

Realtimescheduling im Linux Kernel

Der Scheduler von Linux wird in seiner Funktionsweise erklärt. Die Beschreibung basiert auf dem Beitrag „Der O(1)-Scheduler im Kernel 2.6“ von Timo Hönig im Linux-Magazin.

Daran anschliessend werden die Systemaufrufe diskutiert, mit dem das Verhalten des Schedulers beeinflusst werden kann.

Inhalt

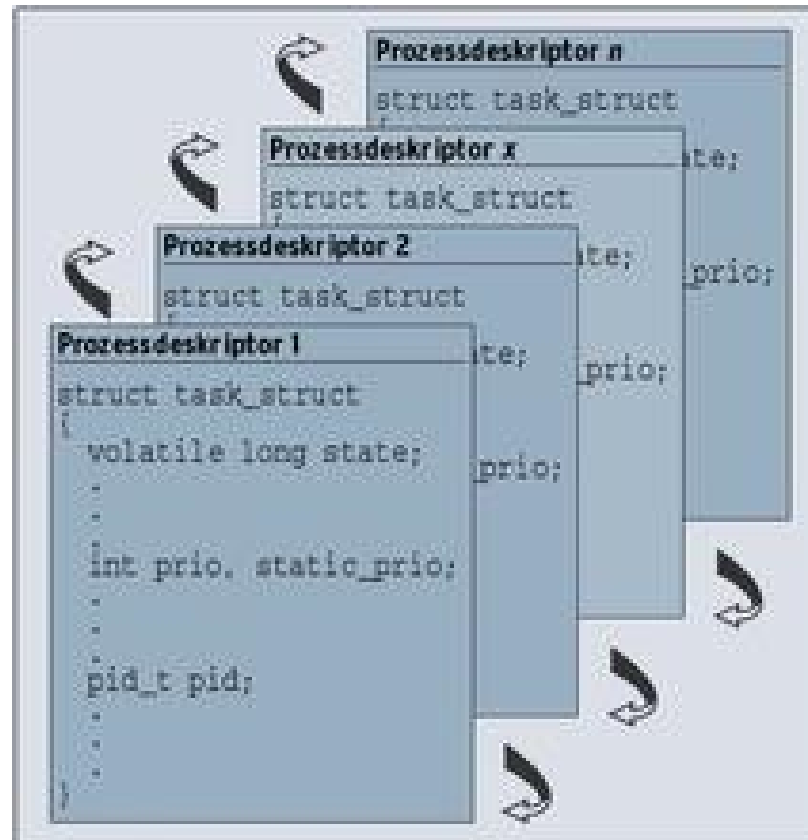
1.Linux Scheduler.....	2
1.1.Prozessverwaltung.....	2
1.2.Entwicklungsziele für den Scheduler.....	5
1.3.Prozessprioritäten.....	5
1.4.Prozess-Zeitscheiben und Neuberechnung der Prioritäten.....	10
1.5.Kernel-Preemption und Realfähigkeit.....	13
1.6.Leistungsvergleich O(n) vs O(1) Scheduler.....	15
2.Systemaufrufe für Scheduleraktivitäten.....	17
2.1.chrt.....	18

1. Linux Scheduler

Aufgabe des Schedulers ist das Multiplexen der CPUs, so daß alle rechenbereiten Prozesse gemäß der Scheduling-Strategie bedient werden.

1.1. Prozessverwaltung

Linux verwaltet alle **Prozessinformationen** mit Hilfe einer **doppelt verketteten Liste** - der Taskliste. Die Listenelemente sind die Prozessdeskriptoren (»task_struct«) der Prozesse.



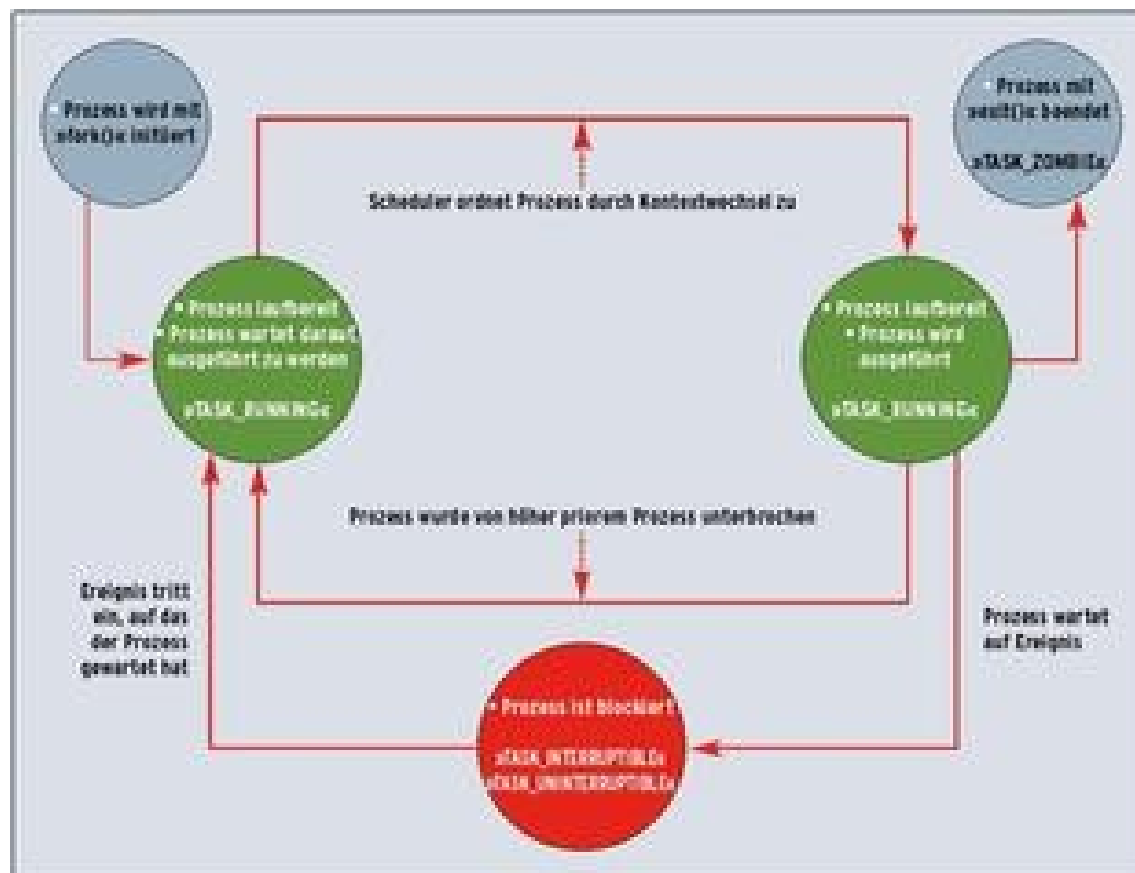
Der Deskriptor hält alle Informationen seines Prozesses fest (im Wesentlichen, das was man mit ps sieht).

