steuert darüber das Verhalten der Funktionen, wenn im Hauptspeicher zuwenig Seiten frei sind. Ein Userprozess kann dann zum Beispiel warten, bis wieder Speicher frei wird, dies ist einer Interruptroutine nicht möglich. Für `gfp_mask` sind dabei die Werte `GFP_BUFFER`, `GFP_ATOMIC`, `GFP_USER`, `GFP_KERNEL`, `GFP_KSWAPD` und `GFP_NFS` zulässig.

`order` beschreibt die Anzahl der zu reservierenden Seiten. Es werden  $2^{\text{order}}$  viele Seiten reserviert.

Angeforderte Seiten können mit den Funktionen

```
void __free_pages(struct page *page, unsigned long order);
void free_pages(unsigned long addr, unsigned long order);
```

wieder freigegeben werden. Dadurch werden die Seiten wieder in die Freispeicherliste eingetragen.

Obwohl dies die Grundoperation für die Anforderung einer Seite darstellt, sollte man sie in dieser Form nicht benutzen. Besser geeignet ist die Funktion

```
unsigned long get_zeroed_page(int gfp_mask);
```

Sie initialisiert den angeforderten Speicher zusätzlich mit 0. Das ist aus zwei Gründen wichtig. Erstens erwarten einige Teile des Kerns, dass neu angeforderter Speicher mit 0 initialisiert ist (z.B. der Systemruf `exec`). Andererseits handelt es sich um eine Sicherheitsmaßnahme. Wenn die Seite vorher schon benutzt wurde, enthält sie vielleicht Daten eines anderen Nutzers (z.B. Passwörter), die dem aktuellen Prozess nicht zugänglich gemacht werden sollen.

Normalerweise ist der C-Programmierer an den Umgang mit `malloc()` und `free()` zur Speicherverwaltung gewohnt. Etwas Ähnliches gibt es auch im LINUX-Kern. Die Funktion

```
void *kmalloc(size_t size, int flags);
```

arbeitet analog zu `malloc()`. Das Argument `flags` gibt, ähnlich wie bei `get_zeroed_page()` an, wie `kmalloc()` neue Speicherseiten anfordern soll. Das Gegenstück zu `kmalloc()` ist die Funktion

```
void kfree( const void * objp);
```

welche einen mit `kmalloc()` angeforderten Speicherbereich wieder freigibt.

Weitere Informationen zur Funktionsweise der Speicherverwaltung unter LINUX findet der Leser in Kapitel 4.

### 3.1.5 Warteschlangen und Semaphore

Oftmals ist ein Prozess vom Eintreten bestimmter Bedingungen abhängig. Sei es, dass der Systemruf *read* darauf warten muss, dass die Daten von der Festplatte in den Speicherbereich des Prozesses geladen werden, oder dass ein Elternprozess mit *wait* auf das Ende eines Kindprozesses wartet. In jedem dieser Fälle ist nicht bekannt, wie lange ein Prozess warten muss.

Dieses „Warten bis eine Bedingung erfüllt ist“ wird in LINUX mit Hilfe der Warteschlangen (*Waitqueues*) realisiert. Eine Warteschlange ist im Prinzip nichts anderes als eine zyklische Liste, welche als Elemente Zeiger auf Tasks enthält.

```
struct list_head {
    struct list_head *next, *prev;
};

typedef struct __wait_queue {
    struct task_struct * task;
    struct list_head task_list;
} wait_queue_t;

typedef struct __wait_queue_head {
    struct list_head task_list;
} wait_queue_head_t;
```

Zyklische Listen sind eine grundlegende Datenstruktur, deswegen wurde mit der Struktur *list\_head* und den dazugehörigen Funktionen, eine einheitliche Implementation für sie geschaffen.

Der Datentyp *wait\_queue\_t* beschreibt ein Element einer Warteschlange und *wait\_queue\_head\_t* die Warteschlange selbst.

Warteschlangen sollten nur mit Hilfe der folgenden beiden Funktionen modifiziert werden. Diese sorgen durch entsprechenden Locking dafür, dass der Zugriff auf die Warteschlangen synchronisiert erfolgt.

```
void add_wait_queue(wait_queue_head_t *q,
                   wait_queue_t *wait);

void remove_wait_queue(wait_queue_head_t *q,
                       wait_queue_t *wait);
```

*q* enthält jeweils die zu modifizierende Warteschlange und *wait* den hinzuzufügenden oder zu entfernenden Eintrag.

Ein Prozess, der auf ein bestimmtes Ereignis warten will, trägt sich nun in eine solche Warteschlange ein und gibt die Steuerung ab. Zu jedem möglichen Ereignis gibt es eine Warteschlange. Wenn das entsprechende Ereignis eintritt, werden alle Prozesse in dieser Warteschlange wieder aktiviert und können weiterarbeiten. Diese Semantik wird durch die Funktionen

```

void sleep_on(wait_queue_head_t *q);
void sleep_on_timeout(wait_queue_head_t *q, long timeout);
void interruptible_sleep_on(wait_queue_head_t *q);
void interruptible_sleep_on_timeout(wait_queue_head_t *q,
                                   long timeout);

```

realisiert. Sie setzen den Status des Prozesses (`current->state`) auf den Wert `TASK_UNINTERRUPTIBLE` beziehungsweise `TASK_INTERRUPTIBLE`, tragen den aktuellen Prozess (`current`) in die Warteschlange ein und rufen den Scheduler auf. Damit gibt der Prozess die Steuerung freiwillig ab.

Er wird erst wieder aktiviert, wenn der Prozessstatus auf `TASK_RUNNING` gesetzt wurde. Das geschieht in der Regel dadurch, dass ein anderer Prozess eines der Makros

```

void wake_up(struct wait_queue **p);
void wake_up_interruptible(struct wait_queue **p);

```

aufruft, um alle in der Warteschlange eingetragenen Prozesse zu wecken.

Mit Hilfe von Warteschlangen stellt LINUX auch Semaphore bereit. Diese dienen der Synchronisation der Zugriffe verschiedener Routinen des Kerns auf gemeinsam benutzte Datenstrukturen. Diese Semaphore sind nicht mit den für Anwenderprogramme bereitgestellten Semaphoren von UNIX SYSTEM V zu verwechseln.

```

struct semaphore {
    atomic_t count;
    wait_queue_head_t wait;
    ...
};

```

Ein Semaphor gilt als belegt, wenn `count` einen Wert kleiner oder gleich 0 hat. In die Warteschlange tragen sich alle Prozesse ein, die den Semaphor belegen wollen. Sie werden dann benachrichtigt, wenn er von einem anderen Prozess freigegeben wird. Zum Belegen oder Freigeben von Semaphoren gibt es zwei Hilfsfunktionen:

```

void down( struct semaphore * sem )
{
    while( sem -> count <= 0 )
        sleep_on( & sem->wait );
    sem -> count -= 1;
}

void up( struct semaphore * sem )
{
    sem -> count += 1;
    wake_up( & sem -> wait );
}

```

Die reale Implementation dieser Funktionen ist aus Effizienzgründen in Assemblercode ausgeführt und erheblich komplizierter.

### 3.1.6 Systemzeit und Zeitgeber (Timer)

Im LINUX-System gibt es genau eine interne Zeitbasis. Sie wird in vergangenen Ticks seit dem Starten des Systems gemessen. Ein Tick entspricht dabei 10 Millisekunden. Diese Ticks werden von einem Zeitgeberbaustein der Hardware generiert und vom Timerinterrupt (siehe Abschnitt 3.2.5) in der globalen Variablen `jiffies` gezählt. Alle im Folgenden genannten Systemzeiten beziehen sich immer auf diese Zeitbasis.

Wofür braucht man Timer? Viele Gerätetreiber möchten eine Meldung erhalten, wenn das Gerät nicht bereit ist. Andererseits muss bei der Bedienung eines langsamen Gerätes vielleicht etwas gewartet werden, ehe die nächsten Daten gesendet werden können.

