Der Linux-O(1)-Scheduler

Hynek Schlawack
Universität Potsdam
hs@ox.cx

19. Juni 2003

Inhaltsverzeichnis

1 Vorwort

Auf einem Desktop-PC mit Linux wird ein normaler Benutzer kaum je in die Verlegenheit kommen, Zweifel an der Qualität und Performance des Linux-Schedulers zu hegen. Warum also ein neuer, wenn der alte funktioniert? Warum ein stabiles und erprobtes Konzept verwerfen, um etwas vollkommen Neues einzuführen? Die Antworten auf diese Fragen gibt der nächste große Abschnitt.

Anschließend wird die Funktionsweise des neuen Schedulers besprochen, und zum Teil am Source untersucht. Nach diesem zentralen Abschnitt sollte ersichtlich sein, warum genau der neue Scheduler, zumindest prinzipiell, erheblich besser ist als der alte.

Gefolgt wird dieser Abschnitt von einigen Benchmarks, die die theoretisch ersichtliche Verbesserungen des Schedulers nachweisen oder widerlegen sollen.

Abschließend werden noch einige Probleme des Schedulers unter die Lupe genommen, denn wie bereits angedeutet, wird ein funktionierender Scheduler durch ein vollkommen neues Konzept ersetzt und Revolutionen verlaufen nur selten völlig reibungslos und der Linux-Scheduler ist an dieser Stelle keine Ausnahme.

Bleibt noch anzumerken, dass vom Auditorium ein Grundwissen über Prozesse bei Linux/UNIX erwartet wird und um Abschnitt 2 zu verstehen ist die grobe Kenntnis des alten Schedulers von Vorteil. Obwohl mein Vortrag auf dem über den alten Scheduler aufbaute, versuche ich diese Ausarbeitung davon unabhängig und für jeden verständlich zu machen.

Des Weiteren weise ich darauf hin, dass sich die systemnahen Abschnitte auf die x86-Architektur von Intel beziehen und unter „dem alten Scheduler“ der Scheduler, der standardmäßig in dem derzeit aktuellen Kernel 2.4.21 vorhanden ist, zu verstehen ist.

1.1 Die O-Notation und Fachbegriffe

Zunächst ist es sinnvoll, einige wichtige Begriffe zu klären, die für das Verständnis des Vortrages essenziell sind.

1.1.1 Die O-Notation

Ich werde an dieser Stelle es vermeiden, die O-Notation in ihrer vollen, theoretischen Breite zu erläutern. Stattdessen wende ich es nur direkt auf die Thematik des Schedulers an.

N: Bezeichnet hier die Anzahl der Tasks, die in einem Augenblick laufbereit sind und um die Prozessorzeit konkurrieren.

O(1): Ist die Klasse der Algorithmen, die ihren Zweck vollkommen unabhängig von N, also in unserem Fall der Anzahl der laufbereiten Tasks, erfüllt. Dies ist der optimale Fall.

O(N): Eine Funktion dieser Klasse ist von N abhängig und das proportional. Diese Komplexität entsteht z.B., wenn lineare Listen sequentiell durchsucht werden müssen.

1.1.2 Fachbegriffe

SMP: Ist eine Abkürzung für „symmetric multiprocessing“, was ein Mehrprozessorsystem beschreibt, in dem 2 oder mehr identische CPUs simultan rechnen und auf die gleiche Hardware (Speicher, Festplatten...) zugreifen.

UP: Gegenteil von SMP. Rechnersysteme mit einer CPU.

Affinität: Beschreibt die Bindung eines Prozesses an eine spezielle CPU. Dies ist bei SMP insofern wichtig, da ein Migrieren eines Prozesses von einer CPU auf die andere ungünstig ist - so werden z.B. die Caches komplett invalidiert und dies wirkt sich spürbar auf die Gesamtperformance aus.

Epoche: Dies ist ein Begriff aus dem alten Scheduler. Am Anfang einer Epoche werden Zeitscheiben (auch Timeslices oder Quanten) vergeben und wenn alle Prozesse ihre Scheiben verbraucht haben endet sie und eine neue fängt an.

Task: Ein Task ist eine Ausführungseinheit. Es kann ein regulärer Prozess sein oder aber ein Thread, Tasklet oder sonstiges.

Prioritäten: Bei UNIX gilt für Prioritäten i.d.R., dass je geringer die Zahl, desto höher die Priorität. Was sie genau bedeutet hängt von der Art der Priorität ab.