Den Zustand eines Prozesses speichert die Variable »**state**« des Prozessdeskriptors. Der Scheduler kennt insgesamt fünf Zustände:

- »**TASK_RUNNING**« kennzeichnet den Prozess als **lauffähig**. Er muss auf kein Ereignis warten und kann daher vom Scheduler der CPU zugeordnet werden. Alle Prozesse im Zustand »TASK_RUNNING« zieht der Scheduler für die Ausführung in Betracht.
- »**TASK_INTERRUPTIBLE**« kennzeichnet blockierte Prozesse. Der Prozess wartet auf ein Ereignis. Ein Prozess im Zustand »TASK_INTERRUPTIBLE« wird über zwei unterschiedliche Wege in den Zustand »TASK_RUNNING« versetzt: Entweder **tritt das Ereignis ein**, auf das er gewartet hat, oder der Prozess wird durch ein **Signal** aufgeweckt.
- »**TASK_UNINTERRUPTIBLE**« gleicht dem Zustand »TASK_INTERRUPTIBLE«, mit dem Unterschied, dass ein **Signal** den Prozess **nicht aufwecken** kann. Der Zustand »TASK_UNINTERRUPTIBLE« wird nur verwendet, wenn zu erwarten ist, dass das Ereignis, auf das der Prozess wartet, zügig eintritt, oder wenn der Prozess ohne Unterbrechung warten soll.
- Wurde ein Prozess beendet, dessen Elternprozess noch nicht den Systemaufruf »wait4()« ausgeführt hat, verbleibt er im Zustand »**TASK_ZOMBIE**«. So kann auch nach dem Beenden eines Kindprozesses der **Elternprozess noch auf seine Daten zugreifen**. Nachdem der Elternprozess »wait4()« aufgerufen hat, wird der Kindprozess endgültig beendet, seine

Datenstrukturen werden gelöscht. Endet ein Elternprozess vor seinen Kindprozessen, bekommt jedes Kind einen neuen Elternprozess zugeordnet. Dieser ist nunmehr dafür verantwortlich, »wait4()« aufzurufen, sobald der Kindprozess beendet wird. Ansonsten könnten die Kindprozesse den Zustand »TASK_ZOMBIE« nicht verlassen und würden als Leichen im Hauptspeicher zurückbleiben.

- Den Zustand »**TASK_STOPPED**« erreicht ein Prozess, wenn er beendet wurde und nicht weiter ausführbar ist. In diesen Zustand tritt der Prozess ein, sobald er eines der Signale »SIGSTOP«, »SIGTST«, »SIGTTIN« oder »SIGTTOU« erhält.



1.2. Entwicklungsziele für den Scheduler

Neben den allgemeinen Zielen (Auslastung der CPU, ...) waren hier zusätzlich folgende Punkte maßgebend:

- gute **SMP-Skalierung**
- geringe Latenz auch bei hoher Systemlast
- **faire Priorität**enverteilung
- **Komplexität** der Ordnung **$O(1)$**

Alle Linux-Scheduler bisher besaßen eine Komplexität der Ordnung $O(n)$: Die Kosten für das Scheduling wuchsen linear mit der Anzahl n der lauffähigen Prozesse. Bei einem Kontextwechsel wird in einer verketteten Liste nach einem Prozess gesucht, dessen Priorität am niedrigsten ist.

Das Ziel des neuen Schedulers war, den Scheduling-Aufwand von der Anzahl der lauffähigen Prozesse abzukoppeln, was der Ordnung $O(1)$ entspricht.

