Ausgewählte Betriebssysteme

Preemption and Low-Latency patches

Scheduling

What is scheduler latency ?

- 'the interval between stimulus and response' (webster.com)
- Linux: time between a wakeup signaling that an event has occurred and the kernel scheduler runs the now runnable activity
- Wakeups are often caused by interrupts
  - Thread induced wakeups possible, too.

Components of response time

- Interrupt latency
  - Time between physical signal and start of interrupt handler execution
- Interrupt handler duration
- Scheduler latency
  - Time spent after completion of IRQ handler and invocation of scheduler
  - Might be non-existent on SMP (real parallelism)
- Scheduling duration
  - Time spent in the scheduler

Why does latency matter ?

- Some applications depend on timely execution
- Delays devaluate computation
- Wide variety of examples
  - Process controlling
    - CD burning
    - Flight control
  - Multi media
    - MPEG playback
      - Delays result in jerks

Preemption patches

- Run the scheduler more often
  - If there might be the need to run the scheduler
  - Minimize the time until the scheduler runs
  - Preempt the kernel if this is safe
- Linux Kernel originally not preemptable
  - Only interrupts and bottom halves were allowed to run asynchronously
  - No synchronization primitives necessary for data that is not modified by IRQ and BH
  - In SMP this requirement is burdensome and partially lifted

Preemption patches (2)

- assumption that code does not rely on non-preemption
  - SMP requires this anyway
- Kernel can be preempted if not holding spin locks
  - Holding spinlocks signals exclusive access
    - If neglected
      - Deadlocks
      - Priority inversion
- Run the scheduler if needed when
  - Return from IRQ
  - Releasing spinlock
- No further code modification (besides making it SMP safe)
- Mitigates scheduler latency problem

Low latency patch (1)

- Explicit preemption points
- Processing large data structures

```
set_lock()
do_all_work()
release_lock()
```

→

```
redo:
   set_lock()
   do_some_work()
   get_into_consistent_state()
   release_lock()
   if not done:
      goto redo
```

Low latency patch (2)

- Work intensive
  - Find long-lasting spots
    - In many short loops it is not obvious how large that processed amount of data is
  - Support by special tools
    - Andrew Morton's `rtc-debug`
- Error prone
  - Find a consistent state that allows reentrant code
  - Ensure Progress
    - Starvation might be possible otherwise