Nice-Wert: Von Hand angegebene statische Priorität im Bereich -20 bis 19. Je höher der Wert, desto „nicer“ ist der Task zu den Anderen. Sie werden meist beim Start mittels z.B. „nice -n 19 emerge xfree“ angegeben, können aber auch im Nachhinein mittels „renice“ geändert werden. Negative Nice-Werte kann nur der Superuser vergeben.

2 Warum ein neuer Scheduler?

Diese Frage wurde bereits im Vorwort aufgeworfen und in diesem Abschnitt soll sie erörtert werden. Grundsätzlich gibt es bei dem alten Scheduler zwei große Problemfelder: SMP und die Schedule-Performance.

2.1 SMP

Allgemein kann man sagen, dass SMP unter Linux bei weitem hinter der erwarteten Leistung bleibt. Wie im späteren Kapitel mit vergleichenden Benchmarks ersichtlich, schlagen sich in machen Disziplinen die Rechner mit mehr CPUs schlechter, als die mit weniger. Dies hat im wesentlichen zwei Gründe:

2.1.1 Globale Runqueue

Es gibt *eine* globale Runqueue, mit *einem* globalen Lock. Das bringt eine Vielzahl von Problemen mit sich, so kann nur *eine* CPU gleichzeitig schedulen, was spätestens bei zwei- bis dreistelligen CPU-Zahlen katastrophal ist (bzw wäre, das ist einer der Gründe, warum auf solchen Rechnern z.B. Solaris läuft).

Noch schlimmer ist jedoch, dass immer das Ende einer Epoche abgewartet werden muss, bevor neue Zeitquanten verteilt werden. Das bedeutet, dass wenn es nur einen einzigen Task gibt, der noch Rechenzeit übrig hat, stehen allen anderen CPUs still und warten auf das Epochenende, wo die Rechenzeit neu verteilt wird.

2.1.2 Wenig CPU-Affinität

Der alte Scheduler leidet ebefalls unter der Problematik der „bouncenden“ Tasks. Dies ist erneut durch die globale Runqueue bedingt, denn sobald eine CPU die Zeitscheiben der ihr zugeteilten Tasks abgearbeitet hat, erhält sie neue Prozesse zugewiesen anstatt, dass die Tasks neue Zeitscheiben erhielten. Diese liefen eigentlich auf anderen CPUs und hatten dort möglicherweise noch Daten in Caches die dadurch wertlos werden.

2.2 Performance

Die Performance der **schedule()**-Funktion ist im alten Scheduler sehr schlecht, dies begründet sich damit, dass bei jedem Taskwechsel die gesamte Runqueue nach dem besten Task durchsucht wird. Dies wird noch durch den Umstand verschärft, dass es eine globale Runqueue gibt. Es müssen also *alle* Tasks auf ihre Eignung untersucht werden. Somit gehört dieser Algorithmus in die Komplexitätsklasse O(N), er hat also mit zunehmender Taskanzahl eine längere Laufzeit.

Zusätzlich dazu, gibt es noch am Epochenende jeweils eine Zeitscheiben-Berechnung für *alle* laufbereiten Tasks, d.h. nocheinmal ein Algorithmus aus O(N).

Unnötig zu sagen, dass in den Zeiten von Java und anderen Thread-Hogs ein solches Laufzeitverhalten inakzeptabel ist. Nicht zu vergessen, dass bei vielen Tasks auch der L1-Cache durch die **schedule()**-Funktion dank der komplexen Berechnungen und Schleifen invalidiert wird.

Zusammenfassend kann man sagen, dass der alte Scheduler *nicht* oder *schlecht skaliert*.

3 Inside sched.c

Da jetzt bewusst ist, warum der alte Scheduler schlecht ist, ist es an der Zeit, den neuen O(1)-Scheduler auf die Eignung diese Probleme zu lösen, zu untersuchen.

3.1 Allgemeines

Der O(1)-Scheduler wurde ursprünglich von Ingo Molnar entwickelt und am 3. Januar 2002 auf der Linux Kernel Mailingliste (lkml) mit den Worten:

> Jetzt wo die Neujahrparties vorbei sind wird es wieder langweilig. Für die, die etwas komplexeres sehen, oder gar ausprobieren wollen, kündige ich diesen Patch, der ein radikaler Rewrite des Linux-Schedulers ist, für 2.5.2-pre-6 an.

angekündigt (aus dem englischen sinngemäß übersetzt). Mittlerweile ist er standardmäßig im 2.5er-Entwicklerkernel und es existieren Backports für den stabilen Kernel 2.4. Falls ich meine Arbeit gut mache und die Audienz am Ende diesen Scheduler haben will, muss ich sagen, dass er längst in praktisch jedem Distributions-Kernel standardmäßig vorhanden ist und ihn somit praktisch jeder bereits hat.