1.3. Prozessprioritäten

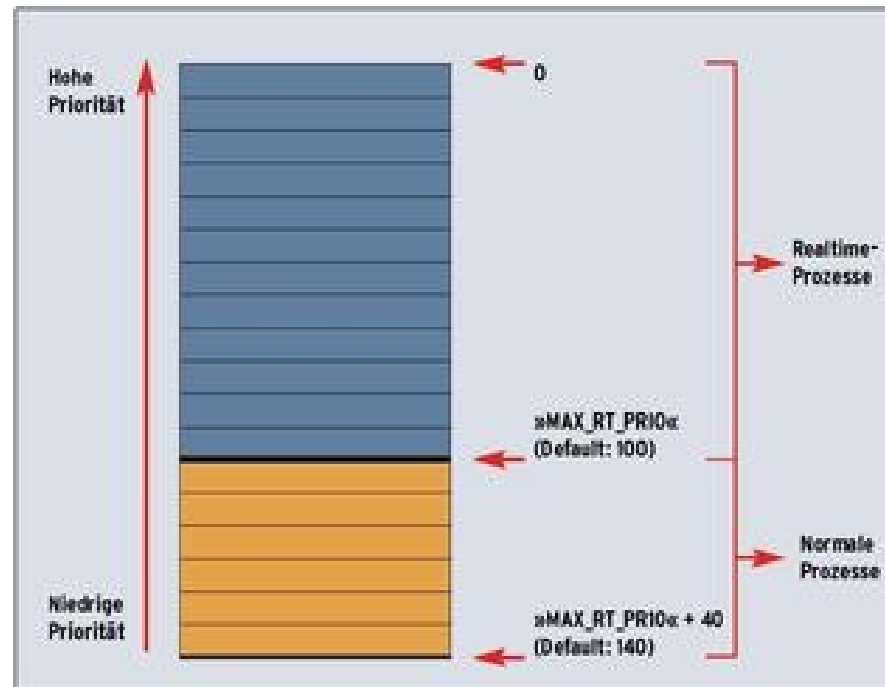
Die **Prozessprioritäten entscheiden**, welchen lauffähigen **Prozess** die CPU beim nächsten Kontextwechsel zugeteilt bekommt - den mit der zum Zeitpunkt des Kontextwechsels höchsten Priorität. Die **Priorität** eines Prozesses **ändert** sich dabei **dynamisch** während seiner Laufzeit.

Es gibt zwei unterschiedliche Prioritäten:

- die **statische** Prozesspriorität, also die vom »nice«-Wert bestimmte »static_ prio«

- die **dynamische** (effektive) Prozesspriorität (»prio«), die **der Scheduler aufgrund der Interaktivität eines Prozesses berechnet**. Der Wertebereich des »nice«-Values reicht von -20 (dem höchsten) bis 19 (dem niedrigsten).

Linux 2.6 kennt standardmäßig **140 Prioritätslevels**. Hierbei entspricht **null** der **höchsten** und 139 der niedrigsten Priorität.



Die Levels von eins bis 99 sind für Tasks mit **Echtzeitpriorität** reserviert. Alle anderen Prozesse erhalten zunächst gemäß ihres »nice«-Werts eine Priorität: Der »nice«-Wert (-20 bis 19) wird einfach in den Bereich ab 101 gemappt. Während des Ablaufs eines Prozesses verändert sich durch seinen Interaktivitätsgrad aber seine Priorität.

Alle **lauffähigen Prozesse** verwaltet der Scheduler in einer **Run-Queue (pro CPU)**. Sie ist die **zentrale Datenstruktur**, auf der der Scheduler arbeitet.

```
struct runqueue {
    /* Spinlock um Run-Queue zu schützen */
    spinlock_t lock;
    /* Zahl der lauffähigen Prozesse */
    unsigned long nr_running;
    /* Zahl der bisherigen Kontextwechsel */
    unsigned long nr_switches;
    /* Zeitstempel des Letzten Tauschs von active- und expired-Array */
    unsigned long expired_timestamp;
    /* Zahl der Prozess im Zustand TASK_UNINTERRUPTIBLE */
    unsigned long nr_uninterruptible;
    /* Verweis auf Prozessdeskriptor des momentan ablaufenden Prozesses */
    task_t *curr;
    /* Verweis auf Prozessdeskriptor der Idle-Task */
    task_t *idle;
    /* Verweis auf Memory Map des zuletzt ablaufenden Prozesses */
    struct mm_struct *prev_mm;
    /* Verweise auf active- und expired-Array */
    prio_array_t *active, *expired;
    /* Priority-Arrays */
    prio_array_t arrays[2];

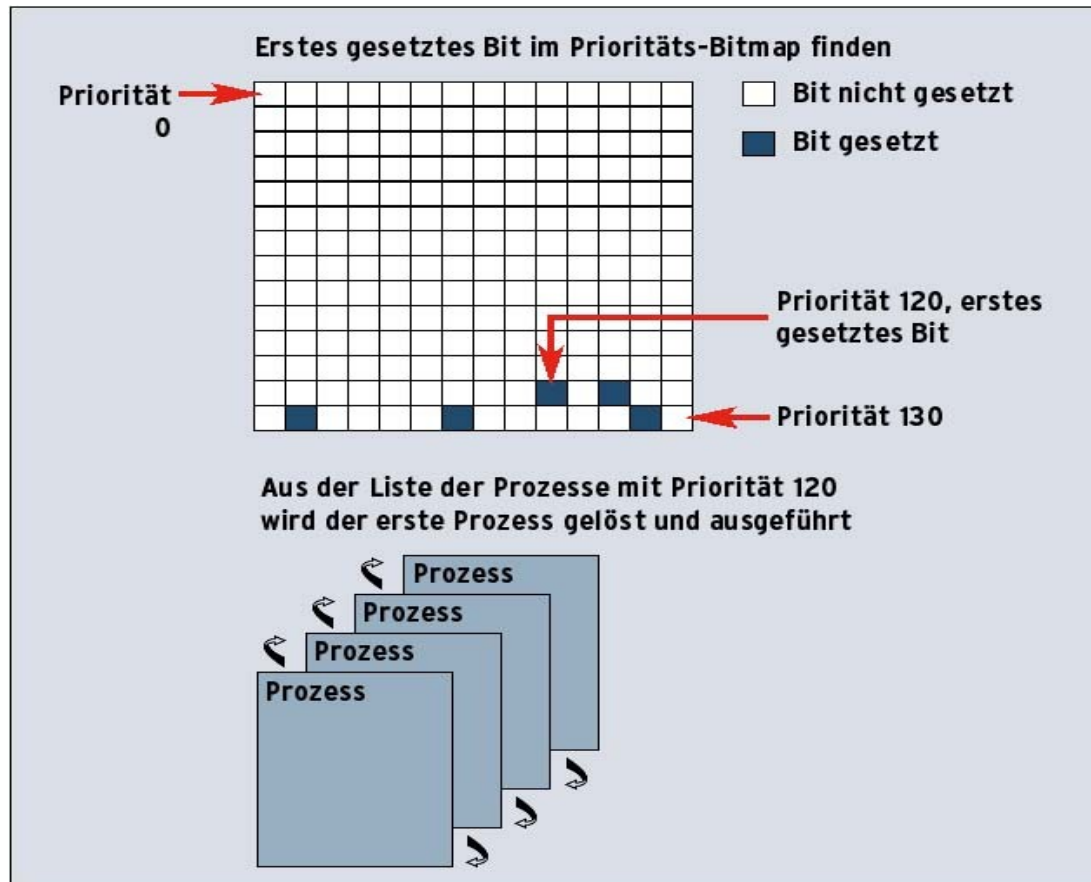
    ...
}
```

