



Prozesse und Prozessverwaltung im Linux-Kernel

Tobias Doerffel

Fakultät für Informatik
TU Chemnitz

10. Mai 2007



Worum geht es?

- Eigenschaften von Prozessen
- Verwaltung von Prozessen
- Wechsel zwischen Prozessen
- Erzeugung und Zerstörung von Prozessen im Kernel
- Implementierung von Threads



Was sind Prozesse?

- ...im Lehrbuch:
Ein Prozess ist eine Instanz eines Programmes, das sich in Ausführung befindet.
- ...aus Sicht des Kernels:
eine Entität, welche mit den ihr zugewiesenen Systemressourcen (Rechenzeit, Speicher usw.) interagiert.
- ...aus Sicht des Nutzers:
i.d.R. ein Programm, das etwas bestimmtes tut



Prozesse im Kernel

- ein Prozess besitzt einen ihm zugewiesenen Adressraum
- innerhalb des Adressraums verschiedene Arten von Speicherseiten:
 - Code
 - Stapelspeicher
 - dynamischer Speicher
- durch Speicherschutzmechanismen von anderen Prozessen abgeschottet



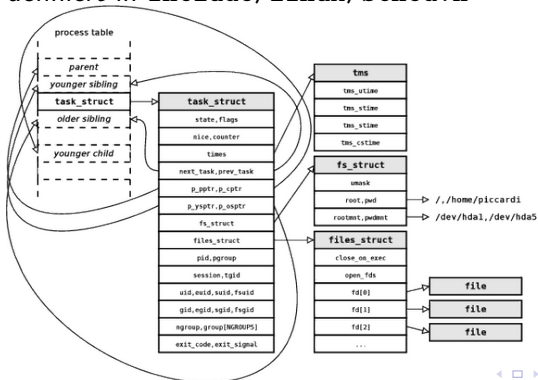
Leichtgewichtprozesse (lightweight processes)

- existieren innerhalb von Prozessen
- gemeinsam genutzter Adressraum und gemeinsame Ressourcen (offene Dateien usw.), **aber:** (logischweise) getrennter Stapelspeicher
- dienen der einfachen Implementierung von Threads durch den Kernel - normale Prozessverwaltungsmechanismen können verwendet werden
- die Menge aller zusammengehörigen Leichtgewichtprozesse bilden Threadgruppen (*thread groups*)



Prozessdeskriptor

- Vielzahl von Informationen über jeden einzelnen Prozess → Speicherung in der Datenstruktur `task_struct` = Prozessdeskriptor
- definiert in `include/linux/sched.h`





Prozessstatus

- `task_struct.state` = Feld mit Flags, die den aktuellen Zustand des Prozesses angeben (exklusiv)
 - `TASK_RUNNING`
 - `TASK_INTERRUPTIBLE`
 - `TASK_UNINTERRUPTIBLE`
 - `TASK_STOPPED`
 - `TASK_TRACED`
 - Sonderfälle `EXIT_ZOMBIE` und `EXIT_DEAD`
- Manipulation mit `set_task_state` und `set_current_state` (Makros)



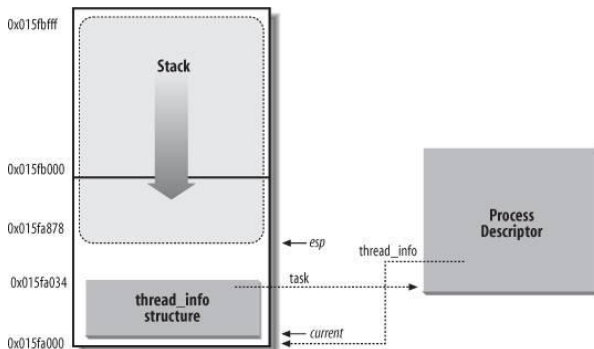
Identifizierung von Prozessen

- `task_struct` für jeden Prozess (auch Leichtgewichtprozesse) praktisch zur Informationsspeicherung - **aber**: zur eindeutigen Identifizierung unpraktisch
- deshalb: Process ID (PID) in Form eines skalaren Wertes (`task_struct.pid`) → fortlaufende Durchnummerierung bis 32.767 (`PID_MAX_DEFAULT-1`)
- Markierung aller belegten PID's in `pidmap_array` (1 Seite)
- Assoziierung von Threads mit Leichtgewichtprozessen → Threads innerhalb eines Prozesses würden unterschiedliche PID's haben → deshalb: Threadgruppen (`task_struct.tgid`)