Um dies zu unterstützen, bietet LINUX die Möglichkeit, Funktionen zu einem definierten zukünftigen Zeitpunkt zu starten. Dafür gibt es das Interface der Form:

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

Der Eintrag `list` in dieser Struktur dient der internen Verwaltung der Timer in einer doppelt verketteten Liste. Die Komponente `expires` gibt den Zeitpunkt an, zu dem die Funktion `function` mit dem Argument `data` aufgerufen werden soll. Die Funktionen

```
extern void add_timer(struct timer_list * timer);
extern int del_timer(struct timer_list * timer);
extern int mod_timer(struct timer_list * timer,
                    unsigned long expires);
```

dienen der Verwaltung einer globalen Timerliste. `add_timer()` aktiviert einen Timer durch Eintrag in die globale Timerliste; `del_timer()` entfernt ihn wieder, und `mod_timer()` ändert den `expire`-Zeitpunkt eines aktivierten Timers.

Der Timerinterrupt (siehe Abschnitt 3.2.5) ruft regelmäßig die Funktion

```
static inline void run_timer_list(void);
```

auf, die nach abgelaufenen Timern sucht und die zugehörigen Funktionen aufruft.

## 3.2 Zentrale Algorithmen

In diesem Kapitel werden zentrale Algorithmen der Prozessverwaltung beschrieben.

### 3.2.1 Signale

Eine der ältesten Möglichkeiten der Interprozesskommunikation unter UNIX sind Signale. Der Kern benutzt Signale, um Prozesse über bestimmte Ereignisse zu informieren.

Der Anwender benutzt Signale in der Regel, um Prozesse abubrechen oder interaktive Programme in einen definierten Zustand zu überführen. Prozesse ihrerseits benutzen Signale, um sich mit anderen Prozessen zu synchronisieren.

Normalerweise werden Signale durch die Funktion

```
int send_sig_info(int sig,
                  struct siginfo *info,
                  struct task_struct *t);
```

versandt.

Sie erhält als Argumente neben der Signalnummer `sig` und einem Pointer auf die Task, die das Signal empfangen soll, noch einen Parameter `info`, welcher den Sender identifiziert. Der Kern darf jedem Prozess ein Signal senden, ein normaler Nutzerprozess darf dies nur unter bestimmten Bedingungen. Er muss dafür entweder Superuser-Rechte besitzen oder dieselbe UID und GID wie der Empfängerprozess haben. Eine Ausnahme bildet das Signal `SIGCONT`. Dieses darf auch von einem beliebigen Prozess derselben Sitzung (*Session*) gesendet werden.

Wenn die Berechtigung zum Senden des Signals vorliegt, und der Prozess dieses Signal nicht ignorieren möchte, wird es dem Prozess geschickt. Dies geschieht, indem das Signal in der Komponente `pending` der Taskstruktur des empfangenden Prozesses eingetragen wird. Damit einfach zu überprüfen ist, ob für einen Prozess Signale vorliegen, wird ebenfalls die Komponente `sigpending` der Taskstruktur gesetzt.

Damit ist das Signal gesendet. Es erfolgt noch keine Behandlung des Signals durch den empfangenden Prozess. Dies geschieht erst, wenn der Scheduler den Prozess wieder in den Zustand `TASK_RUNNING` versetzt (vgl. Abschnitt 3.2.6).

Wenn der Prozess vom Scheduler wieder aktiviert wird, wird vor dem Umschalten in den Nutzermodus die Routine `ret_from_sys_call` (Abschnitt 3.3.1) abgearbeitet. Diese prüft, ob Signale für den aktuellen Prozess vorliegen und bearbeitet werden müssen. Dies ist der Fall, wenn das Flag `sigpending` in der Taskstruktur des Prozesses gesetzt ist. Wenn dies der Fall ist, wird die Funktion `do_signal()` aufgerufen, welche die eigentliche Signalbehandlung übernimmt. Für den Fall, dass beim Eintreffen eines Signals eine nutzerdefinierte Funktion aufgerufen werden soll, ruft `do_signal` die Funktion `handle_signal()` auf.

Offen ist noch die Frage, wie diese Funktion für den Aufruf der vom Prozess definierten Signalbehandlungsroutine sorgt. Das wurde trickreich gelöst. Die Funktion `handle_signal()` manipuliert den Stack und die Register des Prozesses. Der Instruction-Pointer des Prozesses wird auf den ersten Befehl der Signalbehandlungsroutine gesetzt. Des Weiteren wird der Stack um die Parameter der Signalbehandlungsroutine ergänzt. Wenn nun der Prozess seine Arbeit fortsetzt, sieht es für ihn so aus, als ob die Signalbehandlungsroutine wie eine normale Funktion aufgerufen wurde.

Das ist die prinzipielle Vorgehensweise, die allerdings in der realen Implementierung um zwei Punkte erweitert wird.

Zum einen erhebt LINUX den Anspruch, POSIX-kompatibel zu sein. Der Prozess kann angeben, welche Signale während der Abarbeitung einer Signalbehandlungsroutine blockiert werden sollen. Das wird dadurch realisiert, dass der Kern vor dem Aufruf der nutzerdefinierten Signalbehandlungsroutine die Signalmaske `current->blocked` um weitere Signale ergänzt. Ein Problem besteht nun darin, dass die Signalmaske nach der Abarbeitung der Signalbehandlungsroutine wieder restauriert werden muss. Deswegen wird als Return-Adresse der Signalbehandlungsroutine auf dem Stack ein Befehl eingetragen, der den Systemruf `sigreturn` aufruft. Dieser übernimmt dann die Aufräumarbeiten nach dem Beenden der nutzerdefinierten Signalbehandlungsroutine.

Die zweite Erweiterung ist eine Optimierung. Müssen mehrere Signalbehandlungsroutinen aufgerufen werden, so werden auch mehrere Stackframes angelegt. Damit werden dann die Signalbehandlungsroutinen direkt hintereinander ausgeführt.

### 3.2.2 Hardwareinterrupts

Zur Kommunikation der Hardware mit dem Betriebssystem werden Interrupts verwendet. Auf die Programmierung von Interruptroutinen werden wir in Abschnitt 7.3.2 näher eingehen. Hier interessieren wir uns mehr für den prinzipiellen Ablauf beim Aufruf eines Interrupts.

Beim Entwurf einer Interruptroutine steht der Programmierer vor einem Problem. Zum einen muss die eigentliche Interruptroutine die Hardware so schnell wie möglich bedienen und dann den Prozessor für andere Aufgaben, zum Beispiel für die Bearbeitung weiterer Interrupts, wieder freigeben. Zum anderen kann so ein Interrupt aber auch die Verarbeitung einer größeren Datenmenge auslösen.

Um dieses Problem zu lösen, erfolgt die Interruptbehandlung unter Linux in zwei Phasen. Zuerst wird die zeitkritische Kommunikation mit der Hardware durchgeführt, hierbei sind eventuell andere Interrupts gesperrt. Die eigentliche Verarbeitung der Daten erfolgt asynchron zum eigentlichen Interrupt. Dafür werden entweder „Softwareinterrupts“, „Tasklets“ oder „Bottom-Halbs“ verwendet. Dies sind Funktionen, die zu einem späteren Zeitpunkt aufgerufen werden. Sie können ihrerseits auch wieder von anderen Interrupts unterbrochen werden.

Die zentrale Behandlungsroutine für Hardwareinterrupts sieht vereinfacht so aus:

```
unsigned int do_IRQ(struct pt_regs regs) {
    int irq;
    struct irqaction * action;

    /* irq nummer aus den registern entnehmen */
    irq = regs.orig_eax & 0xff;

    /* entsprechende handler finden */
    action = irq_desc[irq].action;

    /* und die Aktionen ausführen */
```

```

while ( action )
{
    action->handler(irq, regs)
    action = action->next;
}

/* hier ist der eigentliche Hardwareinterrupt beendet. */

if( softirq_active & softirq_mask)
    do_softirq();
}

```

### 3.2.3 Softwareinterrupts

LINUX 2.4 führt das Konzept der Softwareinterrupts ein. Ein Softwareinterrupt ist wie ein Hardwareinterrupt ein Ereignis, das ausgelöst werden kann und das zum Abarbeiten von Interruptbehandlungsroutinen führt. Nur werden diese nicht wie Hardwareinterrupts sofort gestartet, sondern nur zu bestimmten Zeitpunkten. Konkret passiert dies nach jedem Hardwareinterrupt und nach jedem Systemcall.

