

Taskablaufplanung in Linux (Scheduling)

Reinhard Bündgen
bueendgen@de.ibm.com

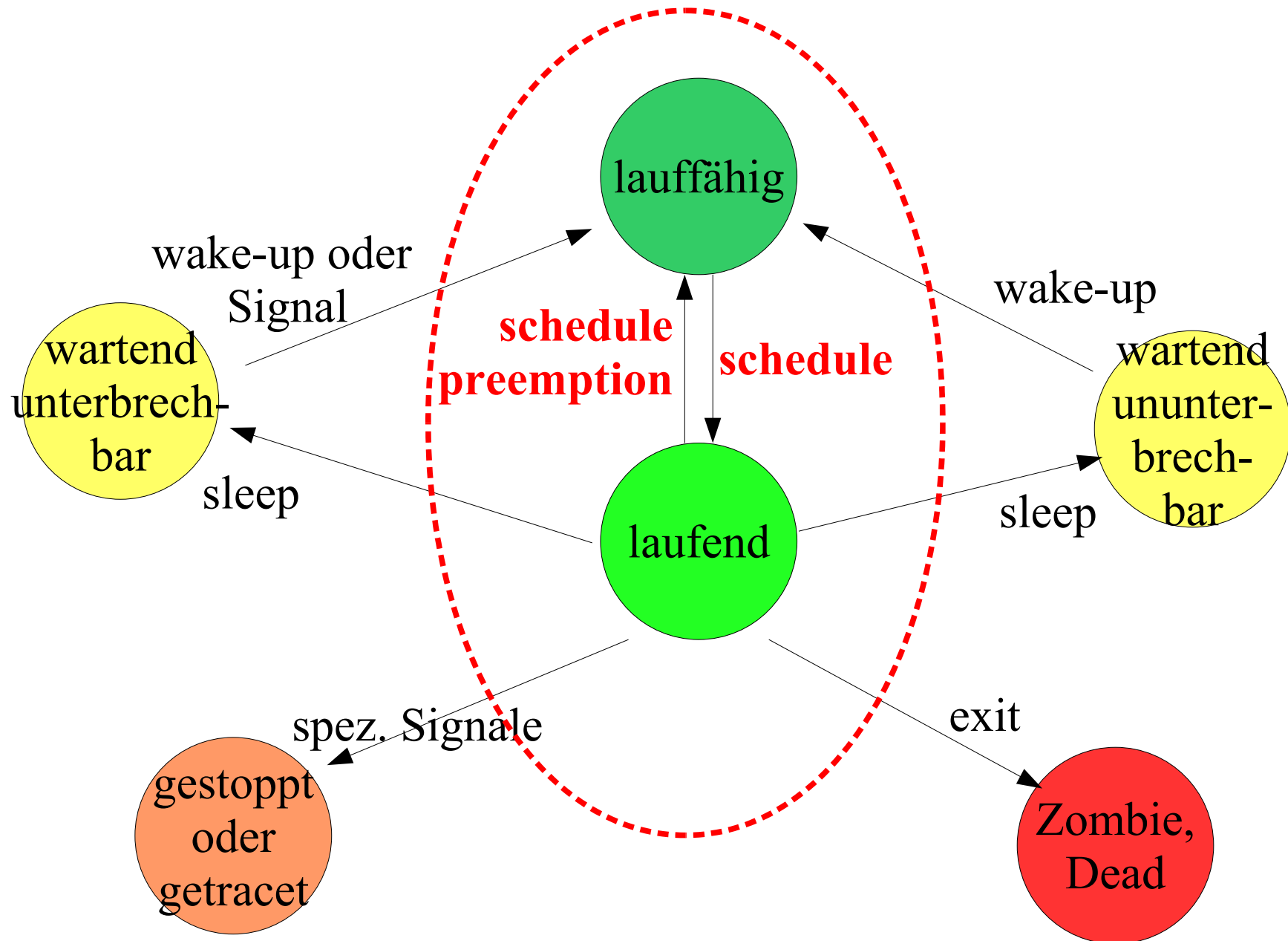
Scheduling: Schlagworte

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

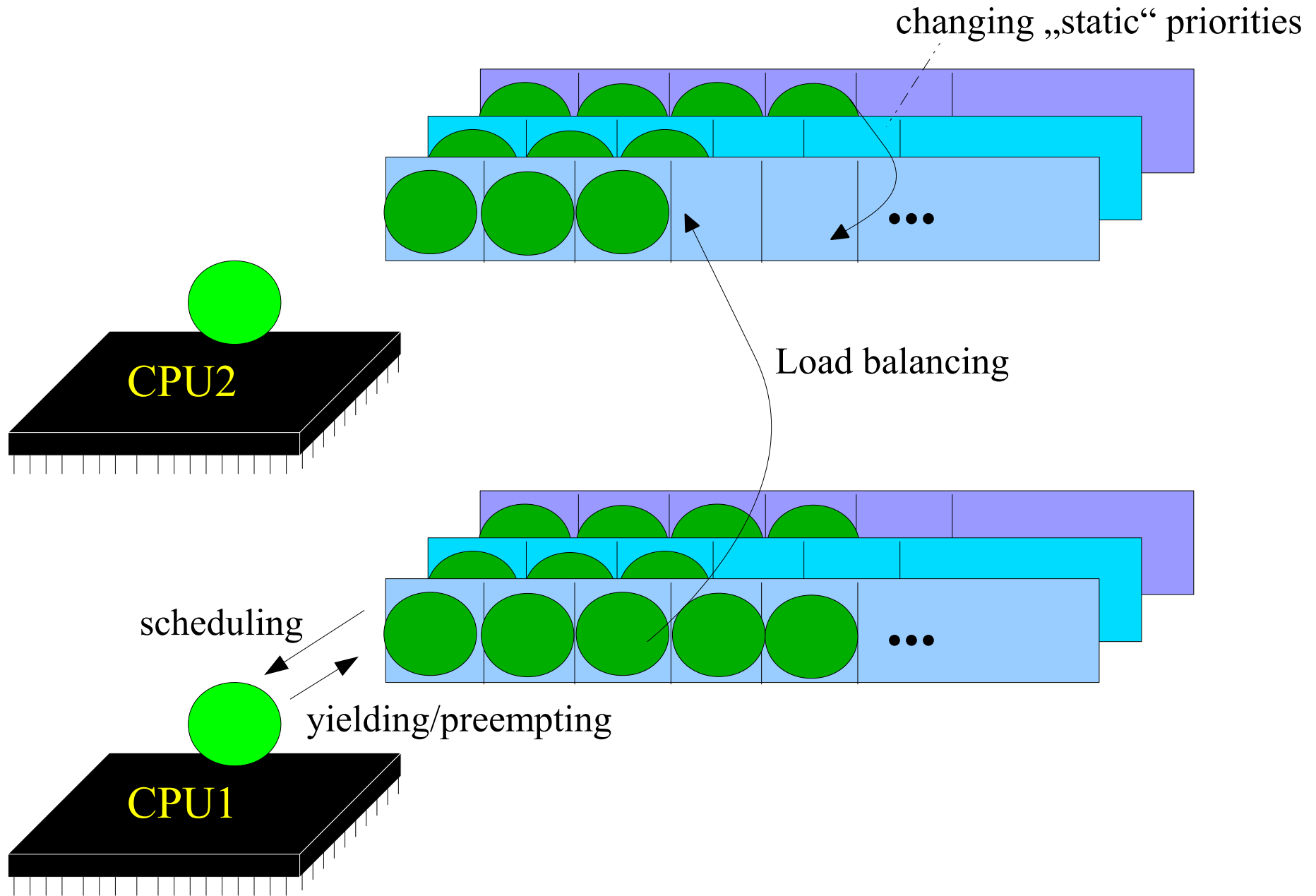
Der Linux Scheduler

- Aufgabe:
 - entscheide welche Task als nächstes (wie lange) läuft
- Anforderung
 - schnell, geringer Overhead,
 - auch bei riesigen Task Listen
 - Optimiere auf üblichen Fall von 1-2 lauffähigen Tasks
 - Bediene wichtige Tasks bald möglichst
 - Linux wird meistens als interaktives OS genutzt : 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

Was macht der Scheduler ?