3.2 Merkmale

Bevor ich nun die Funktionsprinzipien des neuen Schedulers erläutere, möchte ich zunächst noch seine Eckdaten erwähnt haben.

- O(1) für Scheduling - sämtliche Algorithmen, die der neue Scheduler zum reinen schedulen benutzt liegen in der Klasse O(1). Das Scheduling ist also bei jedem Load gleich schnell.
- O(N) für Balancing zwischen CPUs - dies ist nichts besonderes, da einfach alle CPUs überprüft werden und ggf. Balancing vorgenommen wird.
- O(1) für RT-Scheduling

Balancing werde ich in meinem Vortrag nur streifen und RT-Scheduling komplett außen vor lassen, da man bei einem lediglich 45 Minuten kurzen Vortrag Prioritäten setzen muss.

3.3 Prinzip

Der O(1)-Scheduler vereint im Prinzip mehrere bereits bekannte Ansätze zu einem Ganzen.

Eine der wichtigsten Errungenschaften des O(1)-Schedulers ist, dass nun *jede* CPU eine *eigene* Runqueue hat. Damit wird das Problem eliminiert, dass bei n Prozessoren n-1 darauf warten müssen, bis der letzte Prozess einer Epoche seine Zeitscheibe aufgebraucht hat.

Eine solche Runqueue (**struct runqueue**) besteht aus zwei Arrays (**struct prio_array**), die für jede mögliche Priorität eine eigene Liste enthalten. Näheres zum genauen Aufbau dieser Runqueues und Arrays folgt im Abschnitt über die Implementation.

Eines der beiden Arrays enthält die Tasks, die noch etwas von ihrer Zeitscheibe übrig haben. Dieses Array wird über die Strukturvariable **active** angesprochen. Der zweite Array enthält Tasks die Ihre Zeitscheiben aufgebraucht haben und wird über den Pointer **expired** adressiert. Wenn nun ein Task seine Zeitscheibe aufgebraucht hat, wird *sofort* seine neue Zeitscheibe und auch seine neue dynamische Priorität berechnet und der Task wird im Expired-Array korrekt

einsortiert. Sobald das Active-Array leer ist, werden einfach nur die Pointer getauscht und das vorsortierte Expired-Array wird zum Active-Array.

Zusätzlich enthält jedes Array einen Bitmap-Cache mittels dessen, wie im Folgenden ersichtlich wird, es ein einfaches ist, den Task zu finden, der entsprechend seiner Priorität als nächster ausgeführt werden soll.

Prioritäten gibt es im O(1)-Scheduler gleich mehrere:

User Priorities: Im Wesentlichen nur der Nicewert positiv gemacht, d.h. um 20 erhöht. Damit erstreckt sich der Wert von 0 bis 39. Der Sinn ist, dass der Umgang mit positiven Zahlen einfacher ist, als mit negativen.

Static Priorities: Berechnet sich aus einer Basis **MAX_RT_PRIO** die den höchsten Wert für eine Priorität eines Echtzeittasks definiert und der User Priority. $\mathbf{MAX\_PRIO} - 1$ Definiert hierbei das obere Ende. Derzeit ist **MAX_RT_PRIO** = 100, daraus ergibt sich eine maximale Priorität $\mathbf{MAX\_PRIO} - 1$ von 139. Diese Priorität beeinflusst die Berechnung der Zeitscheibe des Tasks.

Dynamic/Effective Priorities: Ist die statische Priorität zzgl. einen Bonus oder Malus im Bereich von -5 bis +5. Die Grenzen der statischen Priorität dürfen jedoch deshalb nicht übertreten werden. Mittels der effektiven Priorität wird die Queue im Priority Array bestimmt zu der der Task hinzugefügt wird. Anzumerken ist, dass im Gegenteil zu dem alten Scheduler die Berechnung nicht-interaktive Tasks *bestraft*, anstatt interaktive zu belohnen.

Auf SMP System ist ebenfalls ein Load-Balancing nötig, um zu verhindern, dass eine CPU überlastet ist und andere idle sind. Hierbei muss man jedoch aufpassen, dass es nicht ausartet, so dass Tasks wahllos zwischen CPUs bouncen und Caches unbrauchbar gemacht werden. Um dies zu verhindern, gibt es Affinitäten. Leider kann ich dieses Thema nicht in voller Breite erörtern, möchte aber noch erwähnt haben, dass in regelmäßigen Abständen (20ms/1ms auf Rechnern mit HZ = 100) versucht wird einen Busy/Idle-Balance durchzuführen, wobei der Idle-Balance gemacht wird, nur wenn die CPU wirklich idle ist.