Wie auch bei Hardwareinterrupts ist die Anzahl von Softwareinterrupts begrenzt

```

enum { HI_SOFTIRQ=0,
        NET_TX_SOFTIRQ, NET_RX_SOFTIRQ,
        TASKLET_SOFTIRQ
};
static struct softirq_action softirq_vec[32];

```

HI\_SOFTIRQ ist dabei der Softwareinterrupt mit der höchsten Priorität, NET\_TX\_SOFTIRQ und NET\_RX\_SOFTIRQ werden vom Netzwerkcode benutzt, und TASKLET\_SOFTIRQ ist der Softwareinterrupt, über den die Abarbeitung der Tasklets realisiert wird.

Die Registrierung eines Interrupthandler erfolgt über die Funktion `open_softirq()`. Ein Aufruf von `raise_softirq()` sorgt dafür, dass die registrierte Behandlungsroutine beim nächsten Aufruf von `do_softirq()` ausgeführt wird.

```

void open_softirq(int nr,
                  void (*action)(struct softirq_action*),
                  void *data);
raise_softirq(int nr);
void do_softirq();

```

Zu beachten ist, dass in einem Multiprozessorsystem durchaus derselbe Interrupthandler gleichzeitig auf verschiedenen Prozessoren abgearbeitet werden kann, die Funktionen müssen also reentrant sein oder explizit eine Serialisierung beim Zugriff auf gemeinsame Ressourcen implementieren.

Etwas einfacher als Softwareinterrupts kann mit Tasklets gearbeitet werden. Hier wird vom System garantiert, dass ein bestimmtes Tasklet zu einem Zeitpunkt immer nur einmal ausgeführt wird, verschiedene Tasklets können allerdings auch parallel abgearbeitet werden.

Die Registrierung eines Tasklets erfolgt durch die Funktion `tasklet_init()`. Mit `tasklet_schedule()` wird ein Tasklet zur Abarbeitung markiert und der Softwareinterrupt `TASKLET_SOFTIRQ` ausgelöst. Dessen Interruptroutine arbeitet dann die Tasklets ab.

```
struct tasklet_struct;

void tasklet_init(struct tasklet_struct *t,
                 void (*func)(unsigned long),
                 unsigned long data);

void tasklet_schedule(struct tasklet_struct *t);
```

Softwareinterrupts und Tasklets sind neu in LINUX 2.4, sehr lange gibt es dagegen schon die Bottom-Halbs. Diese waren bisher ähnlich den Softwareinterrupts implementiert, inzwischen setzen sie dagegen auf die Tasklets auf. Die Bottom-Half-Funktionalität sollte in Neuentwicklungen nicht mehr benutzt werden, deswegen wird sie hier auch nicht mehr detailliert beschrieben. Der wesentliche Unterschied zu den Tasklets besteht darin, dass auch auf einem Multiprozessorsystem zu einem Zeitpunkt immer nur ein Bottom-Half-Handler abgearbeitet wird.

### 3.2.4 Booten des Systems

Das Booten eines UNIX-Systems (eigentlich jedes Betriebssystems) hat etwas Magisches an sich. Das soll in diesem Abschnitt etwas transparenter gemacht werden.

In Anhang D wird erklärt, wie LILO (der *L*inux *L*Oader) den LINUX-Kern findet und in den Speicher lädt. Er beginnt dann seine Arbeit am Eintrittspunkt `start:`, der sich in der Datei `arch/i386/boot/setup.S` befindet. Wie der Name schon sagt, handelt es sich hierbei um Assemblercode, welcher die Initialisierung der Hardware übernimmt. Nachdem wichtige Hardwareparameter ermittelt wurden, erfolgt die Umschaltung des Prozessors in den *Protected Mode* durch Setzen des Protected-Mode-Bit im *Maschinen Status Wort*.

Anschließend bewirkt der Assemblerbefehl

```
jmp 0x100000 , __KERNEL_CS
```

den Sprung zur Startadresse des 32-Bit-Codes des eigentlichen Betriebssystemkerns, und es geht bei `startup_32:` in der Datei `arch/i386/kernel/head.S` weiter. Hier werden weitere Teile der Hardware initialisiert (insbesondere die MMU (Page-Tabelle), der Coprozessor und die Interruptdeskriptortabelle) sowie die Umgebung (Stack, Environment usw.), welche benötigt wird, um die C-Funktionen des Kerns auszuführen. Nach

der vollständigen Initialisierung wird dann die erste C-Funktion, `start_kernel()`, aus `init/main.c` aufgerufen.

Hier erfolgt zunächst in der Funktion `setup_arch()` die Sicherung aller Daten, die der bisherige Assemblercode über die Hardware ermittelt hat sowie die Initialisierung weiterer architekturabhängiger Komponenten. Anschließend werden die hardwareunabhängigen Teile des Kerns initialisiert.

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;

    printk(linux_banner);
    setup_arch(&command_line);
    parse_options(command_line);

    trap_init();
    init_IRQ();
    sched_init();
    time_init();
    softirq_init();

    console_init();

    init_modules();
    ...
}
```

Der jetzt laufende Prozess ist der Prozess 0. Dieser erzeugt dann einen Kernel-Thread, welcher die Funktion `init()` ausführt.

```
kernel_thread(init, NULL, ...)
```

Der Prozess 0 ist anschließend nur noch damit beschäftigt, nicht benötigte Rechenzeit zu verbrauchen.

```
cpu_idle(NULL);
```

Die Funktion `init()` erledigt die restliche Initialisierung. Unter anderem werden hier von der Funktion `do_basic_setup()` sämtliche Treiber für die Hardware initialisiert.

```
static int init()
{
    do_basic_setup();
}
```

Jetzt kann versucht werden, eine Verbindung zur Konsole herzustellen und die Dateide-skriptoren 0, 1 und 2 zu eröffnen.

```
if (open("/dev/console", O_RDWR, 0) < 0)
    printk("Warning: unable to open an initial console.\n");

(void) dup(0);
(void) dup(0);
```

Danach wird versucht, ein vom User beim Booten spezifiziertes Programm oder eines der Programme `/sbin/init`, `/etc/init` oder `/bin/init` auszuführen. Diese starten dann normalerweise die unter LINUX laufenden Hintergrundprozesse und sorgen dafür, dass auf jedem angeschlossenen Terminal das Programm `getty` läuft — ein Nutzer kann sich beim System anmelden.

```
if (execute_command)
    execve(execute_command,argv_init,envp_init);

execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
```

Für den Fall, dass keines der oben genannten Programme existiert, wird versucht, eine Shell zu starten, so dass der Superuser das System reparieren kann. Wenn dies auch nicht möglich ist, wird das System angehalten.

```
execve("/bin/sh",argv_init,envp_init);
panic("No init found..");
```

Die hier beschriebene Vorgehensweise sollte nur einen Überblick über die beim Starten des Systems ablaufenden Aktionen geben. Die Realität ist, bedingt durch Hardwareinitialisierung (MMU, SMP) und die Behandlung von Sonderfällen, wie zum Beispiel dem Benutzen einer Initialen Ramdisk (INITRD), komplizierter.

### 3.2.5 Timerinterrupt

Jedes Betriebssystem braucht eine Zeitmessung und eine Systemzeit. Realisiert wird die Systemzeit in der Regel dadurch, dass die Hardware in bestimmten Abständen einen Interrupt auslöst. Die so angestoßene Interruptroutine übernimmt das „Zählen“ der Zeit. Die Systemzeit wird unter LINUX in Ticks seit dem Start des Systems gemessen. Ein Tick entspricht 10 Millisekunden, der Timerinterrupt wird also 100-mal in der Sekunde ausgelöst. Die Zeit wird in der Variablen

```
unsigned long volatile jiffies;
```

gespeichert, welche nur vom Timerinterrupt modifiziert werden darf. Dieser Mechanismus stellt jedoch nur die interne Zeitbasis zur Verfügung.

Anwendungen interessieren sich aber bevorzugt für die „reale Zeit“. Diese wird in der Variablen

```
volatile struct timeval xtime;
```

mitgeführt und ebenfalls vom Timerinterrupt aktualisiert.

Der Timerinterrupt wird relativ häufig aufgerufen und ist deswegen etwas zeitkritisch. Deswegen ist auch hier die Implementierung zweigeteilt.

Die eigentliche Interruptroutine `do_timer()` aktualisiert nur die Variable `jiffies` und kennzeichnet die *Bottom-Half*-Routine (vgl. die Abschnitte 3.2.2 und 7.3.5) des Timerinterrupts als aktiv. Diese wird vom System zu einem späteren Zeitpunkt aufgerufen und erledigt den Rest der Arbeit.

