PEARSON Education

ADDISON-WESLEY

open source library

Robert Love

Linux-Kernel-Handbuch

Leitfaden zu Design und Implementierung von Kernel 2.6

Übersetzt von Erik Keller

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Prozessmanagement

Der Prozess ist eine der grundlegenden Abstraktionen in Unix-Betriebssystemen.[1] Ein Prozess ist ein Programm (Objektcode, der auf irgendeinem Medium gespeichert ist) während der Ausführung. Prozesse sind jedoch mehr als der ausgeführte Programmcode (dieser wird unter Unix oftmals als *text section* bezeichnet). Prozesse enthalten zusätzlich einen Satz von Ressourcen wie geöffnete Dateien und anhängige Signale, interne Kernel-Daten, Prozessorstatus, einen Adressbereich, einen oder mehrere *Threads zur Ausführung* und eine *data section*, die die globalen Variablen enthält. Prozesse sind effektiv das »lebende« Resultat von gerade ausgeführtem Programmcode.

Threads of Execution (Threads der Ausführung), oftmals verkürzt *Threads* genannt, sind die Objekte, die die Aktivitäten innerhalb des Prozesses darstellen. Jeder Thread enthält einen einzigartigen *program counter*, Prozess-Stack und einen Satz Prozessorregister. Der Kernel verwaltet (engl. *schedules*) einzelne Threads und nicht die Prozesse. In traditionellen Unix-Systemen besteht jeder Prozess aus einem Thread. In modernen Systemen sind jedoch Multithreaded-Programme – Programme, die aus mehr als einem Thread bestehen – der Normalfall. Wie Sie später sehen werden, hat Linux eine einzigartige Implementierung von Threads: Es wird nicht zwischen Threads und Prozessen unterschieden. Für Linux ist ein Thread lediglich eine spezielle Art von Prozess.

Auf modernen Betriebssystemen bieten Prozesse zwei Arten der Virtualisierung: ein virtueller Prozessor und virtueller Speicher. Der virtuelle Prozessor vermittelt dem Prozess die Illusion, dass er das System für sich allein hat, obwohl sich vermutlich Dutzende Prozesse den Prozessor teilen müssen. Kapitel 4 beschreibt diese Virtualisierung. Virtueller Speicher lässt den Prozess Hauptspeicher belegen und verwalten, als wäre er der einzige Prozess auf dem System. Virtueller Speicher wird in Kapitel 11 behandelt. Beachten Sie bitte, dass Threads sich die Abstraktion des virtuellen Speichers teilen, während jeder einzelne Thread seinen eigenen virtuellen Prozessor benutzt.

Ein Programm ist für sich genommen kein Prozess, ein Prozess ist ein *aktives* Programm und seine benutzten Ressourcen. In der Tat könnten zwei oder mehr Prozesse existieren, die gerade das*selbe* Programm ausführen. Tatsächlich können zwei oder mehr Prozesse existieren, die sich verschiedene Ressourcen (wie geöffnete Dateien oder einen Adressbereich) teilen.

[1] Die andere grundlegende Abstraktion ist das File.

Ein Prozess beginnt sein »Leben«, wenn er (wer hätte es gedacht?) erzeugt wird. In Linux passiert dies im Rahmen des Systemcalls `fork()`. Dieser erzeugt einen neuen Prozess, indem er einen vorhandenen Prozess dupliziert. Der Prozess, der `fork()` aufruft, wird als *Parent* bezeichnet, der neue Prozess als *Child*. Der Parent-Prozess nimmt seine Ausführung wieder auf, und der Child-Prozess beginnt seine Ausführung an dem Ort, an dem der Call zurückgekommen ist. Der `fork()`-Systemcall kommt zweimal vom Kernel zurück: einmal in den Parent-Prozess und einmal in den Child-Prozess. Oftmals ist es nach dem fork nötig, ein neues, anderes Programm auszuführen. Die Familie der `exec*()`-Funktionsaufrufe wird benutzt, um den neuen Adressbereich zu erzeugen und ein neues Programm in ihn zu laden. In modernen Linux-Kerneln wird `fork()` tatsächlich mit dem `clone()`-Systemcall implementiert. Dieser wird später in diesem Kapitel behandelt.

Schlussendlich beendet sich ein Prozess mit Hilfe des `exit()`-Systemcalls. Diese Funktion beendet den Prozess und gibt alle Ressourcen, die von ihm benutzt wurden, wieder frei. Ein Parent kann den Status des beendeten Childs über den `wait()`[2]-Systemcall abfragen. Dieser erlaubt es einem Prozess, auf die Beendigung eines spezifischen Prozesses zu warten. Wenn sich ein Prozess beendet, wird er in einen speziellen *Zombie*-Status gesetzt. Dieser Status wird für die Repräsentation von beendeten Prozessen genutzt, bis ihr Parent entweder `wait()` oder `waitpid()` ausgeführt hat.

Ein anderer Name für einen Prozess ist *Task*. Der Linux-Kernel benutzt intern die Bezeichnung Task für einen Prozess. In diesem Buch werde ich die Begriffe austauschbar benutzen, allerdings meine ich, wenn ich den Begriff *Task* benutze, normalerweise einen Prozess aus Sicht des Kernels.

3.1 Der Prozessdeskriptor und die Task-Struktur

Der Kernel speichert die Liste der Prozesse in einer doppelt gelinkten Liste ab, die *Task-List* genannt wird[3]. Jedes Element in der *Task-List* ist ein Prozessdeskriptor (*process descriptor)* vom Typ `struct task_struct`, die in `<linux/sched.h>` definiert ist. Der Prozessdeskriptor enthält alle Informationen über einen bestimmten Prozess.

Die `task_struct` ist eine relativ große Datenstruktur, ungefähr 1.7 kByte auf einem 32-Bit-System. Diese Größe ist jedoch ziemlich gering, wenn man bedenkt, dass die Struktur alle Informationen enthält, die der Kernel von einem Prozess benötigt. Der Prozessdeskriptor enthält die Daten, die das ausführende Programm beschreiben – offene Dateien, den Adressraum des Prozesses, anhängige Signale, den Status des Prozesses und einiges mehr (siehe Abb. 3.1).

[2] Der Kernel implementiert den `wait4()`-Systemcall. Linux-Systeme stellen üblicherweise über die C-Library die Funktionen `wait()`, `waitpid()`, `wait3()` und `wait4()` zur Verfügung. All diese Funktionen geben den Status eines beendeten Prozesses zurück, werden allerdings mit leicht unterschiedlichen Parametern aufgerufen.

[3] Manchmal wird in Abhandlungen über Betriebssystem-Design auch die Bezeichnung *Task-Array* verwandt. Da die Linux-Implementierung eine *Linked-List* und kein statisches Array ist, wird die Bezeichnung *Task-List* benutzt.