Ingo Molnar selbst nennt das Ganze:

> hybride priority-list approach coupled with roundrobin scheduling and the array-switch method of distributing timeslices

3.4 Implementation

Die im letzten Abschnitt beschriebenen Prinzipien und Eigenschaften werden nun teilweise auf Sourceebene nachgewiesen. Zunächst werden die wichtigsten Strukturen erklärt um dann zu ausgewählten Ausschnitten aus Funktionen überzugehen.

3.4.1 struct runqueue

Als erstes beschreiben wir die Struktur, welche die Runqueues beschreibt.

```
struct runqueue {
        spinlock_t lock;
```

Der Globale Runqueue-Lock war einer der Gründe für die schlechte Performance des alten Schedulers. Hier sieht man nun, dass im O(1)-Scheduler jede Runqueue einen eigenen Lock hat. Will man mehr als eine Runqueue locken, so gilt die Vereinbarung von *unten* (also bei der ersten CPU) anzufangen, um Deadlocks zu vermeiden.

```
        unsigned long nr_running, nr_switches,
                expired_timestamp, nr_uninterruptible;
```

Diese Membervariablen speichern die Anzahl der laufenden Tasks in der Runqueue, die Anzahl der Prozesswechsel (wird u.a. vom sehr nützlichen Programm **vmstat** gebraucht), den Timestamp des letzten Expires (in Ticks seit Systemstart) und die Anzahl deaktivierten Tasks (ergo Tasks die in den Zustand TASK_UNINTERRUPTIBLE gingen), sobald sie wieder aufwachen wird die Zahl wieder decrementiert.

```
        task_t *curr, *idle;
```

curr ist der aktuelle ausgeführte Task, während idle ein Thread ist, der während des Bootvorgangs erzeugt wurde. Er wird **curr** zugewiesen, wenn die Runqueue leer ist.

```
struct mm_struct *prev_mm;
```

Die **mm_struct**-Struktur des letzten Tasks - bei allgemeinen Betrachtungen des Schedulers eher nebensächlich.

```
        prio_array_t *active, *expired, arrays[2];
```

Dies sind schließlich die Priority-Arrays, deren Aufbau im nächsten Abschnitt erläutert wird. **active** und **expired** werden bei der Initialisation jeweils arrays[0] und arrays[1] zugewiesen und ab da an wird die arrays-Variable nicht mehr direkt angefasst.

```
        int prev_cpu_load[NR_CPUS];
```

Hier werden die „Loads“ vom letzten Balancing gespeichert.

```
        task_t *migration_thread;
        struct list_head migration_queue;
```

Für diese beiden Member-Variablen gilt analoges zu **mm_prev**.

```
        atomic_t nr_iowait;
}
```

Speichert die Anzahl der Tasks, die auf IO warten.

```
static struct runqueue runqueues[NR_CPUS];
```

Dies ist die Definition, die belegt, dass es pro CPU eine Runqueue gibt.

3.4.2 struct prio_array

```
#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+ \
        sizeof(long)-1)/sizeof(long))
```

Dieses Makro berechnet die Größe des Bitmap-Caches Anhand der Anzahl der Prioritäten. Da es 140 Prioritäten gibt, kommt hier dank Ganzzahlarithmetik 5 raus.

```
struct prio_array {
        int nr_active;
```

Die Anzahl der aktiven Tasks.

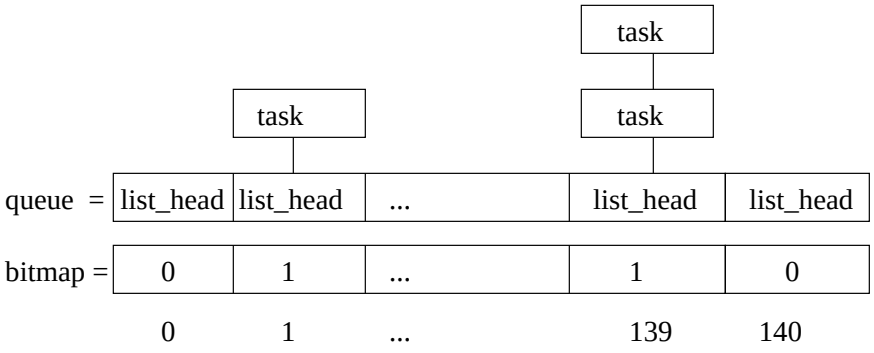
```
        unsigned long bitmap[BITMAP_SIZE];
```

Bitmap-Cache, der mindestens genauso viele Bits hat, wie queue Tasklisten (bzw. Queues wie ich es im restlichen Vortrag nenne) hat.

```
        struct list_head queue[MAX_PRIO];
};
```

Dies sind die eigentlichen Queues, die die Tasks enthalten, deren *dynamische* Priorität dem Index der Queue im Array entspricht. **list_head** ist eine allgemeine im Kernel vorhandene Methode Daten jeder Art in Listen zu verwalten.