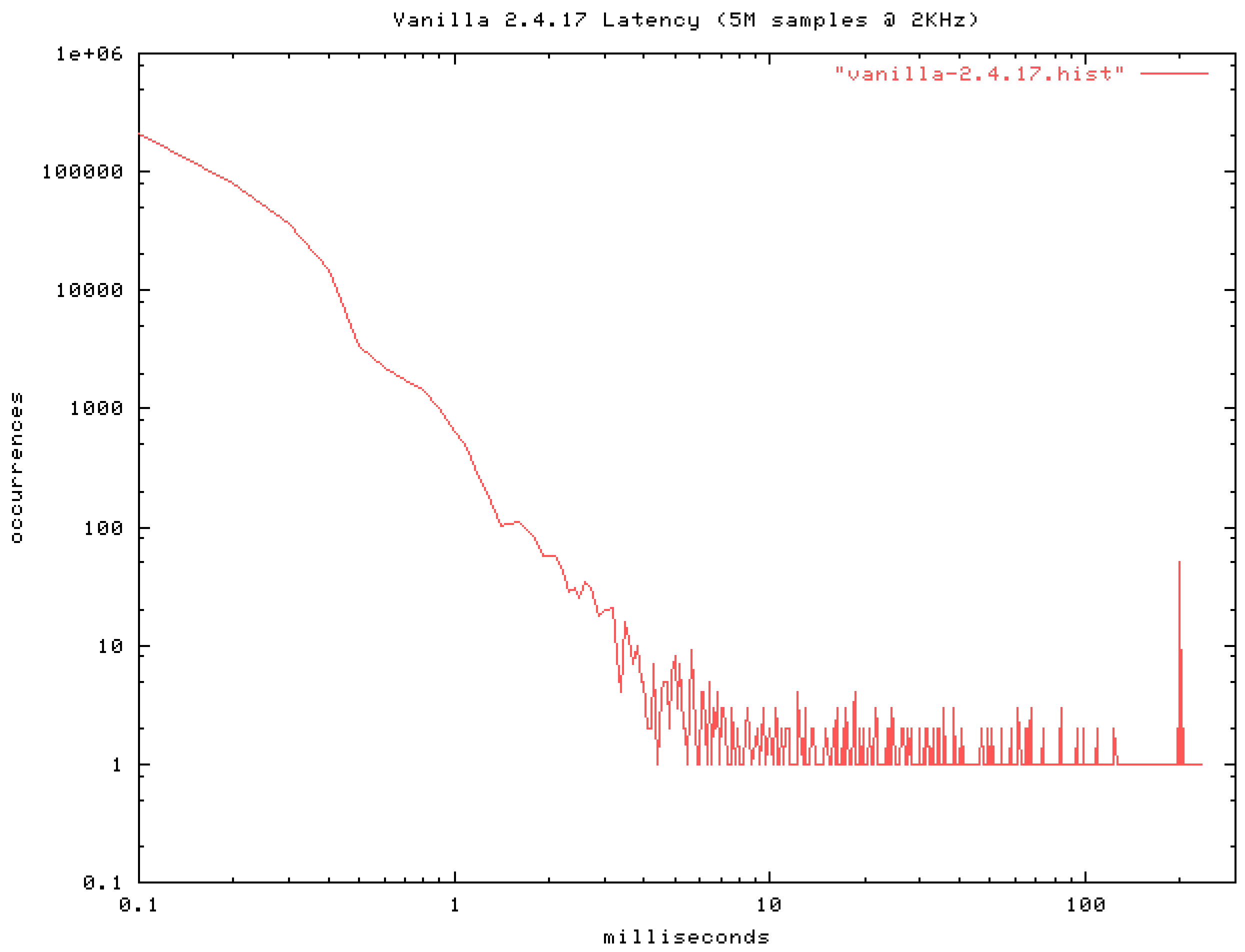
Iterating over infinite data

```
void prune_dcache(int count)
{
    spin_lock(&dcache_lock);
    for (;;) {
        struct dentry *dentry;
        struct list_head *tmp;

        tmp = dentry_unused.prev;
        if (tmp == &dentry_unused)
            break;
        list_del_init(tmp);
        dentry = list_entry(tmp, struct dentry, d_lru);

        /* If the dentry was recently referenced, don't free
it. */
        if (dentry->d_vfs_flags & DCACHE_REFERENCED) {
            dentry->d_vfs_flags &= ~DCACHE_REFERENCED;
            list_add(&dentry->d_lru, &dentry_unused);
            continue;
        }
        dentry_stat.nr_unused--;

        /* Unused dentry with a count? */
        if (atomic_read(&dentry->d_count))
            BUG();

        prune_one_dentry(dentry);
        if (!--count)
            break;
    }
    spin_unlock(&dcache_lock);
}
```