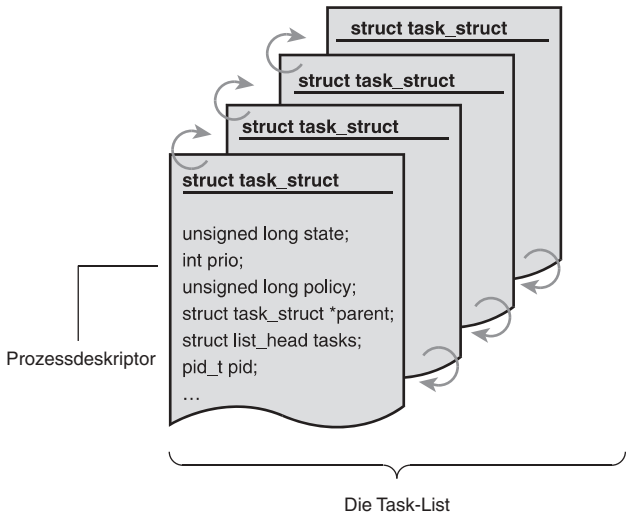

Abbildung 3.1: Prozessdeskriptor und Task-List

3.1.1 Belegen des Prozessdeskriptors

Die `task_struct` wird über den *Slab-Allocator* belegt, um *Object-Reuse* (Objektwiederverwertbarkeit) und *Cache-Coloring* zu ermöglichen (siehe Kapitel 11). Vor der Kernel-Serie 2.6 wurde `struct task_struct` am Ende des Kernel-Stacks eines jeden Prozesses gespeichert. Dadurch konnten Architekturen, denen nur wenige Register zur Verfügung standen (wie etwa x86), den Ablageort des Prozessdeskriptors über den *Stack-Pointer* berechnen, ohne ein zusätzliches Register für die Berechnung zu verwenden. Da der Prozessdeskriptor nun dynamisch mittels des Slab-Allocators erzeugt wird, wurde eine neue Struktur erzeugt: `struct thread_info`, die wieder am Ende des Stacks (bei Stacks, die nach unten wachsen) und am Anfang des Stacks (für Stacks, die nach oben wachsen) steht[4] siehe Abb. 3.2. Die neue Struktur macht es auch wesentlich leichter, Offsets ihrer Werte zur Verwendung in Assembler-Code zu berechnen.

Die Definition der `thread_info`-Struktur für x86 befindet sich in `<asm/thread_info.h>`:

```
struct thread_info {
        struct task_struct   *task;
        struct exec_domain   *exec_domain;
        unsigned long        flags;
        unsigned long        status;
        __u32                cpu;
        __s32                preempt_count;
```

[4] Architekturen, die in Hinblick auf die Anzahl der Register, die ihnen zur Verfügung stehen, als schlecht ausgestattet bezeichnet werden können, waren nicht der einzige Grund dafür, die Struktur `struct thread_info` zu erzeugen.

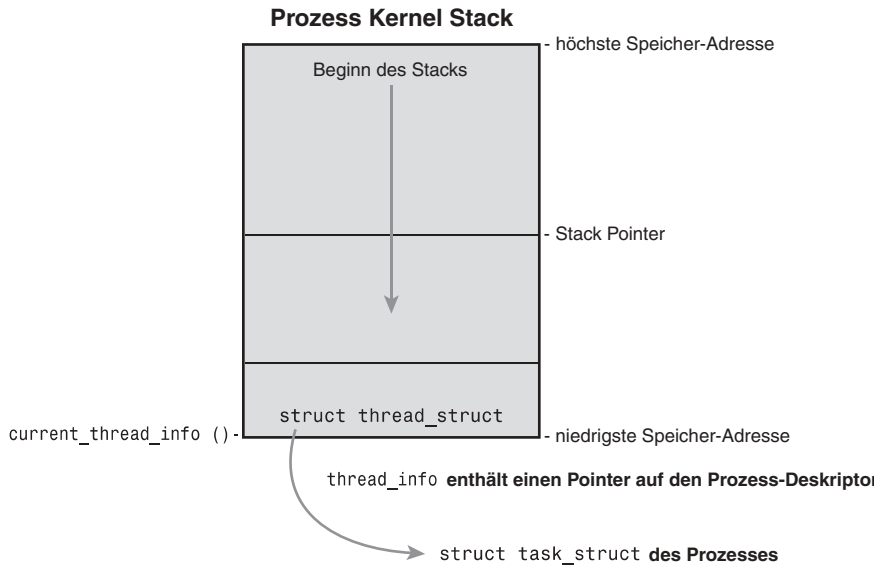

Abbildung 3.2: Der Prozessdeskriptor im Kernel-Stack

```
        mm_segment_t        addr_limit;
        struct restart_block  restart_block;
        unsigned long       previous_esp;
        __u8              supervisor_stack[0];
};
```

Die `thread_info`-Struktur jeder Task wird am Ende ihres Stacks belegt. Das `task`-Element der Struktur ist ein Pointer auf die tatsächliche `task_struct` der Task.

3.1.2 Speicherung des Prozessdeskriptors

Das System identifiziert Prozesse anhand eines eindeutigen *Prozessidentifikations*-wertes, kurz *PID*. Der PID ist ein numerischer Wert, der durch den opaken Typ[5] `pid_t` repräsentiert wird, üblicherweise ein `int`. Um rückwärtskompatibel mit älteren Unix- und Linux-Versionen zu bleiben, wird der maximale Wert standardmäßig auf 32.768 begrenzt (die Größe eines `short int`). Dieser Wert kann jedoch optional bis auf den maximal möglichen Wert des verwendeten Datentyps erhöht werden. Der Kernel speichert diesen Wert als `pid` innerhalb eines jeden Prozessdeskriptors.

Der Maximalwert ist dewegen wichtig, weil er die maximale Anzahl von Prozessen darstellt, die gleichzeitig auf dem System existieren können. 32.768 Prozesse dürften für ein Desktop-System durchaus ausreichen, aber große Server-Systeme benötigen oft deutlich mehr Prozesse. Je kleiner der maximale Wert ist, desto eher werden

[5] Ein opaker Typ ist ein Datentyp, dessen physikalische Repräsentation entweder unbekannt oder irrelevant ist.

die Werte wieder von vorn beginnen (wrap around) und somit die Möglichkeit beschränken, von der Prozessnummer auf den Startzeitpunkt zu schließen (je kleiner die PID, desto eher wurde der Prozess gestartet). Wenn die Kompatibilität mit älteren Systemen oder Programmen nicht benötigt wird, kann der Administrator den Maximalwert über `/proc/sys/kernel/pid_max` erhöhen.

Innerhalb des Kernels werden Tasks üblicherweise direkt durch einen Pointer auf ihre `task_struct`-Struktur referenziert. Tatsächlich arbeitet der größte Teil des Kernel-Codes, der sich mit Prozessen befasst, direkt mit `struct task_struct`. Infolgedessen ist es sehr nützlich, den Prozessdeskriptor der aktuell ausführenden Task schnell zu finden. Diese Suche lässt sich mit dem `current`-Makro durchführen. Dieses Makro muss für jede Architektur einzeln implementiert werden. Manche Architekturen speichern einfach einen Pointer auf die `task_struct` des aktuell laufenden Prozesses in einem Register und erlauben somit einen schnellen Zugriff. Andere Architekturen wie x86 (die zu wenige Register haben, um eines für diesen Zweck zu »verschwenden«) nutzen die Tatsache, dass das `struct thread_info` im Kernel-Stack gespeichert ist, um die Adresse von `thread_info` und somit `task_struct` zu berechnen.