Schematisch kann man sich das ganze wie in der folgenden Grafik vorstellen.



3.4.3 enqueue_task()

Nachdem bekannt ist, wie die Runqueues aussehen, stellt sich die Frage, wie ein Task in eine solche Queue hinein kommt:

```
void enqueue_task(struct task_struct *p,
                  prio_array_t *array)
{
        list_add_tail(&p->run_list,
                      array->queue + p->prio);
```

Der Task **p** wird einfach in die Queue eingefügt, deren Nummer seiner Priorität entspricht.

```
        __set_bit(p->prio, array->bitmap);
```

Ganz wichtig, nun wird sichergestellt, dass das Bit, dessen Nummer der Queue entspricht, in die der Task gerade eingefügt wurde, gesetzt ist und so anzeigt, dass diese Queue Tasks enthält.

```
        array->nr_active++;
```

Die Anzahl der Aktiven Tasks erhöht sich durch dieses hinzufügen, also wird die nr_active-Membervariable inkrementiert, damit sie dies reflektiert.

```
        p->array = array;
}
```

Schließlich wird in der Struktur des Tasks gespeichert, welchem Array es angehört.

3.4.4 dequeue_task()

Beim Entfernen läuft das ganze genau umgekehrt ab:

```
void dequeue_task(struct task_struct *p,
                  prio_array_t *array)
{
        array->nr_active--;
```

Die Anzahl der aktiven Tasks vermindert sich und damit auch **nr_active**.

```
        list_del(&p->run_list);
```

Der Task wird aus seiner **run_list** entfernt.

```
        if (list_empty(array->queue + p->prio))
                __clear_bit(p->prio, array->bitmap);
}
```

Es wird überprüft, ob die Queue noch Tasks enthält und falls dem so nicht ist, wird das Bit im Bitmap-Cache gelöscht.

3.4.5 scheduler_tick()

Diese Funktion wird mit jedem Tick ausgeführt und ist nicht mit **schedule()** zu verwechseln. Sie scheduled nicht, sie zählt Zeitscheiben runter und markiert Tasks, dass sie ausgescheduled werden müssen (ich bitte um Vergebung für dieses eklige Denglisch, mir sind jedoch keine deutschen Begriffen bekannt, die eine analoge Bedeutung hätten).

```
void scheduler_tick(int user_ticks,
                    int sys_ticks)
{
[...]
        if (!--p->time_slice) {
```

Es wird also jedes mal die Zeitscheibe decrementiert. Sobald sie 0 erreicht, wird nun der folgende Code ausgeführt.

```
                dequeue_task(p, rq->active);
                set_tsk_need_resched(p);
```

Der Task wird aus der Queue entfernt und markiert, dass bei der nächsten Gelegenheit der Scheduler ausgeführt werden soll (also **schedule()**).

```
                p->prio = effective_prio(p);
                p->time_slice = task_timeslice(p);
                p->first_time_slice = 0;
```

Sofort wird seine neue effektive Priorität und seine Zeitscheibe Berechnet. Diese Funktionen sind im Prinzip nur Makros, die anhand einiger Parameter (dem Laufzeitverhalten insbesondere) den jeweiligen Wert ausrechnen. In **effektive_prio()** findet dann die bereits erwähnte Bestrafung von nicht interaktiven Programmen.

```
[...]
                if (!TASK_INTERACTIVE(p) ||
                    EXPIRED_STARVING(rq)) {
                        if (!rq->expired_timestamp)
                                rq->expired_timestamp =
                                    jiffies;
                        enqueue_task(p, rq->expired);
                } else
                        enqueue_task(p, rq->active);
        }
[...]
}
```

Falls der entfernte Task nun nicht interaktiv (**TASK_INTERAKTIVE** ist erneut ein Makro und versucht festzustellen ob es sich um einen Batch-Job handelt oder um eine interaktive Applikation) ist oder die Gefahr besteht, dass in der Runqueue Tasks verhungern, so wird er in das Array hinzugefügt, auf das derzeit der Zeiger **expired** zeigt.

Ist er hingegen interaktiv und es gibt keine Gefahr von verhungernden Tasks, dann wird er erneut in den **active**-Array hinzugefügt.