Run Queues



Scheduling-Parameter

- Schedulingverfahren (policies)
- Statische Prioritäten (der Verfahren)
 - pro CPU eine run queue pro statischer Priorität
 - unfair!
- Nice Werte / Gewichte / Anteile (shares)
 - Bestimmen Gewichtung innerhalb einer Priorität
 - Systemruf `setpriority()`

Scheduling Objekte

- Scheduling entities
 - struct sched_entity in linux/sched.h
 - z.B. current->se
 - haben ein Gewicht (weight)
- Atomare Scheduling Objekte
 - Tasks
- Komplexe Scheduling Objekte
 - Bestehen aus einer Liste von Scheduling Objekten, z.B.
 - Prozesse
 - alle Prozesse eines Users
 - beliebige Gruppen von Prozessen
 - Nicht atomare Scheduling Objekte machen rekursive Ablaufplanung innerhalb des Scheduling Objects
 - Realisiert als control groups (cgroups)

Gruppen Scheduling

- Kernbauoptionen `CONFIG_RT_GROUP_SCHED`,
`CONFIG_FAIR_GROUP_SCHED`
- Gruppierungsmechanismus
 - `CONFIG_CGROUP_SCHED`
 - Basiert auf control groups (siehe `Documentation/cgroups`)
 - cgroup Pseudodateisystem