Neben Verweisen auf die gerade laufende Task, enthält sie **Verweise** zu den **zwei Priority-Arrays** »**active**« und »**expired**«.

```
struct prio_array {
    /* Zahl der Prozesse */
    int nr_active;
    /* Priorität-Bitmap */
    unsigned long bitmap[BITMAP_SIZE];
    /* Für jede Priorität eine Liste */
    /* der Prozesse mit entsprechender Priorität */
    struct list_head queue[MAX_PRIO];
};
```

Das »**active**«-Array listet alle **lauffähigen Prozesse**, deren **Zeitscheibe noch nicht abgelaufen** ist. Wenn die **Zeitscheibe** eines Prozesse **abläuft, verschiebt** der Scheduler den Eintrag vom »**active**«- in das zweite Array »**expired**«.

Beide, das »**active**«- und das »**expired**«-**Array**, führen für **jede Priorität eine verkettete Liste** der **Prozesse** mit entsprechender Priorität. Eine **Bitmap** hält fest, für welche Priorität mindestens eine **Task** existiert. Alle Bits werden bei der Initialisierung auf null gesetzt. Beim Eintragen eines Prozesses in eines der beiden Priority-Arrays, wechselt entsprechend der Priorität des Prozesses das korrespondierende Bit im Priorität-Bitmap auf eins.



Startet ein Prozess mit dem Nice null, setzt der **Scheduler das 120. Bit** des Prioritäts-Bitmaps im »active«-Array und reiht ihn in die Prozessliste mit Priorität 120 ein

Analog dazu löscht sich das entsprechende Bit im Prioritäts-Bitmap, sobald der Scheduler den letzten Prozess einer gegebenen Priorität aus einem der beiden Priority-Arrays austrägt.

Es ist - wie noch in Linux 2.4 - nicht mehr nötig, die komplette Liste der lauffähigen Prozesse zu durchsuchen, um für den nächsten Taskwechsel den Prozess mit der höchsten Priorität aus-

zumachen. Der **Scheduler** muss lediglich das **erste gesetzte Bit** des **Priorität-Bitmaps** finden (da Bitmap konstante Größe hat folgt $O(1)$). Anschließend führt der Scheduler den ersten Prozess aus der verketteten Liste dieser Priorität aus. Prozesse gleicher Priorität bekommen die CPU nacheinander in einem Ringverfahren (Round Robin) zugeteilt.

1.4. Prozess-Zeitscheiben und Neuberechnung der Prioritäten

Die Zeitscheibe eines Prozesses gibt an, wie lange er laufen darf ohne verdrängt zu werden. Die **Größe der Zeitscheibe** eines Prozesses ist von seiner **Priorität abhängig**: Prozesse mit hoher Priorität erhalten mehr CPU-Zeit als solche mit niedriger. Die kleinste Zeitscheibe beträgt 10, die längste 200 Millisekunden. Ein Prozess mit dem »nice«-Wert null erhält die Standard-Zeitscheibe von 100 Millisekunden.

Ist die **Zeitscheibe** eines Prozesses **aufgebraucht**, muss der **Scheduler** sie **neu berechnen** und den **Prozess** aus dem »active«- in das »expired«-Array **verschieben**. Sobald »active« **leer** ist - alle lauffähigen Prozesse haben ihre Zeitscheibe aufgebraucht -, **tauscht** der Scheduler einfach das »active«- **gegen** das »expired«-Array aus. Effektiv wechseln nur die zwei Pointer der Run-Queue die Plätze.

In Linux 2.4 werden die Zeitscheiben aller Prozesse auf einmal neu berechnet - immer dann, wenn alle Prozesse ihre Zeitscheiben aufgebraucht haben. Mit steigender Prozesszahl dauert die Berechnung immer länger.

Die **dynamische Priorität** errechnet der Scheduler aus der statischen und der **Prozessinteraktivität**. Gemäß seiner **Interaktivität** erhält ein Prozess vom Scheduler entweder einen **Bonus** oder ein Penalty (**Malus**). **Interaktive Prozesse** gewinnen über einen Bonus maximal **fünf Prioritätslevels** hinzu, während jene Prozesse, die eine geringe Interaktivität aufweisen,

maximal fünf Prioritätslevels durch ein Penalty verlieren. Die dynamische Priorität eines Prozesses mit einem »nice«-Wert fünf beträgt demnach im besten Fall null und im schlechtesten zehn.

Um den **Grad der Interaktivität eines Prozesses** zu bestimmen, muss bekannt sein, ob der Prozess eher **I/O-lastig** (I/O-Bound, z.B. ein Shell) oder eher **CPU-intensiv** (Processor-Bound, z.B. Video-Codec) ist.