3.4.6 schedule()

Jetzt konzentrieren wir uns auf Auszüge der Kernfunktion des Schedulers.

```
void schedule(void)
{ [...]
pick_next_task:
        if (unlikely(!rq->nr_running)) {
                load_balance(rq, 1,
       cpu_to_node_mask(smp_processor_id()));
```

Bei der Auswahl des nächsten Tasks muss zunächst überprüft werden, ob es in der aktuellen Runqueue überhaupt Tasks gibt. Ist dem nicht so, wird versucht dies mit Balancing zu ändern und es wird der folgende Code ausgeführt.

Auffällig an dieser Stelle ist das **unlikely()** - auf den Programmablauf selbst, hat es keine Auswirkungen, hilft jedoch dem Compiler beim optimieren indem es eine Art Branch-Prediction darstellt. Der Compiler optimiert hier den Code für den Fall, dass Queue nicht leer ist, was im laufenden Betrieb auch der Normalfall ist.

```
                if (rq->nr_running)
                        goto pick_next_task;
```

Falls es nun mindestens einen Task gibt, wird zurückgesprungen und neu angefangen.

```
                next = rq->idle;
                rq->expired_timestamp = 0;
                goto switch_tasks;
        }
```

Sonst wird **next** der Idle-Task zugeordnet, das Timestamp neu gesetzt und direkt zum Kontextwechsel gesprungen.

```
[...]
        array = rq->active;
        if (unlikely(!array->nr_active)) {
                rq->active = rq->expired;
                rq->expired = array;
                array = rq->active;
                rq->expired_timestamp = 0;
        }
```

Falls es nun Tasks in der Runqueue gibt muss als nächstes sichergestellt werden, dass auch welche im aktiven Array sind. Ist dem nicht so, werden die Pointer gewechselt. Im Alten Scheduler gäbe es an dieser Stelle eine langwierige Neuberechnung aller Zeitscheiben.

Nun folgt die eigentliche Auswahl des nächsten Tasks:

```
        idx = sched_find_first_bit(array->bitmap);
```

Man besorgt sich den Index der ersten Queue (= der Queue, die Tasks mit der besten effektiven Priorität enthält) indem nach dem ersten gesetzten Bit gesucht wird. Diese Funktion wird noch genauer betrachtet.

```
        queue = array->queue + idx;
        next = list_entry(queue->next, task_t, run_list);
```

Mit diesem Index wird die zuständige Queue berechnet und der Task wird aus ihr geholt. Die letzten drei Zeilen sind die gesamte Logik des Auswählens eines neuen Tasks - der alte Scheduler brauchte hierfür eine Schleife.

```
[...]
        prev = context_switch(rq, prev, next);
[...}
}
```

Schliesslich wird der Kontextwechsel vollzogen.

3.4.7 sched_find_find_bit()

Diese Funktion findet das niedrigste Bit in einem Array aus 5 32Bit Long Words. Die untersten 100 Bit stellen die Prioritäten unterhalb 100 dar und sind somit eher unwahrscheinlich, dass sie im Normalbetrieb auftreten - deshalb sind die Fälle auch so für den Compiler markiert.

```
int sched_find_first_bit(unsigned long *b)
{
        if (unlikely(b[0]))
                return __ffs(b[0]);
        if (unlikely(b[1]))
                return __ffs(b[1]) + 32;
        if (unlikely(b[2]))
                return __ffs(b[2]) + 64;
        if (b[3])
                return __ffs(b[3]) + 96;
        return __ffs(b[4]) + 128;
}
```

Die im Folgenden beschriebene Funktion **__ffs()** durchsucht das übergebene Word, nach dem ersten gesetzten Bit. Effektiv wird also, wie erwartet, in dieser Funktion das erste von 160 Bits (wobei natürlich nur die ersten 140 auftreten können) gesucht und seine Position zurückgeben.

3.4.8 __ffs()

Die folgende Funktion findet das least significant Bit in einem 32Bit Word:

```
unsigned long __ffs(unsigned long word)
{
        __asm__("bsfl %1,%0"
                :"=r" (word)
                :"rm" (word));
        return word;
}
```

Ja, es ist wirklich nur ein Assembler-Befehl um die richtige Queue im Array zu finden.

Dies soll nun als Einblick in den Sourcecode des Schedulers reichen - wer es genauer wissen möchte, möge sich kernel/sched.c näher anschauen.

4 Benchmarks

Nachdem nun das Prinzip sowohl in der Theorie, als auch teilweise in der Praxis, erläutert ist, ist es an der Zeit zu überprüfen, ob der Scheduler hält was er verspricht. Zu diesem Zweck habe ich drei verschiedenen Benchmarks ausgewählt die jeweils etwas anderes messen.