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/cpu
# mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
```
 - Verzeichnis in `cgroup/cpu` Ordner bezeichnet Scheduling Gruppe:
 - `tasks` – liste der PIDs der Mitgliedsprozesse (`task->css_set`)
 - `cpu.shares` – relativer share der Gruppe zur Gewichts Berechnung
 - `release_agent` - Pfad zu Programm (nur in toplevel cgroup)
 - `notify_on_release` - lasse `release_agent` Programm laufen, wenn letzte Task die cgroup verlässt
 - Zuordnung einer Task zu einer cgroup wird bei `fork/clone` vererbt

Scheduling Verfahren (policy)

- Kann mit `sched_setscheduler()` Systemruf Prozessweise gesetzt werden
- Im Kern als `struct sched_class` implementiert
- Echtzeitverfahren
 - Linux unterstützt nur „weiche“ Realzeit
 - jeweils Priorität 1 - 99
 - implementiert durch `rt_sched_class` (`kernel/sched_rt.c`)
 - Varianten: `SCHED_FIFO`, `SCHED_RR`
- „vollständig faire“ Verfahren
 - jeweils Priorität 0, Nice Werte -20 – 19
 - Implementiert durch `fair_sched_class` (`kernel/sched_fair.c`)
 - Varianten:
 - `SCHED_OTHER` (Kern: `SCHED_NORMAL`)
 - `SCHED_BATCH`
 - `SCHED_IDLE` (Nice Wert > 19)

Completely fair scheduler (CFS) I

- Jedes top level Scheduling Object t hat Gewicht $w(t)$
 - $t \rightarrow se \rightarrow load.weight$
- $T_{rq} = \{ t \mid t \text{ ist lauffähige Task in run queue } rq \}$
 - $cfs_rq \rightarrow load.weight$
- $W(T_{rq}) = \text{Summe } \{ w(t) \mid t \text{ in } T_{rq} \}$
- Schedulingperiode: P (default 20ms)
- Ideale Zeitscheibe für t : $slice(t) = P * w(t) / W(T_{rq})$
- Run queue: implementiert als red black tree
 - Ordnungsschlüssel: $vruntime$
 - $vruntime$: mit Gewicht normalisierte bisherige Laufzeit eines Scheduling Objektes
 - merkt sich $min_vruntime$

Exkurs: Red Black Trees

- Semi-balancierter binärer Suchbaum
 - längster Pfad im Baum maximal 2 mal so lang wie kürzester Pfad
 - schnelles Einfügen und Löschen ($O(\log n)$)
 - siehe Wikipediaeintrag für Red Black Trees
- Benutzung im Linux Kern
 - CFS run queues, diverse I/O Scheduler, hochauflösender Timer, ext3fs, VMA tracking, epoll fds, ...
- `linux/rbtree.h`
 - `struct rb_node`
 - `struct rb_root`
 - `rb_entry(ptr, type, member)` analog zu `list_entry()`
 - Funktionen für parent, color, next, prev, first, last, replace, erase
 - Keine(!) Funktionen: für search, insert (siehe Kommentar & Beispiel am Anfang von `rbtree.h`)

Completely Fair Scheduler (CFS) II

- Annäherung an ideale Zeitscheibe für Scheduling Objekt t :
- Wiederhole
 - lasse t eine kurze Zeit auf laufen (ns timer)
 - aktualisiere $vrtime$ von t
 - sei t_0 das erste Scheduling Objekt in der Run Queue
 - wenn $vrtime(t) > vrtime(t_0) + \text{delta}$ dann
 - Füge t in run queue ein
 - entnimm t_0 der run queue
 - Setze $t = t_0$,

RT Scheduler

- Reines prioritätsgesteuertes FIFO (SCHED_FIFO) oder Round-Robin (SCHED_RR)
- Systemweite Attribute
 - `/proc/sys/kernel/sched_rt_period_us`
(default 1s)
 - `/proc/sys/kernel/sched_rt_runtime_us`
(default 0,95s, -1 ist unbegrenzt)
 - $(\text{sched_rt_period_us} - \text{sched_rt_runtime_us}) / \text{sched_rt_period_us}$ CPU-Anteil ist für CFS reserviert
- CONFIG_CGROUP_SCHED (geplant?)
 - `.../<cgroup>/cpu.rt_runtime_us`

Systemrufe zur Prozessablaufplanung