Adding a preemption point

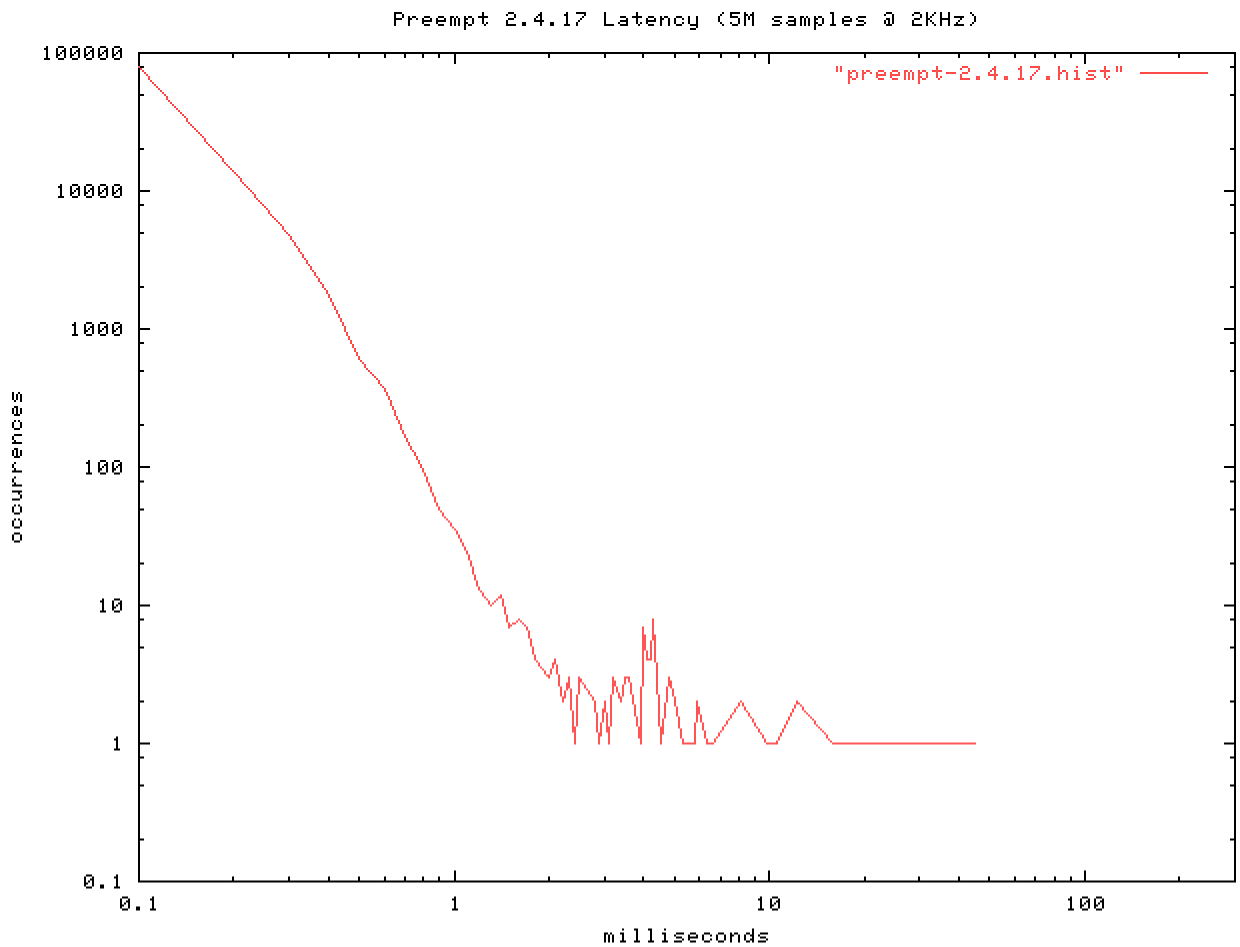
```
void prune_dcache(int count)
 {
      DEFINE_RESCHED_COUNT;
 redo:
      spin_lock(&dcache_lock);
      for (;;) {
          struct dentry *dentry;
          struct list_head *tmp;
          if (TEST_RESCHED_COUNT(100)) {
              RESET_RESCHED_COUNT();
              if (conditional_schedule_needed()) {
                 spin_unlock(&dcache_lock);
                 unconditional_schedule();
                 goto redo;
              }
          }

          tmp = dentry_unused.prev;

          if (tmp == &dentry_unused)
              break;
          list_del_init(tmp);
          dentry = list_entry(tmp, struct dentry, d_lru);

          /* If the dentry was recently referenced,
              don't free it. */
           if (dentry->d_vfs_flags & DCACHE_REFERENCED) {
               dentry->d_vfs_flags &= ~DCACHE_REFERENCED;
               list_add(&dentry->d_lru, &dentry_unused);
               continue;
           }
           dentry_stat.nr_unused--;

           /* Unused dentry with a count? */
           if (atomic_read(&dentry->d_count))
               BUG();

           prune_one_dentry(dentry);
           if (!--count)
               break;
      }
      spin_unlock(&dcache_lock);
 }
```

