Linux 2.4 Scheduling (1)

- **General characteristics**
  - Linux offers three scheduling algorithms.
    - A traditional UNIX scheduler: SCHED_OTHER
    - Two "real-time" schedulers (mandated by POSIX.1b): SCHED_FIFO and SCHED_RR
  - Linux scheduling algorithms for real-time processes are "soft real-time".
    - They give the CPU to a real-time process if any real-time process wants it.
    - Otherwise they let CPU time trickle down to non real-time processes.
  - Here, we study the scheduling algorithm implemented in the Linux 2.4.18 kernel.

Linux 2.4 Scheduling (2)

- **Priorities**
  - Static priority
    - The maximum size of the time slice a process should be allowed before being forced to allow other processes to complete for the CPU.
  - Dynamic priority
    - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.
    - When its dynamic priority falls to 0, the process is marked for rescheduling.
  - Real-time priority
    - Only real-time processes have the real-time priority.
    - Higher real-time priority values always beat lower values.

Linux 2.4 Scheduling (3)

- **Related fields in the task structure**

`long counter;`	time remaining in the task's current quantum (represents dynamic priority)
`long nice;`	task's nice value, -20 to +19. (represents static priority)
`unsigned long policy;`	SCHED_OTHER, SCHED_FIFO, SCHED_RR
`struct mm_struct *mm;`	points to the memory descriptor
`int processor;`	processor ID on which the task will execute
`unsigned long cpus_runnable;`	~0 if the task is not running on any CPU (1<<cpu) if it's running on a CPU
`unsigned long cpus_allowed;`	CPUs allowed to run
`struct list_head run_list;`	head of the run queue
`unsigned long rt_priority;`	real-time priority

Linux 2.4 Scheduling (4)

- **Scheduling policies**
  - SCHED_OTHER
  - SCHED_FIFO
    - A real-time process runs until it either blocks on I/O, explicitly yields the CPU, or is preempted by another real-time process with a higher rt_priority.
    - Acts as if it has no time slice.
  - SCHED_RR
    - It's the same as SCHED_FIFO, except that time slices do matter.
    - When a SCHED_RR process's time slice expires, it goes to the back of the list of SCHED_FIFO and SCHED_RR processes with the same rt_priority.

Linux 2.4 Scheduling (5)

- **Scheduling quanta**
  - Linux gets a timer interrupt or a *tick* once every 10ms on IA-32. (HZ=100)
    - Alpha port of the Linux kernel issues 1024 timer interrupts per second.
  - Linux wants the time slice to be around 50ms.
    - Decreased from 200ms (in v2.2)

```
/* v2.4 */
#if HZ < 200
#define TICK_SCALE(x)    ((x) >> 2)
#endif
#define NICE_TO_TICKS(nice)    (TICK_SCALE(20-(nice))+1)

/* v2.2 */
#define DEF_PRIORITY     (20*HZ/100)
```

Linux 2.4 Scheduling (6)

- **Epochs**
  - The Linux scheduling algorithm works by dividing the CPU time into epochs.
    - In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
    - The epoch ends when all runnable processes have exhausted their quantum.
    - The scheduler recomputes the time-quantum durations of all processes and a new epoch begins.
  - The base time quantum of a process is computed based on the nice value.

Linux 2.4 Scheduling (7)

- **Selecting the next process to run**

```
repeat_schedule:
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
            if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu,
                                      prev->active_mm);
                if (weight > c)
                    c = weight, next = p;
            }
     }
```

Linux 2.4 Scheduling (8)

- Recalculating counters

```
if (unlikely(!c)) {          /* New epoch begins … */
    struct task_struct *p;

    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p)
        p->counter = (p->counter >> 1) +
                     NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
```

Linux 2.4 Scheduling (9)

- **Calculating goodness()**

```
static inline int goodness (p, this_cpu, this_mm) {
    int weight = -1;
    if (p->policy == SCHED_OTHER) {
        weight = p->counter;
        if (!weight) goto out;
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 – p->nice;
        goto out;
    }
    weight = 1000 + p->rt_priority;
out:    return weight;
}
```

`weight = 0`
p has exhausted its quantum.

`0 < weight < 1000`
p is a conventional process.

`weight >= 1000`
p is a real-time process.

Linux 2.4 Scheduling (10)

- **Sharing counters in fork()**

```
/*
"share" dynamic priority between parent and child, thus
the total amount of dynamic priorities in the system
doesn't change, more scheduling fairness. This is only
important in the first timeslice, on the long run the
scheduling behaviour is unchanged.
*/
    p->counter = (current->counter + 1) >> 1;
    current->counter >>= 1;
    if (!current->counter)
        current->need_resched = 1;
```

Linux Scheduler Evolution

- **Scalability problems in Linux 2.4 scheduler**
  - A single run queue
  - Recalculating goodness() for every task on every invocation of the scheduler
- **Linux 2.6: O(1) scheduler (prior to 2.6.23)**
- **The idea of "fair scheduling" by Con Kolivas**
- **CFS (Completely Fair Scheduler) by Ingo Molnar**
  - Many ideas borrowed from the Kolivas' scheduler
  - Officially included in Linux 2.6.23
  - Made Kolivas leave Linux kernel development
  - Kolivas returns with BFS (Brain Fuck Scheduler) in 2009