Kernel-Mode-Stapelspeicher

- 2 zusätzliche Speicherseiten im Kernel für jeden Prozess
 - aufeinanderfolgend
 - an Vielfachen von 2^{13} ausgerichtet





KM-Stapelspeicher

- Realisierung:

```
union thread_union {  
    struct thread_info thread_info;  
    unsigned long stack[2048]  
};
```

- Aktuellen Prozess identifizieren (current-Makro):
 - Ausmaskierung der unteren 13 Bit des Stapelzeigers (ESP)
→ Wert zeigt direkt auf thread_info-Struktur des Prozesses
 - Zeiger auf task_struct ist erstes Feld in thread_info →
einfach vorherigen Wert dereferenzieren → Zeiger auf
Prozessdeskriptor



Die Prozessliste

- Liste aller Prozessdeskriptoren
- Listenkopf = „Prozess 0“/„Swapper“
- Makros `SET_LINKS` und `REMOVE_LINKS` - beachten auch Verwandtschaftsverhältnisse der Prozesse



Liste mit laufenden Prozessen

- allgemein als *runqueue* bezeichnet
- Liste mit Prozessen mit Status `TASK_RUNNING`
- früher: eine Liste mit allen laufenden Prozessen → hoher Verwaltungsaufwand!
- seit Kernel 2.6: *runqueue* aufgeteilt nach Prozessprioritäten in 140 Listen:

```
struct prio_array_t {
    int nr_active;
    unsigned long bitmap[5]
    struct list_head queue;
};
```

- 1 *runqueue* pro CPU



Verwandschaftsbeziehungen zwischen Prozessen

- Prozesse stehen in Eltern/Kind-Beziehung
- vom selben Elternprozess erzeugte Kinder stehen in Geschwisterbeziehung
- weitere Felder im Prozessdeskriptor nötig, um diese Beziehungen abzubilden
 - `real_parent`
 - `parent`
 - `children`
 - `sibling`



PID-Hashtabelle und verkettete Listen 1/2

- Problem: Zuordnung eines Prozessdeskriptors zu einer PID (z.B. kill()-Systemaufruf)
- Möglichkeit: ganze Prozessliste durchsuchen → **ineffizient!**
- stattdessen: 4 Hash-Tabellen für unterschiedliche PID-Typen (PIDTYPE_PID, PIDTYPE_TGID, PIDTYPE_PGID, PIDTYPE_SID)
- Größe der Hash-Tabellen abhängig vom verfügbaren RAM
- Transformation einer PID in einen Hash-Tabellenindex mit dem Makro `pid_hashfn` welches sich wiederum der `hash_long`-Funktion bedient



PID-Hashtabelle und verkettete Listen 2/2

- neues Problem: verschiedene PIDs können denselben Hash-Wert (Tabellenindex) besitzen
- deshalb: Tabelle enthält „nur“ Zeiger auf verkettete Liste `pid_chain` (`include/linux/pid.h`) - Bestandteil der `pid`-Datenstruktur (wiederum Bestandteil eines Prozessdeskriptor)
- `pid_chain` durchlaufen, bis `pid.nr` der gesuchten PID entspricht
- falls Threadgruppenführer gesucht, ist man fertig, ansonsten `pid.pid_list` durchlaufen, um gewünschten Prozessdeskriptor zu finden