```
void do_timer(struct pt_regs * regs)
{
    (*(unsigned long *)&jiffies)++;

    update_process_times(user_mode(regs));

    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

`update_process_times()` wird weiter unten beschrieben. Wir wollen uns aber zuerst die *Bottom-Half*-Routine des Timerinterrupts anschauen.

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

`run_timer_list()` sorgt dabei für das Abarbeiten der im Abschnitt 3.1.6 beschriebenen Funktionen zur Aktualisierung systemweiter Timer. Darunter fallen auch die Real-Zeit-Timer der aktuellen Task. `update_times()` ist für das Aktualisieren der Zeiten verantwortlich.

```
static inline void update_times(void)
{
    unsigned long ticks;

    ticks = jiffies - wall_jiffies;

    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

`update_wall_time()` widmet sich nun dem Aktualisieren der *realen Zeit* `xtime` und wird aufgerufen, wenn seit dem letztem Aufruf dieser Funktion Zeit vergangen ist.

Die Funktion `update_process_time` sammelt die Daten für den Scheduler und entscheidet, ob dieser aufgerufen werden muss.

```
static void update_process_times(int user_ticks);
{
```

```

struct task_struct * p = current;
int cpu = smp_processor_id();

unsigned long user = ticks - system;

```

Zuerst wird die Komponente `counter` der Task-Struktur aktualisiert. Wenn `counter` gleich Null wird, ist die Zeitscheibe für den aktuellen Prozess abgelaufen, und der Scheduler wird bei der nächsten Gelegenheit aktiviert.

```

update_one_process(p, ticks, user, system, 0);
if(p->pid)
{
    p->counter -= 1;
    if (p->counter <= 0) {
        p->counter = 0;
        p->need_resched = 1;
    }
}

```

Danach werden für Statistikzwecke die Komponenten `per_cpu_user` Task-Struktur aktualisiert.

```

    p->per_cpu_user[cpu] += user_ticks;
}

```

Unter LINUX ist es möglich, die Ressource „CPU-Verbrauch“ eines Prozesses zu beschränken. Das geschieht durch den Systemruf `setrlimit`, mit welchem auch andere Ressourcen eines Prozesses beschränkt werden können. Das Überschreiten des Zeitlimits wird im Timerinterrupt geprüft, und der Prozess wird durch Senden des Signals SIGXCPU informiert bzw. durch das Signal SIGKILL abgebrochen. Anschließend müssen noch die Intervalltimer für die laufende Task aktualisiert werden. Wenn diese abgelaufen sind, wird die Task durch ein entsprechendes Signal informiert.

```

void update_one_process(p,user,system,cpu)
{
    p->per_cpu_utime[cpu] += user;
    p->per_cpu_stime[cpu] += system;
    do_process_times(p, user, system);

    do_it_virt(p, user);
    do_it_prof(p);
}

void do_process_times(p,user,system)
{
    psecs = (p->times.tms_utime += user);
    psecs += (p->times.tms_stime += system);
    if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_cur) {
        /* Send SIGXCPU every second.. */
        if (!(psecs % HZ))
            send_sig(SIGXCPU, p, 1);
        /* and SIGKILL when we go over max.. */
    }
}

```

```
        if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_max)
            send_sig(SIGKILL, p, 1);
    }
}

void do_it_virt(p, user) {
    unsigned long it_virt = p->it_virt_value;

    if (it_virt)
    {
        it_virt -= user;
        if (it_virt <= 0)
        {
            it_virt = p->it_virt_incr;
            send_sig(SIGVTALRM, p, 1);
        }
        p->it_virt_value = it_virt - user;
    }
}
```

### 3.2.6 Scheduler

Der Scheduler ist für die Zuteilung der Ressource „Prozessor“ (also der Rechenzeit) an die einzelnen Prozesse verantwortlich. Nach welchen Kriterien das geschieht, ist von Betriebssystem zu Betriebssystem verschieden. UNIX-Systeme bevorzugen traditionell interaktive Prozesse, um kurze Antwortzeiten zu ermöglichen und dem Benutzer dadurch das System subjektiv schneller erscheinen zu lassen. Linux unterstützt, dem Posix-Standard 1003.4 folgend, verschiedene Schedulingklassen, welche mit dem Systemruf `sched_setscheduler()` ausgewählt werden können.

Zum einen gibt es Echtzeitprozesse in den Schedulingklassen `SCHED_FIFO` und `SCHED_RR`. Echtzeit heißt hier nicht „harte Echtzeit“ mit garantierten Prozessumschalt- und Reaktionszeiten sondern eine „weiche Echtzeit“. Wenn ein Prozess mit höherer Echtzeitpriorität (beschrieben in der Komponente `rt_priority` der Task-Struktur) laufen möchte, so werden alle Prozesse mit niedrigerer Echtzeitpriorität verdrängt.

Der Unterschied zwischen `SCHED_FIFO` und `SCHED_RR` besteht darin, dass ein Prozess der Klasse `SCHED_FIFO` so lange laufen kann, bis er die Steuerung freiwillig abgibt oder ein Prozess mit höherer Echtzeitpriorität laufen möchte. Im Gegensatz dazu wird ein Prozess in der Klasse `PROG_RR` auch unterbrochen, wenn seine Zeitscheibe abgelaufen ist und es Prozesse mit derselben Echtzeitpriorität gibt. Dadurch wird unter den Prozessen derselben Priorität ein klassisches *Round-Robin*-Verfahren realisiert.

Zum anderen gibt es die Schedulingklasse `SCHED_OTHER`, die einen klassischen UNIX-Schedulingalgorithmus implementiert. Laut Posix 1003.4 hat jeder Echtzeitprozess eine höhere Priorität als ein Prozess der Schedulingklasse `SCHED_OTHER`.

Der Scheduling-Algorithmus von LINUX ist in der Funktion `schedule()` (`kernel/sched.c`) implementiert. Sie wird an zwei verschiedenen Stellen aufgerufen. Zum einen

gibt es Systemrufe, die die Funktion `schedule()` aufrufen, in der Regel indirekt durch den Aufruf von `sleep_on()` (vgl. Abschnitt 3.1.5). Zum anderen wird nach jedem Systemruf und nach jedem Interrupt von der Routine `ret_from_sys_call()` das Flag `need_resched` in der aktuellen Taskstruktur geprüft. Wenn es gesetzt ist, wird der Scheduler ebenfalls aufgerufen. Da zumindest der Timerinterrupt regelmäßig aufgerufen wird und dabei bei Bedarf auch das Flag `need_resched` setzt, wird der Scheduler regelmäßig aktiviert.

Die Funktion `schedule()` besteht aus drei Teilen. Zuerst werden alle anstehenden Softwareinterrupts abgearbeitet. Danach wird der Prozess mit der höchsten Priorität bestimmt. Dabei haben *Real-Time*-Prozesse immer Vorrang vor „normalen“. Als drittes wird der neue Prozess zum aktuellen Prozess, und der Scheduler hat seine Arbeit erledigt.

Leider ist der eigentliche Quellcode des Schedulers seit der Kernelversion 2.0 relativ unübersichtlich geworden. Zum Teil liegt das an den aus Effizienzgründen erfolgten Umstrukturierungen in der Version 2.0, zum wesentlichen Teil aber auch an dem neu vorhandenen Multiprozessor-Support.

Deswegen werden wir hier eine stark vereinfachte Version der Funktion `schedule()` vorstellen. Unter anderem wurde dabei auf die für den SMP-Support benötigten Details verzichtet.