Auf x86 wird `current` durch Ausmaskieren der 13 Least Significant Bits des Stack-Pointers berechnet, um an die `thread_info`-Struktur zu gelangen. Dies wird durch die `current_thread_info()`-Funktion erledigt. Hier ist der Assembler-Code:

```
movl $-8192, %eax
andl %esp, %eax
```

Hierbei wird angenommen, dass die Stack-Größe 8 kByte beträgt. Wenn 4-kByte-Stacks benutzt werden sollten, dann wird 4096 anstatt 8192 verwendet.

Schließlich dereferenziert `current` das Mitglied `task` von `thread_info`, um die `task_struct` zurückzuliefern:

```
current_thread_info()->task;
```

Stellen Sie nun diese Vorgehensweise derjenigen gegenüber, die von der PowerPC-Architektur (IBMs modernem RISC-basierten Mikroprozessor) verwendet wird. Diese speichert die aktuelle `tast_struct` in einem Register. Somit gibt `current` auf einem PPC einfach den Wert im Register `r2` zurück. Bei einem PPC funktioniert dieser Ansatz einfach aufgrund der Tatsache, dass, anders als beim x86, genügend Register zur Verfügung stehen. Da der Zugriff auf den Prozessdeskriptor einen häufig durchgeführten und wichtigen Vorgang darstellt, haben die PPC-Kernel-Entwickler beschlossen, für diesen Zweck ein Register zu benutzen.

3.1.3 Prozess-Status

Das `state`-Feld des Prozessdeskriptors beschreibt den aktuellen Status des Prozesses (siehe Abb. 3.3). Jeder Prozess auf dem System kann sich jeweils in einem von fünf unterschiedlichen Status befinden. Dieser Wert wird durch eines der fünf möglichen Flags dargestellt:

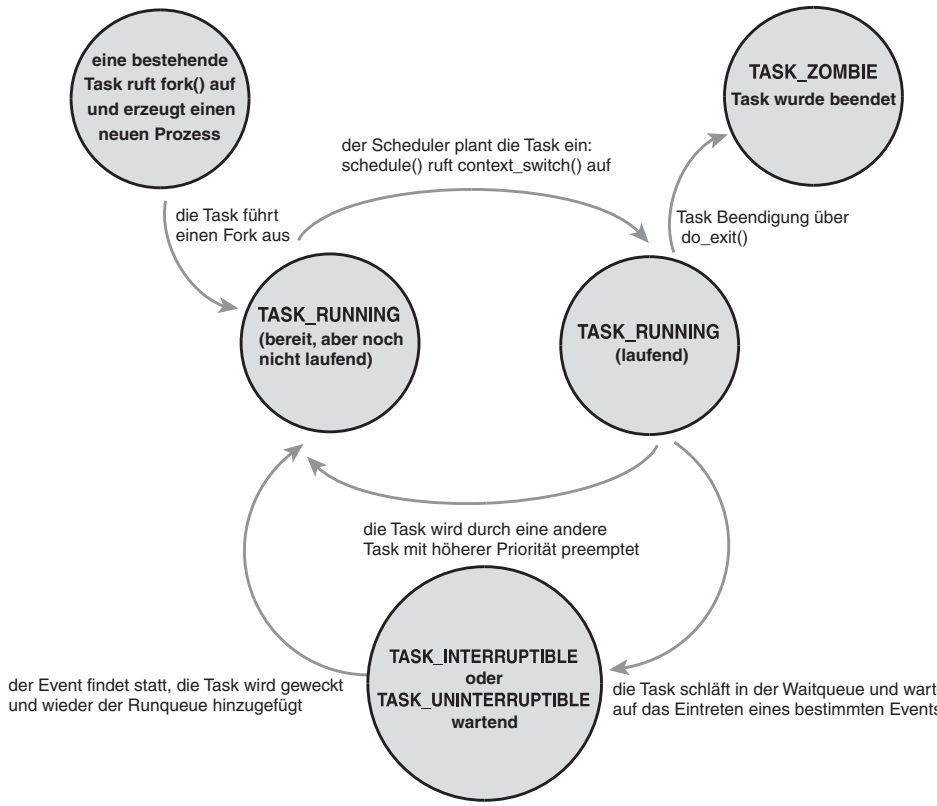

Abbildung 3.3: Ablaufplan der Prozess-Status

- `TASK_RUNNING` – Der Prozess ist lauffähig; er wird entweder gerade ausgeführt oder befindet sich in einer Runqueue und wartet auf seine Ausführung (Runqueues werden in Kapitel 4 ausführlicher behandelt). Dies ist der einzig mögliche Status eines Prozesses, der im User-Space ausgeführt wird; er kann allerdings auch auf einen Prozess im Kernel-Space zutreffen, der gerade ausgeführt wird.

- `TASK_INTERRUPTIBLE` – Der Prozess »schläft« (sleeps), das heißt, er ist geblockt und wartet darauf, dass ein bestimmter Zustand eintritt. Wenn dieser Zustand eintritt, setzt der Kernel den Prozess-Status auf `TASK_RUNNING`. Der Prozess kann auch vorzeitig »aufwachen« (wake up) und lauffähig werden, wenn er ein Signal empfängt.

- `TASK_UNINTERRUPTIBLE` – Dieser Status ist identisch mit `TASK_INTERRUPTIBLE` mit dem Unterschied, dass der Prozess nicht aufwacht und lauffähig wird, wenn er ein Signal empfängt. Dieser Status wird benutzt, wenn der Prozess warten muss, ohne unterbrochen werden zu können, oder wenn damit zu rechnen ist, dass das

Ereignis relativ schnell eintritt. Da die Task in diesem Status nicht auf Signale reagiert, wird `TASK_UNINTERRUPTIBLE` seltener als `TASK_INTERRUPTIBLE` benutzt.[6]

- `TASK_ZOMBIE` – Die Task hat sich beendet, aber ihr Parent hat noch keinen `wait4()`-Systemcall abgesetzt. Der Prozessdeskriptor der Task muss vohanden bleiben, falls der Parent auf ihn zugreifen möchte. Wenn der Parent `wait4()` aufruft, wird der Prozessdeskriptor freigegeben.
- `TASK_STOPPED` – Die Ausführung des Prozesses wurde beendet; die Task läuft nicht mehr und kann auch nicht mehr lauffähig gemacht werden. Dies passiert, wenn die Task eines der Signale `SIGSTOP`, `SIGSTP`, `SIGTTIN` oder `SIGTTOU` empfängt oder wenn sie *irgendein* Signal während des Debuggens empfängt.

3.1.4 Bearbeitung des aktuellen Prozess-Status

Der Kernel muss öfter den Status eines Prozesses ändern. Hierfür wird folgender Mechanismus benutzt:

```
set_task_state(task, state);      /* set task 'task' to state 'state' */
```

Diese Funktion versetzt die angegebene Task in den entsprechenden Status. Falls nötig bietet sie auch eine Speicherbarriere an, um den Prozessor zu wechseln (diese Funktionalität wird nur auf SMP-Systemen benutzt).