Status-spezifische Prozesslisten

- kennen bereits: *runqueue* → Liste mit allen Prozessen im `TASK_RUNNING`-Status
- Prozesse mit anderem Status müssen auch verwaltet werden:
 - `TASK_STOPPED`, `EXIT_ZOMBIE` und `EXIT_DEAD` - keine separaten Listen, da Verwaltung über PIDs und/oder Kinderprozesslisten von Elternprozessen
 - `TASK_INTERRUPTIBLE` und `TASK_UNINTERRUPTIBLE`: Einordnung in spezielle Listen, je nach Typ des Ereignisses, auf die die Prozesse warten → Warteschlangen (`wait queues`)



Warteschlangen

- Prozesse müssen oft auf ein bestimmtes Ereignis warten (Plattenoperation, Freigabe einer Systemresource, Verstreichen einer bestimmten Zeitspanne etc.)
- Prozess ordnet sich in die jeweilige Warteschlange ein und gibt Kontrolle an Kernel ab → Warteschlangen = Listen mit schlafenden Prozessen, die vom Kernel wieder aufgeweckt werden müssen
- wieder einmal: doppelt-verkettete Listen - durch Warteschleifen (*spin locks*) geschützt (wegen Interrupts)
- exklusive und nicht-exklusive Prozesse
- `include/linux/wait.h`



Prozessressourcenlimits

- Schutzmechanismus: jedem Prozess ist eine Menge von *Ressourcenlimits* zugeordnet
- wichtigste Ressourcenlimits:
 - RLIMIT_AS
 - RLIMIT_CPU
 - RLIMIT_DATA
 - RLIMIT_MEMLOCK
 - RLIMIT_STACK

- für jede Resource eine rlimit-Struktur:

```
struct rlimit {  
    unsigned long rlim_cur;  
    unsigned long rlim_max;  
};
```



Prozesswechsel - Übersicht

- Linux=Multitasking-BS → es muss ein ständiger Wechsel zwischen aktiven Prozessen stattfinden
- nicht unbedingt trivial, da sich Prozesse verschiedene Hardware teilen müssen → Zustände müssen gesichert und wiederhergestellt werden → *Hardware-Kontext*
- Hardware-Kontext in z.T. im Prozessdeskriptor gesichert (`thread_struct (include/asm-i386/processor.h)`), Rest im Kernel-Mode-Stapelspeicher
- 80x86-Architektur kann Prozesswechsel komplett in HW ausführen → seit Kernel 2.6 nicht mehr genutzt



Task-State-Segment (TSS)

- Speichersegment von x86-Hardware um Prozesswechsel durchzuführen - seit Kernel 2.4 nur noch rudimentär genutzt
- ein TSS pro CPU (statt pro Prozess) - statt Aktualisierung des tr-Registers, Aktualisierung des TSS bei Prozesswechsel
- Adresse des Kernel-Mode-Stapelspeichers im TSS
- I/O Permission Bitmap
- TSSD



Durchführung des Prozesswechsels

- 2 grundlegende Schritte:
 - Aktualisierung des *Page Global Directory's*, um mit Adressraum des nächsten Prozesses umgehen zu können
 - Kernel-Mode-Stapelspeicher sowie Hardware-Kontext wechseln
- Schritt 2 durch `switch_to`-Makro - sehr hardware-nah/-abhängig
 - 3 Parameter `prev`, `next` und `last`
 - `prev` → `eax`, `next` → `edx`
 - Sichern von `eflags` und `ebp`-Register auf Stapelspeicher von `prev`
 - `esp` in `prev`-Stapelspeicher sichern und von `next` laden
 - Wiedereintrittsadresse in `prev->thread.eip` sichern sowie `next->thread.eip` auf Stapelspeicher
 - Sprung zur `__switch_to()`-Funktion



