

Taskablaufplanung in Linux (Scheduling)

Reinhard Bündgen
bueندگان@de.ibm.com

Scheduling: Schlagworte

- Präemptives Multitasking vs. kooperatives Multitasking
 - Präemptiv: Zeitscheiben, preemption
 - Kooperativ: yielding
- Scheduling Policies
 - E/A gebunden (oft aber kurz)
 - Prozess gebunden (lang dafür weniger oft)
- Prioritäten
- Echtzeit

Der Linux Scheduler

- Aufgabe:
 - entscheide welche Task als nächstes (wie lange) läuft
- Anforderung
 - schnell, geringer Overhead,
 - auch bei riesigen Task Listen
 - ab Linux 2.5: O(1) Scheduler
 - Optimiere auf üblichen Fall von 1-2 läuffähigen Tasks
 - Bediene wichtige Tasks bald möglichst
 - Linux ist interaktives OS: E/A gebunden
 - Fairness
 - keine Task darf verhungern
 - SMP Scheduling
 - verteile Arbeit auf mehrere CPUs
 - wenn möglich Auslastung aller CPUs
 - unterstütze CPU Affinität: verhindere Cache Thrashing
 - skalierbar

Prioritäten und Zeitscheiben

- Prioritäten
 - Bestimmen welche Task als nächstes „an die Reihe kommt“
 - Hohe Priorität: bald
 - Niedrige Priorität: später
 - Rein prioritätsgetriebener Scheduler: unfair
- Zeitscheiben
 - Bestimmen wie lange Task CPU (maximal) nutzen darf
 - Nach Blockieren darf Zeitscheibenrest genutzt werden
 - Größe der Zeitscheibe abhängig von Priorität
 - Aufgebrauchte Zeitscheiben werden erst erneuert, wenn Scheduling-fairness garantiert ist
- „an die Reihe kommt“ = seine Zeitscheibe aufbrauchen darf

Linux Task Prioritäten

3 Scheduling Policies (`current->policy`)

- Echtzeit
 - `SCHED_FIFO`
 - `SCHED_RR`
 - Prioritätswerte
 - `1 ... MAX_RT_PRIO-1`
 - default `MAX_RT_PRIO = 100`
- Normal
 - `SCHED_NORMAL`
 - Prioritätswerte
 - `MAX_RT_PRIO ... MAX_RT_PRIO + 39` (extern `-20 ... 19`)

Echtzeit-Scheduling

- Weiche Echtzeit gemäss POSIX
- Tasks höherer Priorität (kleinerer Wert) werden
 - vor Echtzeit-Tasks niederer Priorität und
 - vor allen normalen Tasks bearbeitet
- Statische Prioritäten
 - `current->prio == current->static_prio`
- SCHED_FIFO
 - unendlich große Zeitscheibe
- SCHED_RR
 - endliche Zeitscheibe
 - echtes round robin innerhalb einer Priorität (Task bleibt aktiv)

Normales Scheduling: Priorität

- Policy: `SCHED_NORMAL`
- Dynamische Priorität
 - Initial:
 - `current->static_prio`
 - nice-Wert + `MAX_RT_PRIO` + 20
 - Effektiv:
 - `current->prio`
 - `effective_prio()` berechnet Boni und Mali (-5 ... +5) gemäß Interaktivität der Task
 - `current->sleep_avg` (0 ... `MAX_SLEEP_AVG`)
 - Nach Aufwachen um Schlafdauer erhöht
 - Während Lauf mit jedem Timer Tick dekrementiert

Normales Scheduling: Zeitscheibe

- Zeitscheibengröße: abh. von Priorität

Zeitscheibe	Dauer	Interaktivität	Nice Wert
Initial	½ des Elternteils	N/A	Elternteil
Minimum	MIN_TIMESLICE >= 5ms	Niedrig	19
Default	100ms	Durchschnitt	0
Maximum	<= 800ms	Hoch	-20

- Nachdem Task ihre Zeitscheibe verbraucht hat, berechnet `task_timeslice()` neue Zeitscheibe.