Ansonsten entspricht sie:

```
task->state = state;
```

Die Methode `set_current_state(state)` ist synonym zu `set_task_state(current, state)`.

3.1.5 Der Prozesskontext

Einer der wichtigsten Teile eines Prozesses ist der ausgeführte Programmcode. Der Code wird aus einer *ausführbaren Datei* gelesen und innerhalb des Adressbereichs des Programms ausgeführt. Die normale Ausführung eines Programms findet im *User-Space* statt. Wenn ein Programm einen Systemcall ausführt (siehe Kapitel 5) oder eine Exception auslöst, begibt es sich in den *Kernel-Space*. Zu diesem Zeitpunkt sagt man, der Kernel »führt im Namen des Prozesses aus« und befindet sich im *Prozess-Kontext*. Wenn der Kernel sich im Prozess-Kontext befindet, ist das `current`-Makro ausführbar.[7] Sobald der Prozess den Kernel wieder verlassen hat, nimmt er seine Ausführung im User-Space wieder auf, es sei denn, ein Prozess mit höherer

[6] Aus diesem Grunde sehen Sie die gefürchteten nicht beendbaren Prozesse im Status `D` in der Ausgabe von `ps(1)`. Da die Task nicht auf Signale reagieren wird, können Sie ihr auch kein `SIGKILL`-Signal senden. Und selbst wenn Sie die Task beenden könnten, dürfte dies keine vernünftige Entscheidung sein, da die Task vermutlich gerade dabei ist, eine wichtige Operation durchzuführen, und eventuell ein Semaphor hält.

[7] Im Gegensatz zum Prozesskontext steht der *Interrupt-Kontext*; dieser wird in Kapitel 6 behandelt. Im Interrupt-Kontext läuft das System nicht im Namen eines Prozesses, sondern führt einen Interrupt-Handler aus. Da kein Prozess an einen Interrupt-Handler gebunden ist, gibt es in diesem Fall auch keinen Prozesskontext.

Priorität ist mittlerweile lauffähig geworden. In diesem Fall wird der Scheduler aufgerufen, um den Prozess mit der höheren Priorität zu starten.

Systemcalls und Exception-Handler sind klar definierte Schnittstellen in den Kernel. Ein Prozess kann die Ausführung nur durch eine dieser Schnittstellen in den Kernel-Space verlagern – *alle* Zugriffe auf den Kernel laufen durch diese Schnittstellen.

3.1.6 Der Prozess-Stammbaum

Die Prozesse auf Unix-Systemen sind in eine eindeutige Hierarchie eingebunden, und Linux bildet hier keine Ausnahme. Alle Prozesse sind Abkömmlinge des `init`-Prozesses, der die PID `1` hat. Der Kernel startet `init` im letzten Schritt des Boot-Vorgangs. Der `init`-Prozess seinerseits liest die *Initscripts* des Systems, führt weitere Programme aus und beendet schließlich den Boot-Vorgang.

Jeder Prozess auf dem System hat genau einen Parent. Oder andersherum, jeder Prozess hat keine oder mehrere Children (Kinder). Prozesse, die alle direkte Abkömmlinge des gleichen Parent sind, werden Siblings (Geschwister) genannt. Die Verhältnisse zwischen Prozessen werden im Prozessdeskriptor gespeichert. Jede `task_struct` enthält einen Pointer auf die `task_struct` des entsprechenden Parents, genannt `parent`, und eine Liste von Children, `children`. Daher ist es möglich, mit folgendem Code den Prozessdeskriptor des Parents des laufenden Prozesses zu bekommen:

```
struct task_struct *my_parent = current->parent;
```

Gleichermaßen ist es möglich, über die Kinder eines Prozesses zu iterieren:

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
          task = list_entry(list, struct task_struct, sibling);
          /* task now points to one of current's children */
}
```

Der Prozessdeskriptor der `init`-Task ist statisch belegt, und zwar als `init_task`. Ein gutes Beispiel für die Beziehungen zwischen den Prozessen ist die Tatsache, dass sich folgender Code immer ausführen lässt:

```
struct task_struct *task;

for (task = current; task != &init_task; task = task->parent)
          ;
/* task now points to init */
```

Tatsächlich ist es möglich, die Prozesshierarchie von irgendeinem beliebigen Prozess zu *jedem* anderen zu verfolgen. Manchmal ist es jedoch wünschenswert, über

alle Prozesse des Systems zu iterieren. Dies ist einfach, da die Tas- List eine Circular-Doubly-Linked-List ist. Um von jeder gültigen Task aus die nächste Task in der Liste zu finden, benutzen Sie:

```
list_entry(task->tasks.next, struct task_struct, tasks)
```

Die vorherige Task zu finden funktioniert auf die gleiche Weise:

```
list_entry(task->tasks.prev, struct task_struct, tasks)
```

Diese zwei Routinen werden von den Makros `next_task(task)` respektive `prev_task(task)` zur Verfügung gestellt. Schließlich existiert noch das Makro `for_each_process(task)`, das über die gesamte Task List iteriert. Bei jeder Iteration zeigt `task` auf die nächste Task in der Liste:

```
struct task_struct *task;