FPU, MMX und XMM-Register beim Prozesswechsel

- FPU-Register für math. Coprozessor
- aus Kompatibilitätsgründen: Abbildung der MMX-Register auf FPU-Register → Vorteil für BS-Programmierer im Zusammenhang des Prozesswechsels
- XMM-Register für SSE(2)-Erweiterung
- alle Register nur bei Bedarf geladen/gesichert:
HW-Unterstützung durch TS-Flag im Kontrollregister cr0
- auch Kernel verwendet FPU/SSE(2)-Register (z.B. Berechnung von Prüfsummen oder in kryptografischen Algorithmen)



Prozesserzeugung - Übersicht

- seit jeher in der UNIX-Welt: Duplizierung des Adressraums des Elternprozesses für neuen Kindprozess
- Problem: sehr ineffizient, zumal oft anschließend `execve()` aufgerufen wird
- Lösungen/Möglichkeiten:
 - Copy on Write
 - Leichtgewichtprozesse
 - `vfork()`



clone()

- Erzeugung eines Leichtgewichtprozesses
- verschiedene Parameter, u.a. Zeiger auf auszuführende Funktion, User-Mode-Stapelspeicher und verschiedene Flags
- Aufruf von `do_fork()` → `copy_process()`



fork() und vfork()

- implementiert als `clone()`-Systemaufruf
- Stapelspeicherzeiger-Parameter=aktueller Stapelspeicherzeiger (→ Copy on Write)
- `fork()` → keine clone-Flags gesetzt
- `vmfork()` → Flags `CLONE_VM|CLONE_VFORK` gesetzt



copy_process()

- Prozessdeskriptor und andere Kernel-Datentstrukturen des neuen Prozesses initialisieren
- Sicherheitsüberprüfungen (SELinux usw.)
- Aufruf von `dup_task_struct()` (u.a. KM-Stapelspeicher reservieren)
- Ressourcenlimit für Anzahl der Prozesse des Nutzers überprüfen (`user_struct`)
- Datenstrukturen kopieren
- `sched_fork()` aufrufen
- Einfügen in Prozessliste mit `SET_LINKS`-Makro
- Zeiger auf Prozessdeskriptor zurückgeben



Kernel-Threads

- Erledigung von Hintergrundaufgaben (Schreibpuffer leeren, Speicherseiten ein- und auslagern usw.)
- laufen nur im Kernel-Mode
- lineare Speicheradressen $>$ PAGE_OFFSET



Beispiele für Kernel-Threads

- Leerlaufprozess (Prozess 0)
 - statisch allozierte Datenstrukturen (Prozessdeskriptor, KM-Stapelspeicher etc.)
 - erzeugt Init-Prozess
 - danach `cpu_idle()`
 - existiert für jede CPU
- `keventd`, `kapmd`, `kswapd`, `pdflush`, `kblockd`, `ksoftirqd`



Prozesse zerstören

- Ausführung von Prozessen irgendwann beendet bzw. unterbrochen (Ausnahme: Init-Prozess!)
 - `exit()`-Funktion
 - nicht behandeltes Signal oder CPU Ausnahme
- Kernel muss über beendete Prozesse informiert werden, um aufzuräumen



Prozessbeendigung

- 2 Systemaufrufe, um UM-Prozesse zu beenden
 - `exit_group()` → `do_group_exit()` → u.a. `zap_other_threads()` und schließlich `do_exit()`
 - `_exit()` → `do_exit()`
- `do_exit()`:
 - Referenzen auf Prozess in Kernel-Datenstrukturen (Paging, Semaphoren, offene Dateideskriptoren etc.) entfernen
 - `exit_code` im Prozessdeskriptor setzen
 - Aufruf von `schedule()`



Prozess entfernen

- Prozess erst vollständig entfernt bei Aufruf von `waitX()` durch Elternprozess (vorher: `EXIT_ZOMBIE`-Status):
 - Entfernung aller Referenzen im Kernel zum Prozess (Signalhandler, Scheduler etc.)
 - Dekrementierung der Anzahl der Prozesse des Nutzers
- bei Beendigung eines Elternprozesses → Kinderprozesse bekommen übergeordneten Elternprozess