```
int getpriority(int which, int who);  
int setpriority(int which, int who, int prio)  
int nice(int inc)    // library function in glibc  
int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);  
int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);  
int sched_setparam(pid_t pid, const struct sched_param *param);  
int sched_getparam(pid_t pid, struct sched_param *param);  
int sched_get_priority_max(int policy);  
int sched_get_priority_min(int policy);  
int sched_rr_get_interval(pid_t pid, struct timespec * tp);  
int sched_setscheduler(pid_t pid, int policy, const struct  
                                sched_param *param);  
int sched_getscheduler(pid_t pid);  
int sched_yield(void)
```

Schlafen und Aufwachen

- Schlafen
 - Eintrag in eine Warteschlange
 - Setzen des Task Status auf `TASK_INTERRUPTIBLE` oder `TASK_UNINTERRUPTIBLE`
 - Aufruf von `schedule()`
 - Ruft `deactivate_task()` (`dequeue_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 `enqueue_task()`:
 - trägt Task in Run Queue ein

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?
 - am Ende von Systemrufen oder Unterbrechungen, wenn `need_resched` Flag in `thread_info` gesetzt
 - wenn `sys_yield()` oder `try_to_wake_up()`
- Wann kann Präemption 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

Anhang:

$O(1)$ Scheduler
seit 2.6.23 durch CFS abgelöst

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 eine Task die 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 | 1/2 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.

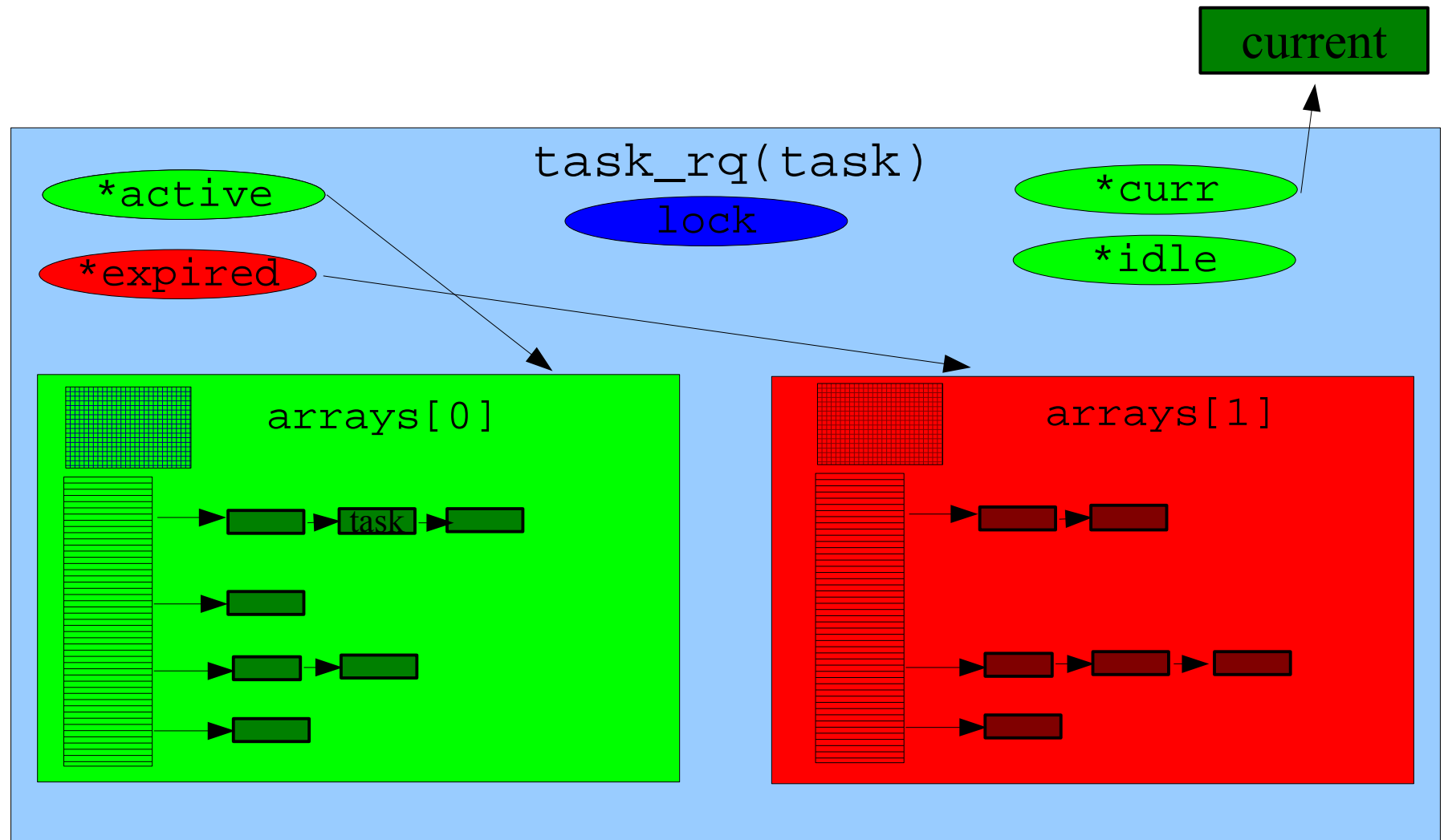
Zeitscheibenverfahren

- 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 Bitman mit einer 1 für jede nicht leere Priorität

Aktive & Expired Tasks in Run Queue



Operationen auf Run Queues

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

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);`
-

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()`