Um Prozesse einer der beiden Kategorien zuordnen zu können, protokolliert der Kernel für jeden Prozess, wie viel Zeit er mit Schlafen verbringt, und wie lange er die CPU in Anspruch nimmt. Die Variable »**sleep_avg**« (Sleep Average) im **Prozessdeskriptor** speichert dafür eine Entsprechung in dem Wertebereich von **null** und **zehn** (»MAX_SLEEP_AVG«).

Läuft ein Prozess, verringert seine »**sleep_avg**« mit jedem **Timer-Tick** ihren Wert. Sobald ein **schlafender Prozess aufgeweckt** wird und in den Zustand »TASK_RUNNING« wechselt, wird »**sleep_avg**« entsprechend seiner Schlafzeit **erhöht** - maximal bis zu »MAX_SLEEP_AVG«. Der Wert von »sleep_avg« ist somit maßgebend, ob ein Prozess I/O- oder Processor-Bound ist. **Interaktive Prozesse** haben eine hohe »**sleep_avg**«, minder interaktive eine niedrige.

Mit der Funktion »effective_prio()« berechnet der Scheduler die **dynamische Priorität** »**prio**« basierend auf der statischen »static_prio« und der Interaktivität »sleep_avg« des Prozesses. Zum **Berechnen** der neuen **Zeitscheibe** greift der Scheduler auf die dynamische Prozesspriorität zurück. Dazu **mappt** er den Wert in den Zeitscheibenbereich »MIN_TIMESLICE« (Default: **10 Millisekunden**) bis »MAX_TIMESLICE« (**200 Millisekunden**).

Interaktive Prozesse mit hohem Bonus und großer Zeitscheibe können ihre Priorität jedoch nicht missbrauchen, um die CPU zu blockieren: Da die Zeit, die der Prozess beim Ausführen im

Zustand »TASK_RUNNING« verbringt, in die Berechnung der »sleep_avg«-Variablen eingeht, verliert solch ein Prozess schnell seinen Bonus und mit ihm seine hohe Priorität und seine große Zeitscheibe.

Ein **Prozess** mit sehr **hoher Interaktivität** erhält nicht nur eine **hohe dynamische Priorität** und somit eine **große Zeitscheibe**: Der Scheduler trägt den Prozess nach Ablauf seiner Zeitscheibe auch wieder sofort in das »**active**«-Array ein, statt wie gewöhnlich ins »expired«-Array. Der Prozess wird dadurch seiner Priorität gemäß wieder zugeordnet und muss nicht auf das Austauschen der Priority-Arrays warten.

Kontextwechsel

Alle **blockierten** Prozesse werden in den so genannten **Wait-Queues** verwaltet. Prozesse, die von »TASK_RUNNING« in den Zustand »TASK_INTERRUPTIBLE« oder »TASK_UNINTERRUPTIBLE« wechseln, gelangen in diese Warteschlange. Anschließend ruft der Kernel »schedule()« auf, damit ein anderer Prozess die CPU erhält.

Sobald das **Ereignis eintritt**, auf das der Prozess in einer Wait-Queue wartet, wird er aufgeweckt, **wechselt seinen Zustand in »TASK_RUNNING«** zurück, verlässt die Wait-Queue und betritt die Run-Queue. Falls der **aufgewachte Prozess eine höhere Priorität** besitzt als der **gerade ablaufende, unterbricht** der Scheduler den **aktuell** laufenden **Prozess** zugunsten des eben aufgewachten.

1.5. Kernel-Preemption und Realtimefähigkeit

Anders als Kernel 2.4 ist der neue Linux-Kernel preemptiv: **Kernelcode**, der gerade ausgeführt wird, **kann unterbrochen** werden. Vor dem Unterbrechen muss gewährleistet sein, dass sich der **Kernel** in einem **Zustand** befindet, der eine Neuordnung der Prozesse zulässt.

Die Struktur »**thread_info**« jedes **Prozesses** enthält zu diesem Zweck den Zähler »preempt_count«. Ist er null, ist der Kernel in einem sicheren Zustand und darf unterbrochen werden. Die Funktion »preempt_disable()« erhöht den Zähler »preempt_count« beim Setzen eines so genannten **Locks** um eins; die Funktion »preempt_enable()« erniedrigt ihn um eins, sobald ein Lock aufgelöst wird. Das Setzen des Locks (und damit das Verbot der Kernel-Preemption) wird immer dann notwendig, wenn beispielsweise eine von zwei Prozessen genutzte Variable vor konkurrierenden Zugriffen zu sichern ist.

Für **Prozesse** mit so genannter *Echtzeitpriorität* (Priorität 1 bis 99) gibt es zwei Strategien:

- »SCHED_FIFO«

ist ein einfacher First-in/First-out-Algorithmus, der **ohne Zeitscheiben** arbeitet. Wird ein Echtzeitprozess mit »SCHED_FIFO« gestartet, **läuft er so lange**, bis er **blockiert oder freiwillig** über die Funktion »sched_yield()« den **Prozessor abgibt**. Alle anderen Prozesse mit einer niedrigeren Priorität sind solange blockiert und werden nicht ausgeführt.

- »SCHED_RR«

verfolgt die gleiche Strategie wie »SCHED_FIFO«, aber zusätzlich mit vorgegebenen Zeitscheiben.