```
asmlinkage void schedule(void)
{
    struct task_struct * prev, * next, *p;

    prev = current;
    prev->need_resched = 0
```

Zuerst werden die Softwareinterrupts aufgerufen. Dabei handelt es sich um zeitkritische Routinen, die aus Effizienzgründen aus den Interrupthandlern ausgelagert worden sind. Da diese aber eventuell Informationen manipulieren, die das Scheduling beeinflussen (zum Beispiel kann durch so eine Routine eine Task wieder in den Zustand `TASK_RUNNING` gebracht werden), müssen sie spätestens hier abgearbeitet werden.

```
    if (softirq_active(this_cpu) & softirq_mask(this_cpu))
        do_softirq();
```

Falls `schedule()` aufgerufen wurde, weil der aktuelle Prozess auf ein Ereignis warten muss, so wird er aus der Run-Queue entfernt. Falls die aktuelle Task der Schedulingklasse `SCHED_RR` angehört und das Zeitfenster für diese Task abgelaufen ist, wird sie in der Run-Queue an letzter Stelle und damit hinter allen anderen lauffähigen Tasks der Schedulingklasse `SCHED_RR` gestellt.

Die Run-Queue ist eine durch die Komponente `run_list` der Task-Struktur doppelt verkettete Liste der Prozesse, die sich um den Prozessor bewerben.

```
    if (!prev->counter && prev->policy == SCHED_RR)
    {
```

```

    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
if( prev->state != TASK_RUNNING )
{
    del_from_runqueue(prev);
}

```

Als nächstes wird der eigentliche Schedulingalgorithmus durchgeführt, das heißt, es wird der Prozess in der Run-Queue gesucht, der die höchste Priorität hat. Die Funktion `goodness()` berechnet dabei für jeden Prozess seine Priorität; Realtime Prozesse werden bevorzugt.

```

next = idle_task;          /* nächster Prozess */
next_p = -1000;          /* und dessen Prioritaet */

list_for_each(p,&runqueue_head)
{
    if( ! can_schedule(p) )
        continue;
    weight = goodness(p,prev,this_cpu);
    if( weight > next_p)
    {
        next_p = weight; next = p;
    }
}

```

Wenn jetzt `next_p` größer 0 ist, so haben wir einen geeigneten Kandidaten gefunden. Ist `next_p` kleiner 0, so gibt es keinen laufbereiten Prozess, und wir müssen die Idle-Task aktivieren. In beiden Fällen zeigt `next` auf die nächste zu aktivierende Task. Wenn jedoch `next_p` gleich 0 ist, so gibt es zwar lafbereite Prozesse, wir müssen aber deren dynamische Prioritäten (den Wert von `counter`) neu berechnen. Hierbei werden dann auch noch die `counter`-Werte aller anderen Prozesse neu berechnet. Danach sollten wir den Scheduler nochmals, diesmal erfolgreich, starten.

```

if( next_p == 0 )
{
    for_each_task(p)
    {
        p->counter = (p->counter / 2) + p->priority;
    }
}

```

Jetzt wird die Task, auf die `next` zeigt, als nächste aktiviert.

```

if( prev != next )
    switch_to(prev,next);
} /* schedule() */

```

Damit ist die Beschreibung des Schedulers abgeschlossen. Es sei nochmals darauf verwiesen, dass es sich bei diesem Quelltext um eine sehr vereinfachte Version des Schedulers

handelt, die jedoch unserer Meinung nach vollständig genug ist, um die Arbeitsweise des Schedulers zu verstehen.

## 3.3 Implementierung von Systemrufen

Die Funktionalität des Betriebssystems wird den Prozessen mit Hilfe von Systemrufen zur Verfügung gestellt. In diesem Kapitel wollen wir uns mit der Implementierung von Systemrufen unter LINUX beschäftigen.

### 3.3.1 Wie funktionieren Systemrufe eigentlich?

Ein Systemruf setzt einen definierten Übergang vom Nutzermodus in den Systemmodus voraus. Dies ist in LINUX nur durch Interrupts möglich. Für die Systemrufe wurde deshalb der Interrupt 0x80 reserviert.<sup>4</sup>

Normalerweise ruft man als Nutzer immer eine Bibliotheksfunktion (wie etwa `fork()`) auf, um bestimmte Aufgaben auszuführen. Diese Bibliotheksfunktion (in der Regel aus den `_syscall`-Makros in `<asm/unistd.h>` generiert) schreibt ihre Argumente und die Nummer des Systemrufs in definierte Übergaberegister und löst anschließend den Interrupt 0x80 aus. Wenn die zugehörige Interruptserviceroutine zurückkehrt, wird der Rückgabewert aus dem entsprechenden Übergaberegister gelesen, und die Bibliotheksfunktion ist beendet.

Die eigentliche Arbeit der Systemrufe wird von der Interruptroutine erledigt. Diese beginnt bei der Einsprungsadresse `system_call()`, die in der Datei `arch/i386/kernel/entry.S` zu finden ist.

Leider ist diese Routine vollständig in Assembler geschrieben. Hier wird der besseren Lesbarkeit halber ein C-Äquivalent dieser Funktion beschrieben. Wo immer im Assemblertext symbolische Marken vorkommen, haben wir sie auch in den C-Text als Marken übernommen.

`sys_call_num` und `sys_call_args` entsprechen der Nummer des Systemrufs (vgl. `<asm/unistd.h>`) und dessen Argumenten.

```
PSEUDO_CODE system_call( int sys_call_num , sys_call_args )
{
system_call:
```

Zuerst werden alle Register des Prozesses gerettet.

```
    SAVE_ALL; /* Makro aus entry.S */
```

Wenn `sys_call_num` einen legalen Wert repräsentiert, wird die der Nummer des Systemrufs zugeordnete Behandlungsroutine aufgerufen. Diese ist im Feld `sys_call_`

<sup>4</sup> Das gilt für die LINUX-Systemrufe auf dem PC. Schon die von LINUX auf dem PC unterstützte iBCS-Emulation verwendet ein anderes Verfahren — das sogenannte `lcall17`-Gate.

table[] (in der Datei arch/i386/kernel/entry.S definiert) eingetragen. Falls für den Prozess das Flag PF\_TRACESYS gesetzt ist, wird dieser von seinem Elternprozess überwacht. Die dazu notwendigen Arbeiten erledigt die Funktion syscall\_trace() (arch/i386/kernel/ptrace.c). Sie ändert den Zustand des aktuellen Prozesses auf TASK\_STOPPED, sendet dem Elternprozess das Signal SIGTRAP und ruft den Scheduler auf. Der aktuelle Prozess wird unterbrochen, bis der Elternprozess ihn wieder aktiviert. Da das vor und nach jedem Systemruf erfolgt, hat der Elternprozess vollständige Kontrolle über das Verhalten des Kindprozesses.

```

        if (sys_call_num >= NR_syscalls)
        {
badsys:
            errno = -ENOSYS;
        }
        else
        {
            if (current->ptrace)
                syscall_trace();

            errno=(*sys_call_table[sys_call_num])(sys_call_args);

            if (current->ptrace)
                syscall_trace();
        }

```

Die eigentliche Arbeit des Systemrufs ist jetzt erledigt. Bevor der Prozess weiterarbeiten kann, gibt es aber eventuell noch einige administrative Aufgaben zu erledigen.

```

ret_from_sys_call:
    if (softirq_active & softirq_mask)
handle_softirq:
    do_softirq()

```

Falls ein Scheduling angefordert wurde (need\_resched != 0), wird der Scheduler aufgerufen. Dadurch wird ein anderer Prozess aktiv. schedule() kehrt erst wieder zurück, wenn der Prozess vom Scheduler neu aktiviert wurde.

```

ret_with_reschedule:
    if (current->need_resched) {
reschedule:
        schedule();
        goto ret_from_sys_call;
    }

```

Falls für den aktuellen Prozess Signale gesendet wurden, und der Prozess ihre Annahme nicht blockiert hat, werden sie jetzt abgearbeitet. Die Funktion do\_signal() ist in Abschnitt 3.2.1 näher beschrieben.

```

        if (current->sigpending)
signal_return:
        do_signal();

```

Damit ist alles erledigt, und der Systemruf kehrt zurück. Dazu werden zuerst alle Register restauriert, und anschließend wird mit dem Assemblerbefehl `iret` die Interrupt-routine beendet.

