Scheduling im Linux Kernel

Hagen P. Pfeifer
www.jauu.net

18. April 2003

Zusammenfassung

Inhaltsverzeichnis

1 Prolog

Dieses Dokument beschreibt in einigen wenigen Seiten den Scheduling Algorithmus des Linux-Kernels. Es wurde an vereinzelten Stellen der Quellcode gekürzt, mit den Ziel ein verständliches Dokument zu erstellen. Der Lesbarkeit wurde eine höhere Priorität zugeordnet als der Vollständigkeit.

1.1 Version

Das Ihnen vorliegende Dokumment ist aufgebaut auf den Kernelsourcen für die Version 2.4.8.

2 Der Task

2.1 Grundlegendes

Der Task reflektiert die innere Sicht auf die im Kernel laufenden Prozesse[1]. Der Wechsel zwischen Routinen, und damit verbunden auch der Wechsel der Benutzerprozesse, erfolgt in einer kooperativen Umgebung. Jeder laufende Prozess oder aber Thread, im unpriviligierten Modus, wird intern als Task behandelt.
Für eine konsistente und bequeme Strukturierung der benötigten Eigenschaften eines Tasks wurde die Task Struktur geschaffen, welche nachfolgend beschrieben werden soll[2].

2.2 Die Task Struktur

```
struct task_struct {

    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags; /* per process flags, defined below */
    int sigpending;
    struct exec_domain *exec_domain;
    volatile long need_resched;

    long counter;
    long nice;
    unsigned long policy;
    struct mm_struct *mm;

    unsigned long cpus_runnable, cpus_allowed;

    struct list_head run_list;
    unsigned long sleep_time;

    struct task_struct *next_task, *prev_task;
    struct mm_struct *active_mm;
```

[1] Der Begriff Prozess darf nicht mit dem Prozess im Nutzermodus synonym gesetzt werden.
[2] Die Struktur wurde um für den Schedulingalgorythmus unrelevante Daten gekürzt.

```
    struct list_head local_pages;
    unsigned int allocation_order, nr_local_pages;

    pid_t pid;
    unsigned long start_time;
    /* CPU-specific state of this task */
    struct thread_struct thread;
};
```

Primäre Beachtung für den Schedulingprozess ist den nachfolgenden Variablen dieser Datenstruktur zu schenken:

long counter In ihr werden die Ticks des Prozesses festgehalten. Diese Variable wird ständig dekrementiert und bei unterschreiten des Wertes 0 ist die Zeit für diesen Prozess abgelaufen. Es wird ein Scheduling erzwungen.

long nice Enthält die Priorität des Prozesses, welche mit den Systemruf nice() verändert werden kann.

unsigned long policy Diese Variable beschreibt welcher Scheduling-Algorithmus verwendet werden soll.
SCHED_OTHER als klassischer Unix Algorithmus, SCHED_RR oder SCHED_FIFO, welche eine Art von Realtime implementieren.

3 Die Implementierung

3.1 Änderungen

Die Ihnen vorliegende Implementierung der zentralen Scheduling Funktion ist dem Umfang des Dokumentes entsprechend gekürzt. Änderungen welche Aufgrund ihrer Komplexität gekürzt worden sind, betreffen unter anderen das Locking, SMP und Fehlerbehandlung.