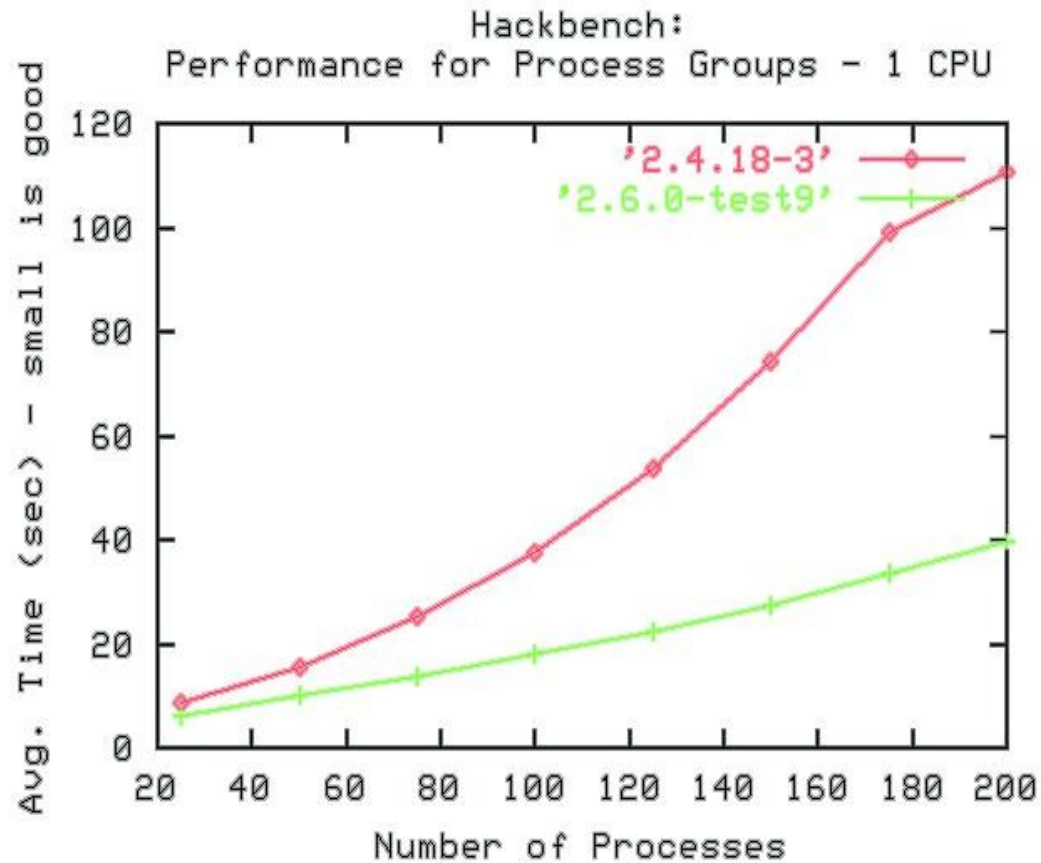
Die CPU-Bedürfnisse der **Echtzeitprozesse gleicher Priorität** befriedigt der Scheduler per **Round-Robin**. Prozesse mit einer niedrigeren Priorität kommen überhaupt nicht zum Zuge. Der Scheduler vergibt für Echtzeitprozesse keine dynamischen Prioritäten. Prozesse ohne Echtzeitpriorität führt er mit der Strategie »SCHED_OTHER« aus.

Die **Echtzeit-Strategien** von Linux **garantieren jedoch keine Antwortzeiten**, was die Voraussetzung für ein hartes Echtzeit-Betriebssystem wäre. Der Kernel stellt jedoch sicher, dass ein lauffähiger Echtzeit-Prozess immer die CPU bekommt, wenn er auf kein Ereignis warten muss, er freiwillig die CPU abgibt und wenn kein lauffähiger Echtzeitprozess höherer Priorität existiert.

Weitere Eigenschaften des Schedulers: Load-Balancing zwischen mehreren CPUs.

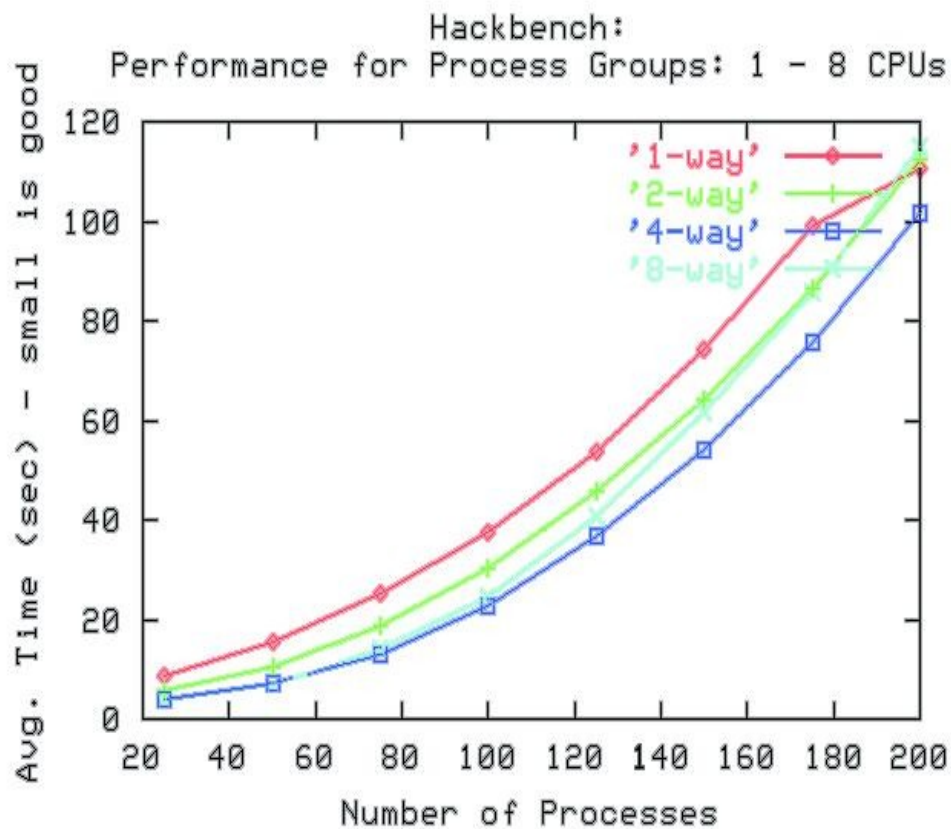
1.6. Leistungsvergleich $O(n)$ vs $O(1)$ Scheduler