Zeitscheibenverfahren

- Alle *aktiven* Tasks in der run-queue verbrauchen ihre Zeitscheibe
 - gemäß Ihrer Priorität
 - Am Stück oder mit Unterbrechungen
- Fairness
 - Sobald Zeitscheibe verbraucht ist, wird Task *inaktiv* („expired“)
 - Alle inaktiven Tasks werden erst wieder reaktiviert, wenn keine aktiven Tasks mehr in der run-queue sind
- Interaktivität
 - Interaktive Tasks dürfen nicht übermäßig lange suspendiert bleiben
 - Können sofort wieder reaktiviert werden, wenn die inaktiven Tasks noch nicht verhungern

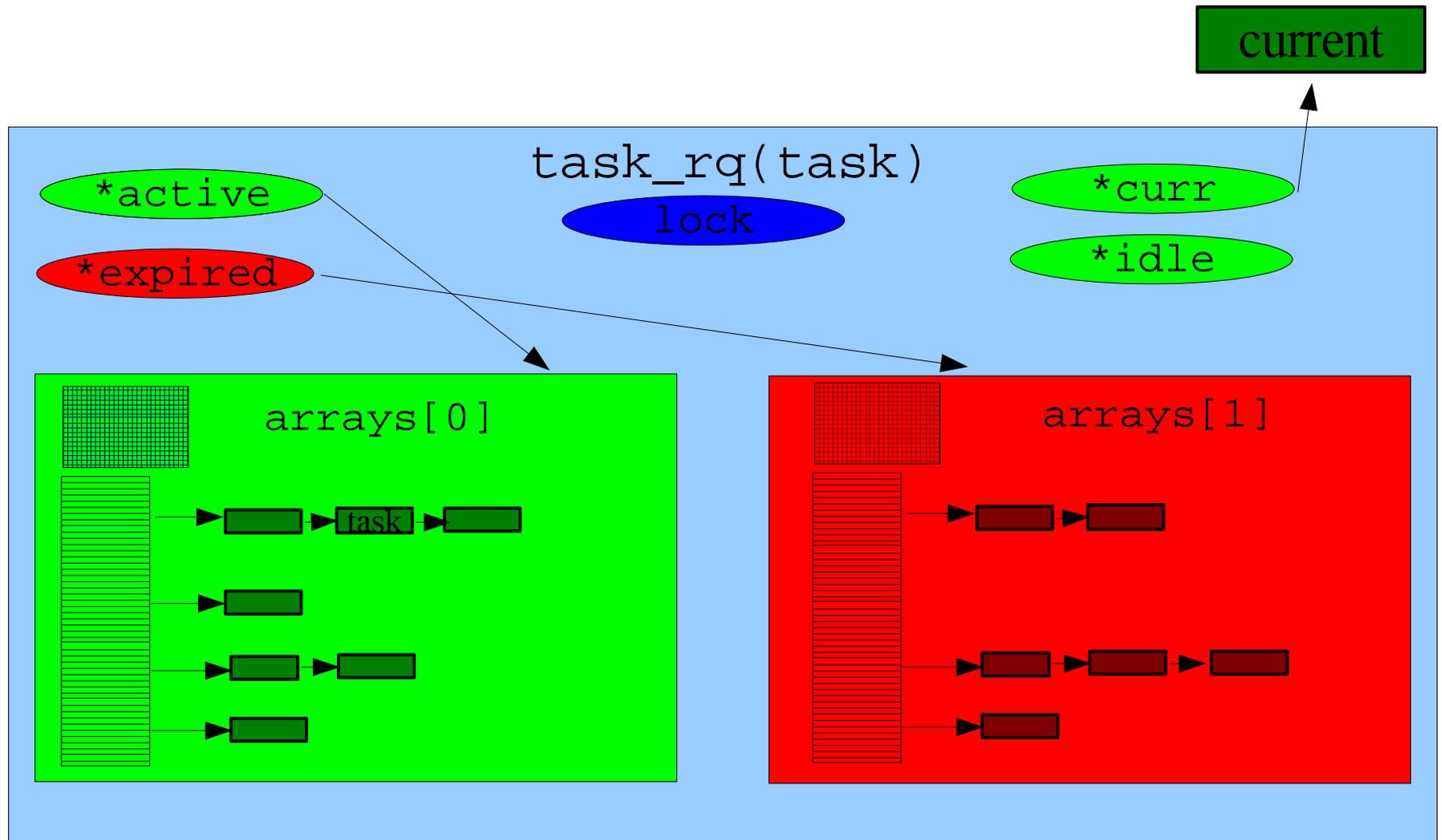
Run Queues

- Struktur `struct runqueue` in `kernel/sched.c`
- Eine pro CPU
 - Die Run Queues enthalten alle Tasks mit Zustand `TASK_RUNNING`
 - Jede Task mit Zustand `TASK_RUNNING` ist in genau einer Run Queue
- Jede Run Queue ist aufgeteilt in 2 SubQueues (`struct prio_array`)
 - aktive SubQueue
 - inaktive (expired) SubQueue
 - Jede SubQueue (aktiv und inaktiv) besteht aus
 - Zähler für die Tasks in der SubQueue
 - ein Array von Queues mit einer Queue für jede mögliche Priorität
 - Einer Bitmap mit einer 1 für jede nicht leere Priorität

Operationen auf Run Queues

- `cpu_rq(processor)`
- `this_rq()`
- `task_rq(task)`
- `task_rq_lock(task, &flags)`
- `task_rq_unlock(task, &flags)`

Active & Expired Tasks in Run Queue



Vertauschen von aktiven und inaktiven SubQueues

- In `schedule()` in `kernel/sched.c`:

```
array = rq->active;
```

```
if (unlikely(!array->nr_active)) {
```

```
    /* Switch the active and expired arrays. */
```

```
    schedstat_inc(rq, sched_switch);
```

```
    rq->active = rq->expired;
```

```
    rq->expired = array;
```

```
    array = rq->active;
```

```
    rq->expired_timestamp = 0;
```

```
    rq->best_expired_prio = MAX_PRIO;
```

```
}
```

Suchen der nächsten Task

- In `schedule()` in `kernel/sched.c`:

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

```
queue = array->queue + idx;
```

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

Schlafen und Aufwachen

- Schlafen
 - Eintrag in eine Warteschlange
 - Setzen des Task Status auf `TASK_INTERRUPTIBLE` oder `TASK_UNINTERRUPTIBLE`
 - Aufruf von `schedule()`
 - Ruft `deactivate_task()` auf
 - entfernt Task von Run Queue
- Aufwachen: `wake_up()` weckt alle Tasks einer Warteschlange via `try_to_wake_up()` auf
 - Setzt Task Status auf `TASK_RUNNING`
 - Aufruf von `activate_task()`:
 - trägt Task in Run Queue ein
 - Setzt `need_resched`, falls Tasks Priorität höher als die von `current`

Context Switch & Preemption

- Context Switch (`context_switch()`) am Ende von `schedule()`
 - `switch_mm()`: tausche virtuelle Speicher mappings
 - `switch_to()`: tausche Prozessorzustand (`thread_info`)
- Wann wird `schedule()` aufgerufen?
 - wenn `need_resched` Flag in `thread_info` gesetzt
 - wenn `scheduler_tick()` oder `try_to_wake_up()`
- Wann kann preemption passieren?
 - User Space
 - bei Rückkehr von Systemruf in User Space
 - bei Rückkehr von Unterbrechung in User Space
 - Kern (wenn `preempt_count = 0` und `need_resched` gesetzt)
 - bei Rückkehr von Unterbrechung in Kernel Space
 - wenn der Kern explizit `schedule()` aufruft
 - wenn eine Task im Kern blockiert

SMP Behandlung

- Load Balancing:
 - Wann?
 - wenn Run Queue einer CPU leer
 - jede ms, wenn System idle
 - alle 200ms
 - `load_balance()`
 - Findet vollste Run Queue mit 25% mehr Tasks als `this_rq()`:
`find_busiest_queue()`
 - Entscheidet von welchem `prio_array` Tasks zu „stehlen“ (`pull_task()`)
- CPU Affinität
 - `current->cpus_allowed`
 - Bitmaske für zugelassene CPUs (ein bit pro CPU)
 - `sched_setaffinity()` / `sched_getaffinity()` Systemrufe
 - Einschränkung für `load_balance()`