4.1 lat_proc

Dieser Benchmark gehört zu dem LMbench-Tools (http://www.bitmover.com/lmbench/) und wurde von den BitMovers, die sich auch für das exzellente, jedoch nicht freie, Repository-System BitKeeper verantwortlich zeichnen.

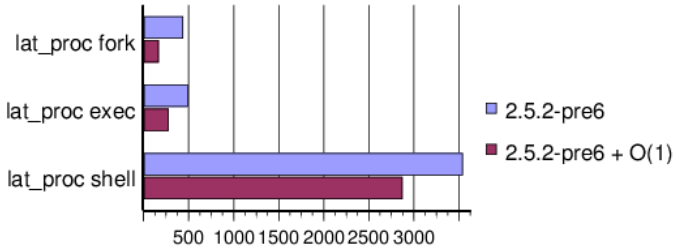
Die lat_proc-Tests messen die Zeit, die ein Programm für ein fork+exit, fork+execve und ein fork+/bin/sh -c.

Es ist leicht einzusehen, das diese Aufgaben in Ihrer Komplexität steigen. Ein exit geht schneller als ein execve und dies wiederum schneller als ein Starten eines Programms über eine Shell, die u.a. Suchpfade durchsuchen muss.

Dieses Benchmark ist insofern relevant, das die Performance dieser Aufgaben sehr wichtig für ein UNIX-System ist, in dem sich Programme ständig forken. Seien es Server wie Apache, inetd oder auch einfach nur die Shell.

Die Angaben im Graphen sind in Mikrosekunden und die Quelle zu diesem Benchmark ist das ursprüngliche Annoucement von Ingo Molnar.

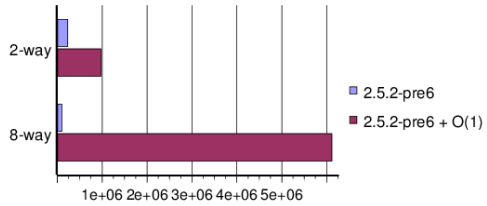
Man sieht ganz klar, dass je mehr die Performance vom Kernel abhängig ist, desto erheblicher der Vorsprung des O(1)-Schedulers. Während bei reinem fork()+exit() der O(1)-Scheduler mehr als doppelt schneller ist, ist es beim /bin/sh -c nur noch um ein Drittel schneller.



4.2 sched_yield

Dieses Benchmark besteht nur aus einem Programm, welches ständig die Funktion **sched_yield()** aufruft. Diese entfernt den aktuellen Task aus dem Active-Array und legt ihn in den Expired-Array. Der Task blockiert also nicht, gibt jedoch den Rest seiner Zeitscheibe auf.

Es werden jeweils doppelt so viele Instanzen wie CPUs gestartet und anschließend die Anzahl der Kontextwechsel/Sekunde mittels **vmstat** ermittelt. Die Angaben im Graphen sind Kontextwechsel/Sekunde und damit misst man direkt die Performance des Schedulers, was jedoch nicht mit seiner Güte zu verwechseln ist. Die Quelle ist erneut das Announcement von Ingo Molnar.



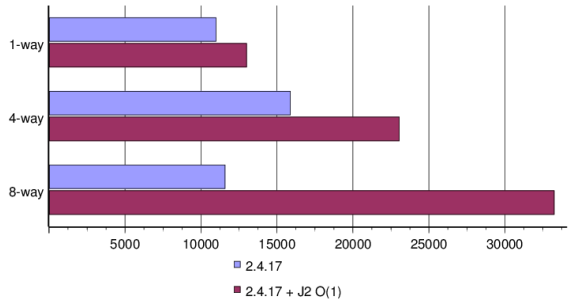
Hier, wo es wirklich nur um die reine Scheduler-Performance geht, sind die Unterschiede zum Teil mehr als Erheblich. Während die Performance des alten Schedulers vom zweifach- auf den achtfach-Rechner sogar einbricht, kommt der O(1)-Scheduler auf dem achtfach-System auf 6 Millionen Kontextwechsel pro Sekunde!

4.3 Volano

Zuletzt noch der Benchmark von Partha Narayanan. Dieser benutzte den Volano-Benchmark (http://www.volano.com/benchmarks.html) - Volano stellt normalerweise Java-Chatsoftware her und deren Benchmark ist hauptsächlich ein JVM-Benchmark, der einen Nachrichten-Durchsatz misst.