```
restore_all:
    RESTORE_ALL;
} /* PSEUDO_CODE system_call */
```

### 3.3.2 Beispiele für einfache Systemrufe

Im Folgenden wollen wir uns die Implementierung einiger Systemrufe genauer ansehen. Dabei soll die Benutzung der in diesem Kapitel erläuterten Algorithmen und Datenstrukturen demonstriert werden.

#### Der Systemruf `getuid`

`getuid` ist ein sehr einfacher Systemruf – er liest lediglich einen Wert aus der Taskstruktur und liefert diesen zurück:

```
asmlinkage int sys_getuid(void)
{
    return current->uid;
}
```

#### Der Systemruf `nice`

Der Systemruf `nice` ist etwas komplizierter. `nice` erwartet als Argument eine Zahl, um die die statische Priorität des aktuellen Prozesses modifiziert werden soll.

Alle Systemrufe, die Argumente verarbeiten, müssen diese auf ihre Plausibilität hin überprüfen.

```
asmlinkage int sys_nice(int increment)
{
    int newpriority;
```

Man beachte, dass ein größeres Argument von `sys_nice()` eine geringere Priorität bedeutet. Deswegen ist der Name `increment` für das Argument von `nice` etwas irreführend.

```
    if (increment < 0 && !capable(CAP_SYS_NICE))
        return -EPERM;
```

`capable()` überprüft, ob der aktuelle Prozess die Berechtigung hat, seine Priorität zu erhöhen. In klassischen Unix-Systemen ist dies der Fall, wenn der Prozess Superuserrechte besitzt. Linux stellt mittlerweile ein Konzept zur Verfügung, dieses Superuserrecht feiner zu unterteilen.

Jetzt kann die neue Priorität des Prozesses ermittelt werden. Dabei wird unter anderem sichergestellt, dass sich die neue Priorität des Prozesses in einem sinnvollen Bereich bewegt.

```
newpriority = ...

if (newpriority < -20)
    newpriority = -20;
if (newpriority > 19)
    newpriority = 19;

current->nice = newpriority;
return 0;
} /* sys_nice */
```

## Der Systemruf *pause*

*pause* unterbricht die Programmausführung so lange, bis der Prozess durch ein Signal wieder aktiviert wird. Dazu wird lediglich der Status des aktuellen Prozesses auf `TASK_INTERRUPTIBLE` gesetzt und anschließend der Scheduler aufgerufen. Dadurch wird eine andere Task aktiv.

Der Prozess kann nur wieder aktiv werden, wenn der Status des Prozesses wieder auf `TASK_RUNNING` gesetzt wird. Dies geschieht beim Eintreffen eines Signals (vgl. Abschnitt 3.2.6). Danach kehrt der Systemruf *pause* mit dem Fehler `ERESTARTNOHAND` zurück und führt (wie in Abschnitt 3.2.1 beschrieben) die notwendigen Aktionen zur Signalbehandlung durch.

```
asmlinkage int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    return -ERESTARTNOHAND;
}
```

### 3.3.3 Beispiele für komplexere Systemrufe

Jetzt wollen wir uns etwas komplexeren Systemrufen zuwenden. In diesem Abschnitt werden wir die Systemrufe zur Prozessverwaltung (*fork*, *execve*, *\_exit* und *wait*) beschreiben.

#### Der Systemruf *fork*

Der Systemruf *fork* ist die einzige Möglichkeit, einen neuen Prozess zu starten. Das geschieht, indem eine (fast) identische Kopie des Prozesses erzeugt wird, welcher *fork* aufgerufen hat.

*fork* ist eigentlich ein recht aufwendiger Systemruf. Es müssen alle Daten des Prozesses kopiert werden. Das können durchaus einige Megabyte sein. Im Laufe der Entwicklung von UNIX wurden verschiedene Wege eingeschlagen, um den Aufwand für *fork* so gering wie möglich zu halten. In dem häufig vorkommenden Fall, dass nach *fork* direkt *exec* aufgerufen wird, ist das Kopieren der Daten nicht notwendig. Sie werden nicht benötigt. In den UNIX-Systemen der BSD-Reihe wurde deswegen der Systemruf *vfork* etabliert. Er erzeugt wie *fork* einen neuen Prozess, teilt das Datensegment aber zwischen beiden Prozessen auf. Dies ist eine recht unsaubere Lösung, da ein Prozess die Daten des anderen Prozesses beeinflussen kann. Um diese Beeinflussung so gering wie möglich zu halten, wird die weitere Ausführung des Elternprozesses so lange gestoppt, bis der Kindprozess entweder durch *\_exit* beendet wurde oder durch *exec* ein neues Programm gestartet hat.

Neuere UNIX-Systeme wie zum Beispiel LINUX schlagen einen anderen Weg ein. Sie benutzen die *Copy-On-Write*-Technik. Hintergedanke dabei ist, dass mehrere Prozesse durchaus gleichzeitig auf denselben Speicher zugreifen dürfen — solange sie die Daten nicht modifizieren.

Die Speicherseiten werden also unter LINUX beim Systemruf *fork* nicht kopiert, sondern vom alten und vom neuen Prozess gleichberechtigt benutzt. Allerdings werden die von beiden Prozessen benutzten Speicherseiten als schreibgeschützt markiert — sie können von den Prozessen also nicht modifiziert werden. Wenn nun einer der Prozesse eine Schreiboperation auf diesen Speicherseiten ausführen will, wird von der Speicherwartungshardware (MMU) ein Seitenfehler (*page fault*) ausgelöst, der Prozess unterbrochen und der Kern benachrichtigt. Erst jetzt kopiert der Kern die mehrfach benutzte Speicherseite und teilt dem schreibenden Prozess seine eigene Kopie zu. Dieses Verfahren erfolgt vollständig transparent — d.h. die Prozesse merken nichts davon. Der große Vorteil dieses Copy-On-Write-Verfahrens besteht darin, dass aufwendiges Kopieren von Speicherseiten nur bei Bedarf stattfindet.

Neuere Betriebssystemkonzepte kennen außer dem Begriff des Prozesses noch den Begriff des Threads. Darunter versteht man einen unabhängigen „Faden“ im Programmablauf eines Prozesses. Mehrere Threads können innerhalb eines Prozesses parallel und unabhängig voneinander abgearbeitet werden. Der wesentliche Unterschied zum Konzept eines Prozesses besteht darin, dass die verschiedenen Threads innerhalb eines Prozesses auf demselben Speicher operieren und sich damit gegenseitig beeinflussen können. Es gibt verschiedene Konzepte zur Implementierung von Threads. Einfache Varianten, wie die weitverbreitete Pthread-Bibliothek, kommen ohne Unterstützung durch den Betriebssystemkern aus. Nachteile dieser Konzeptionen sind, dass das Scheduling der einzelnen Threads vom Nutzerprogramm vorgenommen werden muss. Für den Betriebssystemkern handelt es sich um einen normalen Prozess. Das führt dazu, dass ein blockierender Systemruf (z.B. ein *read* vom Terminal) den ganzen Prozess und damit alle Threads blockiert. Ideal wäre es aber, wenn nur der Thread, der den Systemruf verwendet hat, blockiert. Dies setzt aber eine Unterstützung des Thread-Konzepts durch den Kern voraus. Neuere UNIX-Versionen (z.B. Solaris 2.x) bieten diese Unterstützung.

LINUX unterstützt Threads durch die Bereitstellung des (linuxspezifischen) Systemrufs *clone*, welcher die nötige Kernelunterstützung zur Implementierung von Threads liefert.

*clone* arbeitet ähnlich wie *fork*, erzeugt also eine neue Task. Der wesentliche Unterschied zu *fork* besteht darin, dass nach dem Systemruf *clone* beide Tasks auf gemeinsamem Speicher arbeiten können.

Da *fork* und *clone* im Wesentlichen dasselbe tun, werden sie auch durch eine gemeinsame Funktion realisiert. Diese wird je nach Systemruf nur auf unterschiedliche Weise aufgerufen.

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs,0);
}
```

```
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs,0);
}
```

Die eigentliche Arbeit erledigt die Funktion `do_fork()`:

```
int do_fork(unsigned long clone_flags,
            unsigned long start_stack,
            struct pt_regs *regs,
            unsigned long stack_size)
{
    int error = -ENOMEM;
    unsigned long new_stack;
    struct task_struct *p;
```