Benötigte Zeit zur Interprozess- Kommunikation in Abhängigkeit von der Anzahl beteiligter Prozesse auf einem Singleprozessor-System mit Kernel 2.4 (rot) und 2.6 (grün):

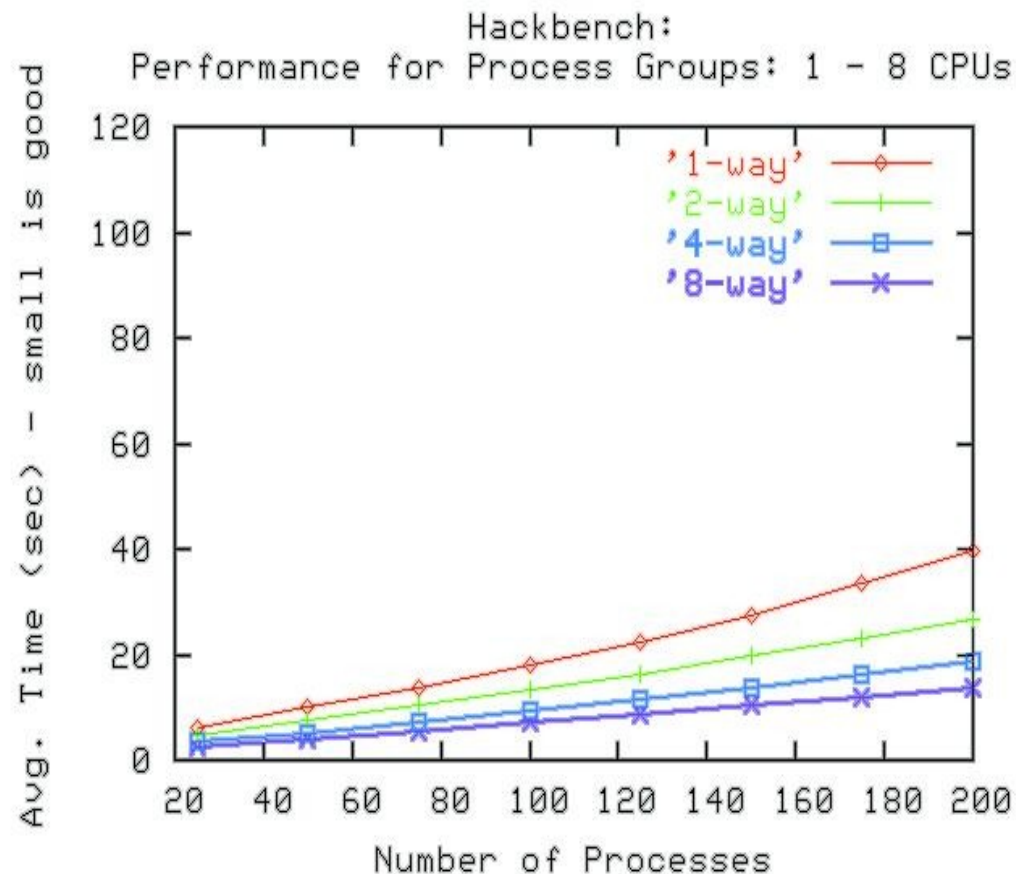


Benötigte Zeit zur Interprozess- Kommunikation in Abhängigkeit von der Anzahl beteiligter Prozesse auf Systemen mit ein, zwei, vier und acht CPUs.

Kernel 2.4



Kernel 2.6



Es ist deutlich zu sehen, dass Linux 2.6 bei steigender Prozesszahl auf allen vier Systemen ungleich besser skaliert als der alte Kernel.

2. Systemaufrufe für Scheduleraktivitäten

Die Systemaufrufe, die den Scheduler in seiner Arbeitsweise beeinflussen, bzw. abfragen werden am Beispiel einiger Open-Source Tools von Robert Love (rlove@rlove.org) erklärt.

```
$ man sched_getscheduler
NAME
    sched_setscheduler, sched_getscheduler - set and get scheduling
    algorithm/parameters

SYNOPSIS
    #include <sched.h>

    int sched_setscheduler(pid_t pid, int policy,
                          const struct sched_param *param);

    int sched_getscheduler(pid_t pid);

    struct sched_param {
        ...
        int sched_priority;
        ...
    };
```

DESCRIPTION

`sched_setscheduler()` sets both the scheduling policy and the associated parameters for the process identified by `pid`. If `pid` equals zero, the scheduler of the calling process will be set.

The interpretation of the parameter `param` depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`, and `SCHED_BATCH`; their respective semantics are described below.

`sched_getscheduler()` queries the scheduling policy currently applied to the process identified by `pid`. If `pid` equals zero, the policy of the calling process will be retrieved.

...