Diese Messung fand mit dem „VolanoMark 2.1.2 Loopback test“ auf einem achtfach 700MHZ Pentium III und alle Angaben sind in Nachrichten/Sekunde. Eine URL zu der Quelle ist im Quellen-Kapitel zu finden.

Auch hier stellt man fest, dass das mit dem alten Scheduler das achtfach-System *schlechter* performt, als das vierfach-System, wobei es nur wenig besser ist, als der UP-Rechner. Die Performance des O(1)-Scheduler hingegegen wächst relativ linear mit der Anzahl der CPUs, er skaliert also relativ gut.



5 Probleme

Bei all den Vorteilen darf man nicht vergessen, auch Probleme die sich durch einen radikalen Wandel ergeben zu erkennen und zu lösen. Heute, 1.5 Jahre nach dem ersten Announcement verdient der Scheduler mit Sicherheit die Bezeichnung „stabil“ und soweit erkennbar, gibt es derzeit nur zwei nennenswerte Probleme, die von David Mosberger dokumentiert sind.

5.1 Verhungernde Prozesse

Verhungernde Prozesse sind eigentlich das Schlimmste, was einem Scheduler passieren kann. Es sind Prozesse die aus irgendeinem Grund nicht mehr ausgeführt werden.

Genau dies passiert in diesem Fall, wo ein Prozess fork()-t und das Child 10.000.000 SIGUSR1-Signale an das Parent schickt. Innerhalb weniger Sekunden nach dem Start, reißt das Child sich 99% der CPU-Zeit an sich und das Parent wird praktisch nicht mehr ausgeführt.

Woran dies liegt ist im Augenblick noch unklar, es wird vermutet, dass es an einem Bug in der dynamischen Prioritätsanpassung liegt.

5.2 Übertriebene Affinity

Am Anfang dieses Vortrags wurde erläutert, warum CPU-Affinität auf SMP-System sehr wichtig ist - doch zu viel Affinität erweist sich auch fatal, wie im Folgenden gezeigt werden soll.

Diesesmal erzeugt ein Prozess so viele Unterthreads, wie CPUs vorhanden sind. Diese Threads führen nun jeweils 10.000.000 mal die aus den Benchmarks bekannte Funktion **sched_yield()**. Da dies sofort nach dem Start geschieht, „fällt der Thread dem Scheduler auf“. Aufgrund dieser Tatsache, und der, dass dies immer wieder, in sehr kurzen Abständen geschieht erkennt der Scheduler die Threads als *cache hot* und somit affin zu der aktuellen CPU. Dies geschieht mit alles Threads und als Ergebnis werden alle Threads auf einer einzigen CPU ausgeführt, da das Balancing keine Tasks von einer CPU holt, die *cache hot* sind.

Einzige Lösung dieses Problems ist, dass die Affinitäten auf längere Sicht weniger streng beachtet werden, um eine solche entartete Ausführung auf einem SMP-System zu verhindern.

6 Zusammenfassung

Unter den frei einsehbaren Betriebssystemen hat der O(1)-Scheduler von Linux derzeit keine ernsthafte Konkurrenz. Die Kombination von bekannten Elementen zu diesem Meisterhaften Konzept ist jeden Lob wert. Als einziger Makel bleiben einige Probleme, die aber mit Sicherheit ihre Begründung in der Umsetzung haben und auf Dauer gelöst werden können.

Kurz zusammengefasst ist es sehr empfehlenswert den Scheduler zu benutzen, egal ob auf UP- oder SMP-Rechnern, wobei ein SMP-System mit mehr als 2 CPUs, erst mit dem O(1)-Scheduler sein Geld wert ist.

7 Quellen

- Die Linux Kernel Mailingliste (lkml)

  `http://vger.kernel.org/`

- Understanding the Linux Kernel, 2nd Edition
- Interview mit Ingo Molnar auf KernelTrap.org

  `http://kerneltrap.org/node.php?id=517`

- A closer look at the Linux O(1) scheduler von David Mosberger.

  `http://www.hpl.hp.com/research/linux/kernel/o1.php`

- Ingo Molnars erste Ankündigung des Patches auf der lkml:

  `http://www.ussg.iu.edu/hypermail/linux/kernel/0201.0/0810.html`

- Der Linux Kernel Sourcecode Version 2.5.69. Insbesondere *kernel/sched.c*, sonstige Änderungen sind nur marginal, siehe Patch.
- O(1)-Patches von Ingo Molnar.

  `http://people.redhat.com/mingo/O(1)-scheduler/`

- O(1) Scheduler Benchmarks von Partha Narayanan.

  `http://www.kerneltrap.org/node.php?id=343`