Als Erstes wird der notwendige Speicherplatz für die neue Taskstruktur und den Kernel-Stack alloziert.

```
    p = alloc_task_struct();
    if (!p)
        goto fork_out;
```

Falls der Nutzer sein Limit an Prozessen überschritten hat, wird die Funktion abgebrochen. Dasselbe passiert, wenn es im System bereits zu viele Tasks gibt.

```
    if (current->user->count >=
        current->rlim[RLIMIT_NPROC].rlim_cur)
        goto bad_fork_free;
    if (nr_threads >= max_threads)
        goto bad_fork_cleanup_count;
```

Der Kindprozess `p` erbt alle Einträge des Elternprozesses.

```
*p = *current;
```

Einige der Einträge müssen für einen neuen Prozess jedoch auch neu initialisiert werden.

```
p->state = TASK_UNINTERRUPTIBLE;
p->did_exec = 0;
p->swappable = 0;
p->pid = get_pid(clone_flags);
...
p->run_list.next = NULL;
p->run_list.prev = NULL;
...
p->start_time = jiffies;
...
```

Jetzt werden die Unterstrukturen der Taskstruktur kopiert. Je nach Wert der `clone_flags` werden hier Datenstrukturen entweder kopiert oder gemeinsam benutzt. Damit werden die Unterschiede zwischen den Systemrufen *fork* und *clone* realisiert.

```
if (copy_files(clone_flags, p))
    goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))
    goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p))
    goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))
    goto bad_fork_cleanup_sighand;
copy_thread(0, clone_flags, start_stack, stack_size, p, regs);
```

Abschließend wird der Zustand der neuen Task auf `TASK_RUNNING` gesetzt, damit er vom Scheduler aktiviert werden kann. Die alte Task (der Elternprozess) kehrt vom Systemruf mit der Prozessidentifikationsnummer (PID) des neuen Prozesses zurück.

```
++nr_threads;
wake_up_process(p);
return p->pid;
```

Wenn irgend etwas schiefgelaufen sein sollte, so müssen bis dahin angeforderte Datenstrukturen wieder freigegeben werden.

```
bad_fork:
...
return error;
}
```

Die oben aufgerufene Funktion `copy_thread()` ist auch für die Initialisierung der Register des neuen Prozesses verantwortlich. Unter anderem wird der Befehlszeiger `p->tss.eip` auf die Assembleroutine `ret_from_sys_call()` gelegt, so dass der neue Prozess seine Abarbeitung so beginnt, als ob auch er den Systemruf *fork* ausgeführt hätte. Gleichzeitig wird der Rückgabewert auf Null gesetzt, damit das Programm

Eltern- und Kindprozess anhand des unterschiedlichen Rückgabewertes unterscheiden kann.

## Der Systemruf `execve`

Der Systemruf `execve` erlaubt es einem Prozess, sein ausführendes Programm zu wechseln. LINUX erlaubt mehrere Formate für ausführbare Dateien. Sie werden, wie in UNIX üblich, an den sogenannten „Magic-Numbers“ — den ersten Bytes einer ausführbaren Datei erkannt. Traditionell verwendet jedes UNIX-System sein eigenes Format für ausführbare Dateien. In den letzten Jahren hat sich ein Standard herausgebildet — das ELF-Format.<sup>5</sup> Dieses hat sich durchgesetzt, da sich hier die Behandlung von dynamischen Bibliotheken drastisch vereinfacht. Weitere Informationen zum ELF-Format findet der interessierte Leser in [ELF].

Des Weiteren unterstützt LINUX die aus der BSD-Welt stammenden Scriptdateien. Wenn eine Datei mit den beiden Zeichen „#!“ beginnt, wird sie nicht direkt geladen, sondern einem in der ersten Zeile der Datei spezifizierten Interpreterprogramm zur Bearbeitung übergeben. Die bekannte Version davon ist eine Zeile der Form

```
#!/bin/sh
```

am Anfang von Shell-Skripten. Ein Ausführen dieser Datei (d.h. ein `execve`) ist äquivalent zum Ausführen der Datei `/bin/sh` mit der ursprünglichen Datei als Argument. Hier folgt nun der kommentierte und stark gekürzte Quelltext von `do_execve()`.

```
static int do_execve(char *filename, char **argv, char **envp,
                    struct pt_regs * regs)
{
```

Zuerst wird versucht, aus dem Namen des auszuführenden Programms die zugehörige Datei (ihre Inode) zu finden. Die Struktur `bprm` wird benutzt, um alle Informationen über die Datei zu speichern.

```
    struct linux_binprm bprm;
    struct file *file;

    file = open_exec(filename);

    if ( IS_ERR(file))
        return PTR_ERR(file);

    bprm.file = file;
    bprm.filename = filename;
    bprm.argc = count(argv);
    bprm.envc = count(envp);
    ...
```

---

<sup>5</sup> ELF steht für *Executable and Linkable Format*.

Die Funktion `prepare_binrpm` prüft die Zugriffsrechte und liest die ersten 128 Byte der Datei ein.

```
    retval = prepare_binrpm(&brpm);
    if (retval <= 0)
        goto error;
```

Jetzt kann anhand der ersten Bytes der Datei geprüft werden, wie diese geladen werden soll. LINUX verwendet für jedes ihm bekannte Dateiformat eine eigene Funktion zum Laden der Datei. Sie werden nacheinander aufgerufen und „gefragt“, ob sie die Datei laden können. Wenn die Datei geladen werden kann, endet `execve()` erfolgreich, wenn nicht, liefert es `ENOEXEC` zurück.

```
    return seach_binary_handler(&brpm, regs);
} /* do_execve() */

int search_binary_handler(struct linux_binprm *bprm,
                        struct pt_regs *regs)
{
    for( fmt = formats; fmt ; fmt = fmt->next )
    {
        if (!fmt->load_binary)
            continue;
        retval = (fmt->load_binary)(bprm, regs);
        if (retval >= 0) {
            current->did_exec = 1;
            return retval;
        }
        if( retval != -ENOEXEC )
            break;
    }
    return(retval);
}
```

Die eigentliche Arbeit wird also von der Funktion `fmt->load_binary()` erledigt. Betrachten wir eine solche Funktion einmal genauer:

```
int load_aout_binary(struct linux_binprm *bprm,
                   struct pt_regs *regs)
{
```

`bprm->buf` enthält die ersten 128 Bytes der zu ladenden Datei. Zuerst wird anhand dieses Dateianfangs geprüft, ob es sich um das richtige Dateiformat handelt. Wenn dies nicht der Fall ist, liefert diese Funktion den Fehler `ENOEXEC`. Daraufhin kann `search_binary_handler()` weitere Formate überprüfen. Bei den Überprüfungen werden auch gleich einige später benötigte Informationen aus dem Header extrahiert.

```
    struct exec ex;

    ex = *((struct exec *) bprm->buf);
    if ((N_MAGIC(ex) != ZMAGIC && N_MAGIC(ex) != OMAGIC &&
```

```

        N_MAGIC(ex) != QMAGIC && N_MAGIC(ex) != NMAGIC) ||
        N_TRSIZE(ex) || N_DRSIZE(ex) ||
        bprm->dentry->inode->i_size < ex.a_text+ ... )
    {
        return -ENOEXEC;
    }
    ...
    fd_offset = N_TXTOFF(ex);
    ...

```

Wenn diese Prüfungen erfolgreich abgeschlossen wurden, wird das neue Programm geladen. Dazu wird als erstes der Speicher des Prozesses freigegeben, er enthält ja noch das alte Programm. Nach dieser Freigabe kann `execve()` nicht mehr in das alte Programm zurückkehren. Wenn jetzt ein Fehler beim Laden der Datei auftritt, muss der Prozess abgebrochen werden.

```
flush_old_exec(bprm);
```

Jetzt kann die Taskstruktur aktualisiert werden. Dabei wird in der Komponente `personality` registriert, dass es sich um ein Programm im LINUX-eigenen Format handelt.

```
set_personality(PER_LINUX);
```

```

current->mm->end_code = ex.a_text +
    (current->mm->start_code = N_TXTADDR(ex));
...