for_each_process(task) {
          /* this pointlessly prints the name and PID of each task */
          printk("%s[%d]\\n", task->comm, task->pid);
}
```

Bitte beachten Sie: Es kann sehr aufwändig (expensive) sein, über jede Task eines Systems mit vielen Prozessen zu iterieren. Sie sollten einen sehr guten Grund (und keine Alternative) haben, diesen Vorgang durchzuführen.

3.2 Erzeugen eines Prozesses

Unix verwendet ein einmaliges System, um Prozesse zu erzeugen. Die meisten Betriebssysteme implementieren einen *Spawn*-Mechanismus, um einen neuen Prozess in einem neuen Adressbereich zu erzeugen, das ausführbare Programm einzulesen und mit der Ausführung zu beginnen. Unix benutzt den ungewöhnlichen Ansatz, diese Schritte in zwei verschiedene Funktionen zu separieren: `fork()` und `exec()`[8]. Die erstere, `fork()`, erzeugt einen Child-Prozess, der eine Kopie der aktuellen Task darstellt. Diese unterscheidet sich vom Parent nur durch ihre PID (die immer einmalig sein muss), ihre PPID (die PID des Parents) und bestimmte Ressourcen und Statistiken, wie ausstehende Signale, die nicht vererbt werden. Die zweite Funktion, `exec()`, lädt den ausführbaren Code in den Adressbereich und beginnt mit der Ausführung. Die Kombination von `fork()`, gefolgt von `exec()`, entspricht der einzelnen Funktion, die die meisten anderen Betriebssysteme zur Verfügung stellen.

3.2.1 Copy-on-write

Traditionell werden bei einem `fork()` alle Ressourcen, die dem Parent gehören, dupliziert, und die Kopie wird an das Child übergeben. Dieses Vorgehen ist überaus

[8] Mit `exec()` schließe ich hier jedes Mitglied in der Familie der `exec()`-Funktionen mit ein. Der Kernel implementiert den `execve()`-Systemcall, auf dem `execlp()`, `execle()`, `execv()` und `execvp()` aufbauen.

naiv und ineffizient, da viele Daten kopiert werden, die durchaus gemeinsam genutzt werden könnten. Schlimmer noch: Wenn der neue Prozess sofort ein neues Image ausführen sollte, war die ganze Kopiererei sinnlos. `fork()` wurde in Linux durch die Verwendung von *Copy-on-write*-Pages implementiert. Copy-on-write ist eine Technik, um den Kopiervorgang der Daten zu verzögern oder vollständig wegzulassen. Anstatt den Adressbereich des Prozesses zu kopieren, können der Parent und das Child sich eine einzelne Instanz dieser Daten teilen. Die Daten werden jedoch auf eine Weise markiert, dass im Falle eines Schreibzugriffs die Daten dupliziert werden und jeder Prozess mit seiner eigenen Kopie weiterarbeitet. Daher findet der Kopiervorgang nur dann statt, wenn ein Schreibvorgang durchgeführt werden sollte; ansonsten befinden sich die Daten im gemeinsamen *Read-only*-Zugriff. Diese Technik verzögert das Kopieren jeder Page in den Adressraum so lange, bis tatsächlich geschrieben wird. Falls keine Schreibzugriffe auf die Page erfolgen (wenn beispielsweise `exec()` direkt nach `fork()` aufgerufen wird), besteht auch keine Notwendigkeit, den Kopiervorgang durchzuführen. Der einzige Overhead, der bei `fork()` anfällt, ist das Duplizieren der Page Tables des Parents und das Erzeugen des einmaligen Prozessdeskriptors für das Child. Für den häufigen Fall, dass ein Prozess sofort nach dem `fork()` ein neues Image ausführt, verhindert diese Optimierung das unnötige Kopieren großer Datenmengen (den Adressbereich eingeschlossen, dabei kann es leicht um Megabytes im zehnstelligen Bereich gehen). Dies ist eine wichtige Optimierung, da die Unix-Philosophie eine schnelle Prozessausführung erfordert.

3.2.2 fork()

Linux implementiert `fork()` über den `clone()`-Systemcall. Dieser Aufruf akzeptiert eine Reihe von Flags, die spezifizieren, welche Ressourcen sich der Parent- und der Child-Prozess gegebenenfalls teilen sollen (die Flags werden in Abschnitt 3.3 später in diesem Kapitel genauer erläutert). Die Calls `fork()`, `vfork()` und `__clone()` in der Library benutzen alle `clone()` mit den entsprechenden Flags. Der `clone()`-Systemcall benutzt seinerseits `do_fork()`.

Der größte Teil der Arbeit beim *forking* wird von `do_fork()` übernommen, die in `kernel/fork.c` definiert wurde. Diese Funktion benutzt `copy_process()` und startet anschließend den Prozess. Der interessante Teil der Arbeit wird hierbei von der Funktion `copy_process()` durchgeführt:

- Sie ruft `dup_task_struct()` auf, die einen neuen Kernel-Stack, die `thread_info()`-Struktur und `task_struct` für den neuen Prozess erzeugt. Die neuen Werte sind die der aktuellen Task. Zu diesem Zeitpunkt sind die Prozessdeskriptoren des Parent- und des Child-Prozesses identisch.
- Sie überprüft dann, ob der neue Child-Prozess nicht die Anzahl der Prozesse überschreitet, die der aktuelle Benutzer erzeugen darf.
- Nun müssen die Voraussetzungen dafür geschaffen werden, das Child vom Parent unterscheiden zu können. Verschiedene Bestandteile des Prozessdeskriptors werden geleert oder auf Startwerte gesetzt. Die Bestandteile des Prozessdeskrip-

tors, die nicht vererbt werden, enthalten primär statistische Informationen. Der größte Teil der Daten im Prozessdeskriptor wird gemeinsam genutzt.

- Als Nächstes wird der Status des Childs auf `TASK_UNINTERRUPTIBLE` gesetzt, um sicherzustellen, dass es noch nicht gestartet werden kann.
- Nun ruft `copy_process()` `copy_flags()` auf, um `flags` in der `task_struct` zu aktualisieren. Das Flag `PF_SUPERPRIV`, das angibt, ob eine Task mit Superuser-Rechten läuft, wird gelöscht. Dann wird das `PF_FORKNOEXEC`-Flag gesetzt, das einen Prozess kennzeichnet, der noch nicht `exec()` aufgerufen hat.
- Als Nächstes ruft sie `get_pid()` auf, um der Task eine verfügbare PID zuzuweisen.
- Abhängig von den Flags, die an `clone()` übergeben wurden, werden von `copy_process()` entweder Duplikate von offenen Dateien, Dateisystem-Informationen, Signal-Handlern, vom Prozessadressbereich und vom Name-Space erzeugt oder zur gemeinsamen Nutzung freigegeben. Diese Ressourcen werden üblicherweise von den Threads eines Prozesses gemeinsam genutzt; andernfalls werden sie als einmalig (unique) betrachtet und somit kopiert.
- Als Nächstes wird die übrige Timeslice (Zeitscheibe) zwischen dem Parent und dem Child aufgeteilt (siehe Kapitel 4).
- Letztendlich räumt `copy_process()` auf und übergibt dem Aufrufer einen Pointer auf das neue Child.

Zurück in `do_fork()` wird, falls `copy_process()` erfolgreich war, das neue Child »aufgeweckt« und gestartet. Der Kernel startet den Child-Prozess bewusst als Erstes[9]. Für den häufig vorkommenden Fall, dass der Child-Prozess sofort `exec()` aufruft, eliminiert dies allen Copy-on-write-Overhead, der auftreten würde, wenn der Parent zuerst gestartet wäre und begonnen hätte, in den Adressbereich zu schreiben.

3.2.3 vfork()