SEE ALSO

`getpriority(2)`, `mlock(2)`, `mlockall(2)`, `munlock(2)`, `munlockall(2)`,
`nice(2)`, `sched_get_priority_max(2)`, `sched_get_priority_min(2)`,
`sched_getaffinity(2)`, `sched_getparam(2)`, `sched_rr_get_interval(2)`,
`sched_setaffinity(2)`, `sched_set-param(2)`, `sched_yield(2)`,
`setpriority(2)`, `capabilities(7)`

\$

2.1.chrt

Das nachfolgende Programm ermöglicht es, die Realtime-Parameter eines Prozesses abzufragen oder einen Prozess mit neuen Parametern zu starten.

```

/*
 * chrt.c - chrt
 * Command-line utility for manipulating a task's real-time attributes
 *
 * Robert Love <rml@tech9.net>
 * 27-Apr-2002: initial version
 *
 */

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <getopt.h>
#include <errno.h>

static void show_usage(const char *cmd)
{
    fprintf(stderr, "chrt version " VERSION "\n");
    fprintf(stderr, "usage: %s [options] [prio] [pid | cmd [args...]]\n",
            cmd);
    fprintf(stderr, "manipulate real-time attributes of a process\n");
    fprintf(stderr, "  -f, --fifo                "
            "set policy to SCHED_FF\n");
    fprintf(stderr, "  -p, --pid                "

```

```

        "operate on existing given pid\n");
fprintf(stderr, "  -m, --max                "
        "show min and max valid priorities\n");
fprintf(stderr, "  -o, --other                "
        "set policy to SCHED_OTHER\n");
fprintf(stderr, "  -r, --rr                "
        "set policy to SCHED_RR (default)\n");
fprintf(stderr, "  -h, --help                "
        "display this help\n");
fprintf(stderr, "  -v, --verbose                "
        "display status information\n");
fprintf(stderr, "  -V, --version                "
        "output version information\n\n");
fprintf(stderr, "You must give a priority if changing policy.\n\n");
fprintf(stderr, "Report bugs and send patches to <rml@tech9.net>\n");
}

```

```
static void show_rt_info(const char *what, pid_t pid)
```

```

{
    struct sched_param sp;
    int policy;

    /* don't display "pid 0" as that is confusing */
    if (!pid)
        pid = getpid();

    policy = sched_getscheduler(pid);

```

```

if (policy == -1) {
    perror("sched_getscheduler");
    fprintf(stderr, "failed to get pid %d's policy\n", pid);
    exit(1);
}

printf("pid %d's %s scheduling policy: ", pid, what);
switch (policy) {
case SCHED_OTHER:
    printf("SCHED_OTHER\n");
    break;
case SCHED_FIFO:
    printf("SCHED_FIFO\n");
    break;
case SCHED_RR:
    printf("SCHED_RR\n");
    break;
default:
    printf("unknown\n");
}

if (sched_getparam(pid, &sp)) {
    perror("sched_getparam");
    fprintf(stderr, "failed to get pid %d's attributes\n", pid);
    exit(1);
}

```

```

    printf("pid %d's %s scheduling priority: %d\n",
           pid, what, sp.sched_priority);
}

static void show_min_max(void)
{
    int max, min;

    max = sched_get_priority_max(SCHED_FIFO);
    min = sched_get_priority_min(SCHED_FIFO);
    if (max >= 0 && min >= 0)
        printf("SCHED_FIFO min/max priority\t: %d/%d\n", min, max);
    else
        printf("SCHED_FIFO not supported?\n");

    max = sched_get_priority_max(SCHED_RR);
    min = sched_get_priority_min(SCHED_RR);
    if (max >= 0 && min >= 0)
        printf("SCHED_RR min/max priority\t: %d/%d\n", min, max);
    else
        printf("SCHED_RR not supported?\n");

    max = sched_get_priority_max(SCHED_OTHER);
    min = sched_get_priority_min(SCHED_OTHER);
    if (max >= 0 && min >= 0)
        printf("SCHED_OTHER min/max priority\t: %d/%d\n", min, max);
    else

```

```

    printf("SCHED_OTHER not supported?\n");
}
int main(int argc, char *argv[])
{
    int i, policy = SCHED_RR, priority = 0, verbose = 0;
    struct sched_param sp;
    pid_t pid = 0;

    struct option longopts[] = {
        { "fifo", 0, NULL, 'f' },
        { "pid", 0, NULL, 'p' },
        { "help", 0, NULL, 'h' },
        { "max", 0, NULL, 'm' },
        { "other", 0, NULL, 'o' },
        { "rr", 0, NULL, 'r' },
        { "verbose", 0, NULL, 'v' },
        { "version", 0, NULL, 'V' },
        { NULL, 0, NULL, 0 }
    };

    while((i = getopt_long(argc, argv, "+fphmorvV", longopts, NULL)) != -1)
    {
        int ret = 1;

        switch (i) {
        case 'f':

```

```
        policy = SCHED_FIFO;
        break;
case 'm':
    show_min_max();
    return 0;
case 'o':
    policy = SCHED_OTHER;
    break;
case 'p':
    errno = 0;
    pid = strtol(argv[argc - 1], NULL, 10);
    if (errno) {
        perror("strtol");
        fprintf(stderr, "failed to parse pid!\n");
        return 1;
    }
    break;
case 'r':
    policy = SCHED_RR;
    break;
case 'v':
    verbose = 1;
    break;
case 'V':
    printf("chrt version " VERSION "\n");
    return 0;
case 'h':
```

```

        ret = 0;
default:
    show_usage(argv[0]);
    return ret;
}

}

if ((pid && argc - optind < 1) || (!pid && argc - optind < 2)) {
    show_usage(argv[0]);
    return 1;
}

if (pid && (verbose || argc - optind == 1)) {
    show_rt_info("current", pid);
    if (argc - optind == 1)
        return 0;
}

errno = 0;
priority = strtol(argv[optind], NULL, 10);
if (errno) {
    perror("strtol");
    fprintf(stderr, "failed to parse priority!\n");
    return 1;
}

```

```
sp.sched_priority = priority;
if (sched_setscheduler(pid, policy, &sp) == -1) {
    perror("sched_setscheduler");
    fprintf(stderr, "failed to set pid %d's policy\n", pid);
    return 1;
}

if (verbose)
    show_rt_info("new", pid);

if (!pid) {
    argv += optind + 1;
    execvp(argv[0], argv);
    perror("execvp");
    fprintf(stderr, "failed to execute %s\n", argv[0]);
    return 1;
}

return 0;
}
```