```

Nun kann das Text- und das Datensegment des Programmes mit `do_mmap()` in den Speicher eingeblendet werden. Man beachte, dass `do_mmap()` die Datei hier nicht lädt, sondern nur die Page-Tabellen aktualisiert und dem Paging-Algorithmus damit angibt, woher er die Speicherseiten bei Bedarf laden kann. Das Paging ist in Abschnitt 4.4 beschrieben.

```

do_mmap(bprm->file, N_TXTADDR(ex), ex.a_text,
        PROT_READ | PROT_EXEC,
        MAP_FIXED | MAP_PRIVATE | ..., fd_offset);
...

```

Jetzt wird das BSS-Segment geladen. Es enthält unter UNIX die uninitialisierten Daten eines Prozesses. Die Funktion `set_brk()` erledigt das. Anschließend werden die Register und speziell der Instruction-Pointer für das neue Programm initialisiert. Dies wird durch die Funktion `start_thread()` erledigt. Wenn der Systemruf `execve` jetzt seine Arbeit beendet, wird die Programmausführung des Prozesses an der neuen Adresse fortgesetzt.

```

set_brk(current->mm->start_brk, current->mm->brk);
current->mm->start_stack = ...;
start_thread(regs, ex.a_entry, current->mm->start_stack);
return 0;
} /* load_aout_binary */

```

Im Kern sind die Funktionen `do_execve()` und `load_aout_binary()` erheblich komplizierter. Das liegt zum einen an der notwendigen Fehler- und Ausnahmebehandlung<sup>6</sup>. Andererseits haben wir in der Beschreibung auch viele „unwichtige“ Details weggelassen; „unwichtig“ hier in dem Sinne, dass sie zum Verständnis der Grundprinzipien von `do_execve()` nicht notwendig sind. Wer sich ernsthaft damit beschäftigen und etwa ein neues Dateiformat implementieren will, wird um das Studium der Originalquellen nicht herumkommen.

## Der Systemruf `exit`

Ein Prozess wird immer durch den Aufruf der Kernfunktion `do_exit()` beendet. Dies geschieht entweder direkt durch den Systemruf `_exit` oder indirekt beim Auftreten eines Signals, welches nicht abgefangen werden kann.

Eigentlich hat `do_exit()` nicht viel zu tun. Es müssen nur die vom Prozess belegten Ressourcen freigegeben und eventuell andere Prozesse benachrichtigt werden. Hier steckt jedoch viel Aufwand im Detail, weswegen die folgende Beschreibung der Funktion `do_exit()` auch wieder stark gekürzt ist. Wir betrachten zum Beispiel nicht die Aktionen, die zur sauberen Verwaltung der Prozessgruppen und Threads notwendig sind.

```
NORET_TYPE void do_exit(long code)
{
```

Als erstes gibt der Prozess alle von ihm belegten Ressourcen frei.

```
    del_timer(&current->real_timer);
    sem_exit();
    __exit_mm(current);
    __exit_files(current);
    __exit_fs(current);
    __exit_sighand(current);
    exit_thread();
```

Der Elternprozess wird vom Ableben des Kindprozesses informiert. Eventuell wartet dieser ja schon mit dem Systemruf `wait` auf das Ende des Kindprozesses. Wenn ein Prozess seine Arbeit beendet, müssen auch alle Kindprozesse einen neuen Elternprozess bekommen. Standardmäßig erbt der Prozess 1 alle Kindprozesse. Falls er nicht mehr existieren sollte, werden sie an den Prozess 0 vererbt. Dieses erledigt alles die Funktion `exit_notify()`. Desweiteren setzt sie auch den Status des aktuellen Prozesses von `TASK_RUNNING` auf `TASK_ZOMBIE`

```
    exit_notify();
```

Alle Aufräumarbeiten sind erledigt. Für den Prozess wird jetzt (außer für die Taskstruktur) kein Speicherplatz mehr benötigt. Er wird zu einem Zombie-Prozess. Der Prozess bleibt so lange ein Zombie-Prozess, bis der Elternprozess den Systemruf `wait` ausführt.

<sup>6</sup> Es kann zum Beispiel vorkommen, dass ältere LINUX-Binaries nicht mit `do_mmap()` geladen werden können. Dann muss `load_aout_binary()` den Programmcode und die Daten vollständig laden und kann sich nicht auf das *Demand-Loading* verlassen.

```
current->exit_code = code;
```

Schließlich ruft `do_exit()` den Scheduler auf und erlaubt anderen Prozessen die Weiterarbeit. Da der Status des aktuellen Prozesses `TASK_ZOMBIE` ist, kehrt die Funktion `schedule()` an dieser Stelle nie mehr zurück.

```
    schedule();
    /* NOTREACHED */
} /* do_exit */
```

## Der Systemruf `wait`

Der Systemruf `wait4` ermöglicht das Warten auf das Ende eines Kindprozesses und die Abfrage des vom Kindprozess gelieferten Exit-Codes. `wait4` wartet dabei, je nach übergebenem Argument, entweder auf einen bestimmten Kindprozess, einen Kindprozess aus einer bestimmten Prozessgruppe oder auf jeden Kindprozess. Genauso lässt sich steuern, ob `wait4` wirklich auf das Ende eines Kindprozesses warten oder ob er nur bereits beendete Kindprozesse beachten soll. Da all diese Fallunterscheidungen ziemlich langweilig sind, beschreiben wir im Folgenden eine modifizierte Version von `wait4`, deren Semantik ungefähr der von `wait` entspricht. (Normalerweise ist `wait` eine Bibliotheksfunktion, die `wait4` mit passenden Argumenten aufruft.)

```
int sys_wait( ... )
{
repeat:
```

`sys_wait()` besteht aus zwei Teilen. Zuerst wird geprüft, ob es bereits einen Kindprozess im Zustand `TASK_ZOMBIE` gibt. Wenn ja, haben wir den gesuchten Prozess gefunden und `sys_wait()` kann erfolgreich zurückkehren. Vorher werden noch statistische Daten aus der Prozesstabelle des Kindprozesses übernommen (verbrauchte Systemzeit, Exit-Code usw.) und anschließend die Taskstruktur des Kindprozesses freigegeben. Dies ist die einzige Möglichkeit, einen Prozesseintrag wieder aus der Prozesstabelle zu entfernen.

```
    nr_of_childs = 0;
    for (p = current->p_cptr ; p ; p = p->p_osptr)
    {
        ++nr_of_childs;

        if(p->state == TASK_ZOMBIE)
        {
            current->times.tms_cutime += p->times.tms_utime +
                p->times.tms_cutime;
            current->times.tms_cstime += p->times.tms_stime +
                p->times.tms_cstime;

            if (stat_addr)
                put_user(p->exit_code, stat_addr);

            release(p);
```

```

        return p;
    }
}

```

Falls es keine Kindprozesse gibt, kehrt `sys_wait()` sofort zurück.

```

    if (nr_of_chlds == 0)
        return 0;

```

Falls es doch Kindprozesse gibt, wird auf das Ende eines der Kindprozesse gewartet. Dazu trägt sich der Elternprozess in die dafür bestimmte Warteschlange seiner eigenen Taskstruktur ein. Wie wir oben gesehen haben, weckt jeder Prozess beim Systemruf `_exit` alle in dieser Warteschlange wartenden Prozesse mit Hilfe der Funktion `wake_up()` auf. Dadurch ist garantiert, dass der Elternprozess über das Ende eines Kindprozesses informiert wird.

```

    interruptible_sleep_on(&current->wait_chldexit);

```

Das beim Beenden des Kindprozesses von `do_exit()` gesendete Signal `SIGCHLD` wird ignoriert. Falls zwischendurch ein Signal empfangen wurde (immerhin kann `interruptible_sleep_on()` auch von einem Signal unterbrochen worden sein), wird der Systemruf mit einer Fehlermeldung beendet. Ansonsten wissen wir, dass es jetzt einen Kindprozess gibt, der sich im Zustand `TASK_ZOMBIE` befindet, und können wieder von vorne anfangen, ihn zu suchen.

```

    current->signal &= ~(1<<(SIGCHLD-1));

    if (current->signal & ~current->blocked)
        return -EINTR;

    goto repeat;
} /* sys_wait */

```