Der `vfork()`-Systemcall tut das Gleiche wie `fork()`, außer dass die Page-Table-Einträge des Parent-Prozesses nicht kopiert werden. Stattdessen wird das Child als einziger Thread im Adressbereich des Parent-Prozesses ausgeführt, und der Parent ist so lange blockiert, bis das Child entweder `exec()` aufruft oder sich beendet. Das Child hat *keine* Berechtigung, in den Adressbereich zu schreiben. Dies war eine willkommene Optimierung, als der Call in der Zeit von 3BSD eingeführt wurde, da zu dieser Zeit Copy-on-write-Pages nicht zur Implementierung von `fork()` genutzt wurden. Heutzutage ist (durch die Copy-on-write- und Child-runs-first-Semantik) der einzige Vorteil von `vfork()`, dass die Page-Table-Einträge des Parents nicht kopiert werden. Wenn Linux eines Tages Copy-on-write-Page-Table-Einträge bekommen sollte, dann bleibt überhaupt kein Vorteil mehr übrig[10].

[9] Amüsanterweise funktioniert dies momentan noch nicht korrekt, das Ziel ist jedenfalls, das Child zuerst zu starten.

[10] Tatsächlich existieren bereits Patches, um diese Funktionalität zu Linux hinzuzufügen. Mit der Zeit wird dieses Feature vermutlich in den Mainline-Linux-Kernel aufgenommen.

Da die Semantik von `vfork()` durchaus als verzwickt bezeichnet werden darf (was passiert beispielsweise, wenn `exec()` fehlschlägt?), wäre es schön, wenn `vfork()` einen langsamen schmerzhaften Tod sterben würde. Es ist durchaus möglich, `vfork()` als ganz normalen `fork()` zu implementieren – wie dies in Linux bis 2.2 tatsächlich geschehen ist.

Der `vfork()`-Systemcall ist durch ein spezielles Flag im `clone()`-Systemcall implementiert:

- In `copy_process()` wird der Bestandteil `vfork_done` von `task_struct` auf `NULL` gesetzt.
- In `do_fork()`zeigt, falls das spezielle Flag benutzt wurde, `vfork_done` auf eine spezifische Adresse.
- Nachdem das Child als Erstes gestartet wurde, wartet der Parent – anstatt zurückzukehren – darauf, dass das Child ein Signal über den `vfork_done`-Pointer schickt.
- In der `mm_release()`-Funktion, die aufgerufen wird, wenn eine Task einen Speicheradressbereich verlässt, wird `vfork_done` daraufhin überprüft, ob sein Wert `NULL` ist. Wenn dies nicht der Fall sein sollte, wird dem Parent ein Signal geschickt.
- Zurück in `do_fork()`, wacht der Parent auf und kehrt zurück.

Wenn alles nach Plan gelaufen ist, wird der Child-Prozess in einem neuen Adressbereich ausgeführt und der Parent läuft in seinem ursprünglichen Adressbereich weiter. Der Overhead ist kleiner, aber das Design ist nicht besonders schön anzusehen.

3.3 Die Linux-Implementierung von Threads

Threads sind eine populäre, moderne Abstrahierung in der Programmierung. Sie erlauben mehrere ausführende Threads innerhalb desselben Programms in einem gemeinsam genutzten Adressbereich. Mit ihrer Hilfe können offene Dateien und andere Ressourcen gemeinsam genutzt werden. Threads ermöglichen das *Concurrent Programming* (die Programmierung von parallelen Aufgaben) und, auf Systemen mit mehreren Prozessoren, echte Parallelverarbeitung.

Linux hat eine einzigartige Implementierung von Threads. Der Linux-Kernel kennt das Konzept eines Threads überhaupt nicht. Linux implementiert alle Threads als Standardprozesse. Der Linux-Kernel bietet keine spezielle Scheduling-Semantik oder Datenstrukturen, um Threads zu repräsentieren. Stattdessen ist ein Thread nur ein Prozess, der sich bestimmte Ressourcen mit anderen Prozessen teilt. Jeder Thread hat eine einmalige `task_struct` und erscheint dem Kernel als ganz normaler Prozess (der sich eben Ressourcen wie einen Adressbereich mit anderen Prozessen teilt).

Dieser Ansatz, Threads zu behandeln, unterscheidet sich stark von dem anderer Betriebssysteme wie Microsoft Windows oder Sun Solaris. Diese haben eine explizite Kernel-Unterstützung für Threads (und bezeichnen diese manchmal als *lightweight processes*). Der Name »lightweight process« bringt die Unterschiede in der

Flag	Bedeutung
CLONE_FILES	Parent und Child benutzen offene Dateien gemeinsam.
CLONE_FS	Parent und Child teilen sich Dateisystem-Informationen.
CLONE_IDLETASK	Setzt die PID auf null (wird nur von den *idle*-Tasks benutzt).
CLONE_NEWNS	Erzeugt einen neuen Namespace für das Child.
CLONE_PARENT	Das Child bekommt den*selben* Parent wie sein Parent.
CLONE_PTRACE	Das Child wird ebenfalls getract.
CLONE_SETTID	Die TID (Thread ID) wird in den User-Space zurückgeschrieben.
CLONE_SETTLS	Für das Child wird ein neuer TLS (thread-local storage) erzeugt.
CLONE_SIGHAND	Parent und Child haben die gleichen Signal-Handler und blockierten Signale.
CLONE_SYSVSEM	Parent und Child benutzen gemeinsam die System V-SEM_UNDO-Semantik.
CLONE_THREAD	Parent und Child befinden sich in der gleichen Thread-Gruppe.
CLONE_VFORK	vfork() wurde benutzt, und der Parent wartet darauf, vom Child wieder geweckt zu werden.
CLONE_UNTRACED	Der Tracing-Prozess kann kein CLONE_PTRACE auf das Child erzwingen.
CLONE_STOP	Der Prozess wird im Status TASK_STOPPED gestartet.
CLONE_SETTLS	Es wird ein neuer TLS (thread-local storage) für das Child erzeugt.
CLONE_CHILD_CLEARTID	Die TID im Child wird gelöscht.
CLONE_CHILD_SETTID	Setzt die TID im Child.
CLONE_PARENT_SETTID	Setzt die TID im Parent.
CLONE_VM	Parent und Child teilen sich den Adressbereich.

Tabelle 3.1: clone()-Flags

Philosophie zwischen Linux und den anderen Systemen auf den Punkt. Für die anderen Betriebssysteme sind Threads eine Abstraktion, um eine kleinere, schnellere ausführbare Einheit als einen echten Prozess zur Verfügung zu haben. Für Linux sind Threads einfach eine Art und Weise, um Ressourcen zwischen Prozessen (die man sowieso als »lightweight« bezeichnen kann)[11] gemeinsam zu nutzen. Nehmen Sie an, Sie haben einen Prozess, der aus vier Threads besteht. Auf Systemen, die Threads explizit unterstützen, dürfte ein Prozessdeskriptor existieren, der seinerseits auf die vier unterschiedlichen Threads zeigt. Der Prozessdeskriptor beschreibt die gemeinsam genutzten Ressourcen wie Adressbereich oder offene Dateien. Die Threads ihrerseits beschreiben die Ressourcen, die ihnen allein gehören. Im Gegen-

[11] Machen Sie sich den Spaß und messen Sie die Zeit, die Linux braucht, um einen Prozess zu erzeugen, und vergleichen Sie sie mit der Zeit, die die anderen Systeme brauchen, um einen Prozess (oder einen Thread!) zu erzeugen. Die Resultate werden Ihnen gefallen.

satz hierzu existieren in Linux nur vier Prozesse und somit vier normale task_struct-Strukturen. Die vier Prozesse sind nur konfiguriert, sich bestimmte Ressourcen zu teilen.

Threads werden wie normale Tasks erzeugt, mit dem Unterschied, dass dem clone()-Systemcall Flags übergeben werden, die mit bestimmten gemeinsam zu benutzenden Ressourcen korrespondieren:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Der gezeigte Code entspricht in seinem Verhalten dem normalen fork(), außer dass der Adressbereich, File-System-Ressourcen, Dateideskriptoren und Signal-Handler gemeinsam genutzt werden. Mit anderen Worten: Die neue Task und ihr Parent sind das, was allgemein als Threads bezeichnet wird.

Im Gegensatz dazu kann ein normaler fork() so durchgeführt werden:

```
clone(SIGCHLD, 0);
```

Und vfork() auf diese Weise:

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```

Die Flags, die an clone() übergeben werden, spezifizieren das Verhalten der neuen Prozesse und beschreiben, welche Ressourcen der Parent und das Child sich teilen werden. Tabelle 3.1 listet die Flags, die in <linux/sched.h> definiert sind, und ihre Auswirkungen auf.

3.3.1 Kernel-Threads

Es ist oft hilfreich, wenn der Kernel in der Lage ist, manche Operationen im Hintergrund durchzuführen. Der Kernel ermöglicht dies mit Hilfe von *Kernel-Threads* – Standardprozessen, die nur im Kernel-Space existieren. Der grundlegende Unterschied zwischen Kernel-Threads und normalen Prozessen ist die Tatsache, dass Kernel-Threads keinen Adressbereich haben (ihr mm-Pointer ist NULL). Sie operieren nur im Kernel-Space und führen keine Context-Switches in den User-Space durch. Kernel-Threads sind jedoch *schedulable* und *preemptable* wie normale Prozesse.

Linux delegiert verschiedene Aufgaben an Kernel-Threads, insbesondere die *pdflush*-Task und die *ksoftirqd*-Task. Diese Threads werden während des System-Boot-Vorgangs von anderen Kernel-Threads erzeugt. Ein Kernel-Thread kann in der Tat von einem anderen Kernel-Thread erzeugt werden. Die Schnittstelle, um einen neuen Kernel-Thread von einem bestehenden aus zu erzeugen (spawn), ist:

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

Die neue Task wird mit dem üblichen clone()-Systemcall und den spezifizierten flags als Argument erzeugt. Wenn der Parent-Kernel-Thread sich beendet, gibt er einen Pointer auf die task_struct des Childs zurück. Das Child führt die Funk-

tion, die in fn spezifiziert wurde, mit dem Argument in arg aus. Ein spezielles *Clone-Flag*, CLONE_KERNEL, spezifiziert die üblichen Flags für Kernel-Threads: CLONE_FS, CLONE_FILES und CLONE_SIGHAND. Die meisten Kernel-Threads benutzen diese als ihren flags-Parameter.

Üblicherweise führt ein Kernel-Thread die ihm zugewiesene Funktion für immer aus (oder zumindest, bis das System durchgestartet wird, aber bei Linux weiß man nie). Die Funktion, die der Kernel-Thread startet, implementiert normalerweise eine Schleife, in der der Kernel-Thread bei Bedarf geweckt wird. Er führt dann seine Aufgaben durch und legt sich wieder schlafen.

Wir werden uns mit speziellen Kernel-Threads in späteren Kapiteln genauer befassen.

3.4 Beenden von Prozessen

Es ist traurig, aber Prozesse müssen irgendwann sterben. Wenn ein Prozess beendet wird, gibt der Kernel die von ihm benutzten Ressourcen wieder frei und setzt die Parents über sein unerfreuliches Schicksal in Kenntnis.

Üblicherweise wird ein Prozess zerstört, wenn der Prozess den exit()-Systemcall aufruft – entweder explizit, weil er bereit ist, sich zu beenden, oder implizit, wenn er aus der *main*-Subroutine des Programms zurückkehrt (der C-Compiler platziert einen Aufruf von exit() an die Stelle, an die main() zurückkehrt). Ein Prozess kann also auch unfreiwillig beendet werden. Dies passiert, wenn ein Prozess ein Signal oder eine Exception empfängt, die er nicht verarbeiten oder ignorieren kann. Egal wie ein Prozess beendet wird, der größte Teil der Arbeit wird von dem Systemcall do_exit() erledigt, der eine Anzahl von lästigen Pflichten übernimmt:

- Zuerst setzt er das PF_EXITING-Flag in flags in task_struct.
- Als Zweites ruft er del_timer_sync() auf, um die Kernel-Timer zu entfernen. Wenn diese Funktion zurückkehrt, ist sichergestellt, dass kein Timer mehr in der Warteschlage steht und dass keine Timer-Handler mehr laufen.
- Als Nächstes ruft do_exit(), wenn das BSD Process Accounting aktiviert sein sollte, acct_process() auf, um die Accounting-Informationen zu schreiben.
- Nun ruft er __exit_mm() auf, um die mm_struct, die vom Prozess benutzt wurde, wieder freizugeben (release). Wenn kein anderer Prozess diesen Adressbereich benutzen sollte (mit anderen Worten, wenn er nicht gemeinsam genutzt wird), dann wird die Zuordnung aufgehoben (deallocate).
- Dann ruft er exit_sem() auf. Falls der Prozess sich in einer Warteschlange für ein IPC-Semaphor befunden haben sollte, wird er aus diesem entfernt.
- Danach folgen die Aufrufe __exit_files(), __exit_fs(), exit_namespace() und exit_sighand(), um den *Usage-Count* der Objekte herunterzusetzen, die in Verbindung mit Dateideskriptoren, Dateisystem-Daten, dem Prozessnamensraum und Signal-Handlern stehen. Wenn der Usage-Count bei einem dieser Objekte null errei-

chen sollte, dann wird das Objekt nicht länger von einem Prozess benutzt und wird entfernt.

- Anschließend übergibt er den *Exit-Code* der Task, der in `exit_code` in `task_struct` gespeichert ist, an den Code, den `exit()` benutzt hat, oder an einen anderen Kernel-Mechanismus, der für die Beendigung verwendet wurde. Der Exit-Code wird dort gespeichert, falls der Parent ihn abrufen möchte.
- Dann ruft er `exit_notify()` auf, um Signale an den Parent des Tasks zu senden, verbindet eventuell vorhandene Children des Tasks mit andere Threads in ihrer Thread-Gruppe oder mit den init-Prozess und setzt den Status der Task auf `TASK_ZOMBIE`.
- Schlussendlich ruft `do_exit()` noch `schedule()` auf, um den Switch zu einem anderen Prozess zu vollziehen (Kapitel 4). Da `TASK_ZOMBIE`-Tasks vom Scheduler nicht beachtet werden, ist dies der letzte Code, den die Task jemals ausführen wird.

Der Code für `do_exit()` befindet sich in `kernel/exit.c`.

An diesem Punkt wurden alle Objekte, die in Verbindung mit der Task standen (vorausgesetzt, die Task war der einzige Benutzer) freigesetzt. Die Task ist nicht mehr lauffähig (runnable), hat tatsächlich auch gar keinen Adressbereich mehr, um darin zu laufen, und befindet sich im `TASK_ZOMBIE`-Status. Der einzige Speicher, den sie noch belegt, ist ihr Kernel-Stack, die `thread_info`-Struktur und die `task_struct`-Struktur. Die Task existiert nur noch zu dem Zweck, Informationen für ihren Parent zur Verfügung zu stellen. Nachdem der Parent sich diese Informationen abgeholt hat oder er dem Kernel mitgeteilt hat, dass sie ihn nicht interessieren, wird der verbleibende Speicher, den der Prozess gehalten hat, freigesetzt und an das System zur weiteren Verwendung übergeben.

3.4.1 Die Entfernung des Prozessdeskriptors

Nachdem `do_exit()` gelaufen ist, existiert immer noch der Prozessdeskriptor des beendeten Prozesses, aber der Prozess ist ein Zombie und unfähig, gestartet zu werden. Wie ich erwähnt habe, ermöglicht dies dem System, Informationen über den Child-Prozess zu bekommen, nachdem dieser beendet wurde. Aus diesem Grund ist die Aufgabenstellung zweigeteilt: zum einen den Prozess »aufzuräumen« und zum anderen den Prozessdeskriptor zu entfernen. Nachdem der Parent die Informationen über den beendeten Child-Prozess abgeholt hat oder dem Kernel sein Desinteresse signalisiert hat, wird die `task_struct` freigegeben.

Die Familie der `wait()`-Funktionen ist über einen einzigen (und komplizierten) Systemcall implementiert. Das Standardverhalten ist, die Ausführung der aufrufenden Task so lange zu unterbinden, bis einer der Child-Prozesse beendet wird. Zu diesem Zeitpunkt kehrt die Funktion mit der PID des beendeten Childs zurück. Zusätzlich wird ein Pointer an die Funktion zurückgeliefert, der den *Exit-Code* des beendeten Child-Prozesses enthält. Wenn der Prozessdeskriptor endgültig freigegeben werden soll, wird `release_task()` aufgerufen und führt Folgendes aus:

- Zuerst wird `free_uid()` aufgerufen, um den *Usage-Count* der laufenden Prozesse des Benutzers herunterzusetzen. Linux benutzt einen *per user cache*, um Informationen bezogen auf die Anzahl der Prozesse und Dateien, die ein Benutzer belegt hat, zu verwalten. Wenn der Usage-Count auf null steht, hat der Benutzer keine offenen Prozesse oder Dateien mehr im Zugriff und der Cache wird gelöscht.

- Als zweiten Schritt ruft `release_task() unhash_process()` auf, um den Prozess vom *pidhash* und aus der Task-List zu entfernen.

- Danach, falls die Task *ptraced* wurde, `release_task()`, um die Task wieder mit dem eigentlichen Parent zu verbinden und sie von der Ptrace-Liste zu entfernen.

- Zum Schluss ruft `release_task() put_task_struct()` auf, um die Pages freizugeben, die den Prozess-Kernel-Stack und die `thread_info`-Struktur enthalten, und den *Slab-Cache*, der die `task_struct` enthält, zurückzugeben.

Zu diesem Zeitpunkt wurden der Prozessdeskriptor und alle Ressourcen freigegeben, die ausschließlich zu dem Prozess gehörten.

3.4.2 Das Dilemma der »elternlosen« Tasks

Wenn ein Parent vor seinen Child-Prozessen beendet wird, muss es einen Mechanismus geben, um das Child mit einem anderen Prozess zu assoziieren (reparent), ansonsten würden beendete Child-Prozesse ohne Parent für immer in ihrem Zombie-Status verharren und Speicher im System verbrauchen. Die bereits erwähnte Lösung ist, die Child-Prozesse beim Beenden der Task mit einem anderen Prozess in der aktuellen Thread-Group oder, wenn dies fehlschlagen sollte, mit dem `init`-Prozess zu verbinden. In `do_exit()` wird `notify_parent()` aufgerufen. Diese Funktion benutzt `forget_original_parent()`, um die Verbindung umzustellen:

```
struct task_struct *p, *reaper = father;
struct list_head *list;

if (father->exit_signal != -1)
          reaper = prev_thread(reaper);
else
          reaper = child_reaper;

if (reaper == father)
          reaper = child_reaper;
```

Dieser Code setzt `reaper` auf eine andere Task in der Thread-Gruppe des Prozesses. Wenn keine andere Task in der Thread-Gruppe verfügbar sein sollte, setzt er `reaper` auf `child_reaper`, den `init`-Prozess. Da jetzt ein angemessener Parent für die Child-Prozesse gefunden wurde, muss jeder Child-Prozess gefunden und mit `reaper` verbunden werden.

```
list_for_each(list, &father->children) {
          p = list_entry(list, struct task_struct, sibling);
          reparent_thread(p, reaper, child_reaper);
}

list_for_each(list, &father->ptrace_children) {
          p = list_entry(list, struct task_struct, ptrace_list);
          reparent_thread(p, reaper, child_reaper);
}
```

Dieser Code iteriert über zwei Listen: die *child-list* und die *ptraced-child-list* und assoziiert jeden Child-Prozess. Die Idee dahinter, beide Listen zur Verfügung zu haben, ist interessant; dies ist ein neues Feature im 2.6er Kernel. Wenn eine Task ge-`ptrace`t wird, wird sie temporär mit dem Debugging-Prozess verbunden. Wenn der Parent der Task jedoch beendet wird, muss sie mit den anderen Siblings an einen neuen Prozess gebunden werden. In den früheren Kernel-Serien resultierte dies in einer Schleife über *jeden Prozess im System,* der auf die Rückkehr eines Childs wartete. Die Lösung besteht, wie zuvor beschrieben, einfach darin, eine separate Liste für alle Children eines Prozesses zu führen, die gerade ge-`ptrace`t werden – dies reduziert die Suche auf zwei relativ kleine Listen; somit muss nicht jeder Prozess untersucht werden.

Da der Prozess erfolgreich mit einem neuen Parent verbunden wurde, wurde das Risiko eliminiert, herumirrende Zombie-Prozesse zu produzieren. Der `init`-Prozess ruft routinemäßig `wait()` für seine Child-Prozesse auf und entledigt sich somit aller Zombie-Prozesse, die ihm zugeordnet wurden.

3.5 Zusammenfassung Prozesse

In diesem Kapitel haben wir uns die berühmte Betriebssystem-Abstraktion des Prozesses angesehen. Wir haben dann untersucht, wie Linux Prozesse speichert und repräsentiert (mit Hilfe von `task_struct` und `thread_info`), wie Prozesse erzeugt werden (mit `clone()` und `fork()`), wie neue ausführbare Images in den Adressbereich geladen werden (mit der `exec()`-Familie von Systemcalls), wie die Hierarchie der Prozesse aussieht, wie Parent-Prozesse Informationen über ihre beendeten Child-Prozesse erhalten (mit der `wait()`-Familie von Systemcalls) und wie Prozesse letztendlich sterben (gewaltsam oder beabsichtigt mit `exit()`).

Der Prozess ist eine grundlegende und entscheidende Abstraktion im Inneren eines jeden modernen Betriebssystems und der Grund, warum wir überhaupt ein Betriebssystem haben (um Programme laufen zu lassen).

Das nächste Kapitel befasst sich mit dem *Process Scheduling*, der heiklen und interessanten Art und Weise, wie der Kernel entscheidet, welche Prozesse er zu welcher Zeit und in welcher Reihenfolge startet.