3.2 Die Funktion sched.c

```
 1 asmlinkage void schedule(void)
 2 {
 3 struct task_struct *prev, *next, *p;
 4
 5 /* move an exhausted RR process to be last.. */
 6 if (unlikely(prev->policy == SCHED_RR))
 7 if (!prev->counter) {
 8 prev->counter = NICE_TO_TICKS(prev->nice);
 9 move_last_runqueue(prev);
10 }
11
12 switch (prev->state) {
13 case TASK_INTERRUPTIBLE:
```

```
14 if (signal_pending(prev)) {
15 prev->state = TASK_RUNNING;
16 break;
17 }
18 default:
19 del_from_runqueue(prev);
20 case TASK_RUNNING:;
21 }
22 prev->need_resched = 0;
23
24 /*
25  * this is the scheduler proper:
26  */
27
28 repeat_schedule:
29 /*
30  * Default process to select..
31  */
32 next = idle_task(this_cpu);
33 c = -1000;
34 list_for_each(tmp, &runqueue_head) {
35 p = list_entry(tmp, struct task_struct, run_list);
36 if (can_schedule(p, this_cpu)) {
37 int weight = goodness(p, this_cpu, prev->active_mm);
38 if (weight > c)
39 c = weight, next = p;
40 }
41 }
42
43 /* Do we need to re-calculate counters? */
44 if (unlikely(!c)) {
45 struct task_struct *p;
46
47 spin_unlock_irq(&runqueue_lock);
48 read_lock(&tasklist_lock);
49 for_each_task(p)
50 p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
51 read_unlock(&tasklist_lock);
52 spin_lock_irq(&runqueue_lock);
53 goto repeat_schedule;
54 }
55
56 /*
57  * from this point on nothing can prevent us from
58  * switching to the next task, save this fact in
59  * sched_data.
60  */
```

```
    61 switch_to(prev, next, prev);
    62 __schedule_tail(prev);
    63
    64 same_process:
    65 reacquire_kernel_lock(current);
    66 if (current->need_resched)
    67 goto need_resched_back;
    68 return;
}
```

3.3 Die Funktion schedule näher betrachtet

Zeile 5-9 Verschiebt den aktuellen Prozess, wenn er zu der Kategorie der Echtzeitprozesse gehört (SCHED_RR) und seine Zeitscheibe abgelaufen ist, an das Ende der Warteschlange.

Zeile 12-21 Befindet sich der aktuell beendete Prozess im Status TASK_INTERRUPTABLE und wartet auf Signale, wird sein Status auf TASK_RUNNING gesetzt und das need_resched Flag negiert. Befand sich der Prozesse in einem anderen Zustand, wird er von der Warteschlange entfernt und ebenfalls sein need_resched Flag negiert. Die Variable need_resched beschreibt in diesen Fall, dass kein Scheduling durchgeführt werden muss.

28-41 Dieser Algorithmus berechnet den Prozess mit der höchsten Priorität und linkt next auf den vorläufig als nächstes auszuführenden Task. Die Funktion goodness berechnet die Wertigkeit der jeweiligen Prozesse. Diese Funktion wird nachfolgend detaillierter beschrieben.

43-54 Liefert das Resultat von goodness einen Wert von 0 ist dies ein Hinweis, dass die Zeit jedes Prozesses abgelaufen ist, aber lauffähige Prozesse vorhanden sind[3]. An dieser Stelle wird für jeden Prozess ein neuer counter Wert zugeordnet, die Zeile 28 angesprungen und die Prozedur beginnt von neuem.

61 An dieser Stelle wird in den neuen Task gewechselt.

4 Das Auswahlverfahren kann beginnen

4.1 Die Funktion goodness()

```
 1 static inline int
 2 goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
 3 {
 4 int weight;
 5
 6 weight = -1;
 7 if (p->policy == SCHED_OTHER) {
 8 weight = p->counter;
```

[3]Siehe Funktion googdness; Zeile 7 bis 10

```
 9 if (!weight)
10 goto out;
11
12 #ifdef CONFIG_SMP
13 /* Give a largish advantage to the same processor...   */
14 /* (this is equivalent to penalizing other processors) */
15 if (p->processor == this_cpu)
16 weight += PROC_CHANGE_PENALTY;
17 #endif
18
19 /* .. and a slight advantage to the current MM */
20 if (p->mm == this_mm || !p->mm)
21 weight += 1;
22 weight += 20 - p->nice;
23 goto out;
24 }
25
26 /*
27  * Realtime process, select the first one on the
28  * runqueue (taking priorities within processes
29  * into account).
30  */
31 weight = 1000 + p->rt_priority;
32 out:
33 return weight;
34 }
```

4.2 Prioritätenvergabe

Zeile 5-9 Fällt der zu überprüfende Prozess in die Kategorie SCHED_OTHER und ist seine Zeitscheibe abgelaufen, liefert diese Funktion einen Rückgabewert von 0.

Zeile 19-22 Die Priorität wird um den nice-Wert erhöht[4]. Zusätzlich wird ein kleines Bonbon für das aktuelle Speicherlayout gegeben.

Zeile 31 Diese Quellcodezeile gibt Echtzeitprozessen ihre Potenz.

[4]Siehe Bibliotheksfunktion nice