Vanilla Linux 2.4.17



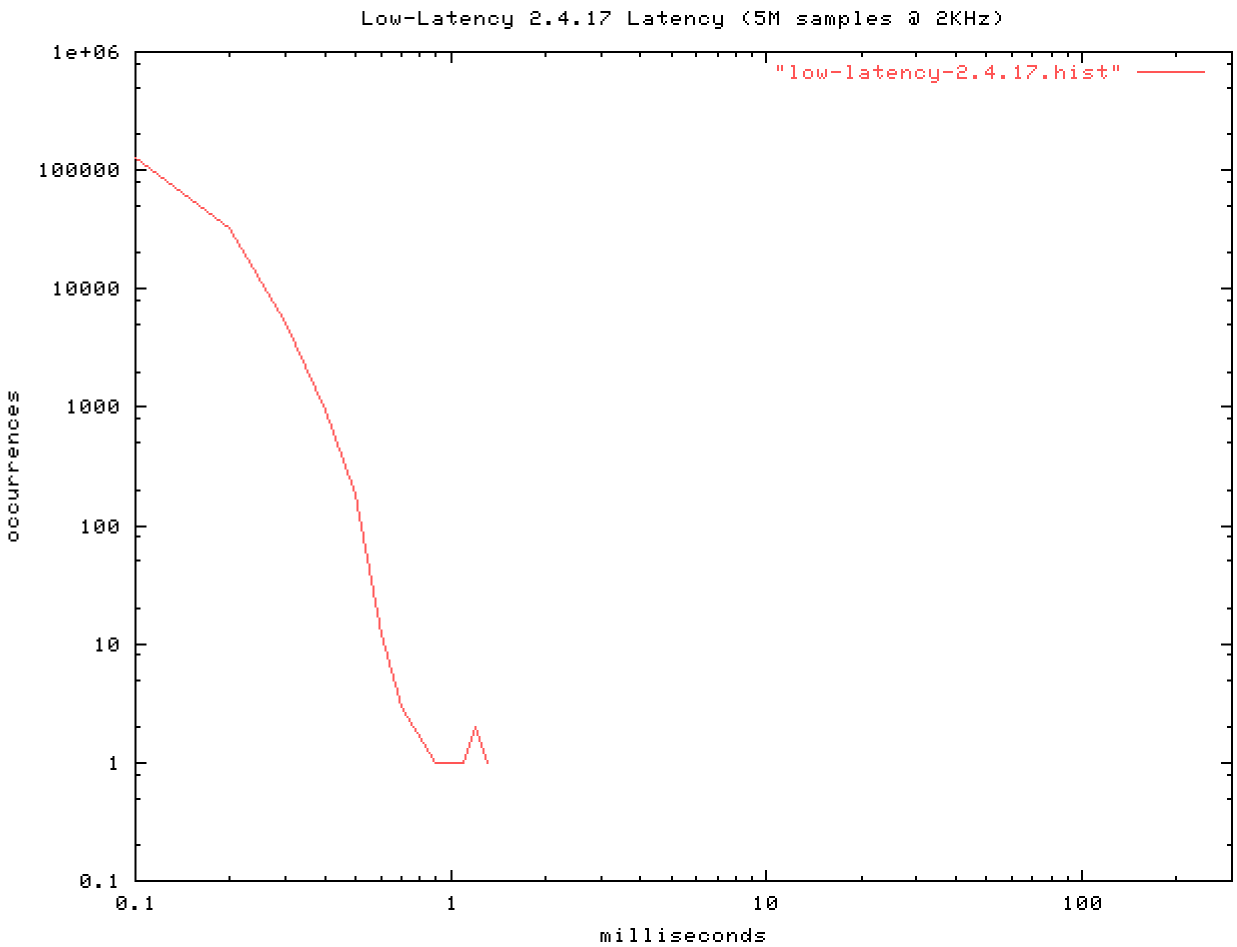
Max. latency: 232.7ms

Linux 2.4.17 + preemption patches



Max. latency: 45.3ms

Linux 2.4.17 + low latency patches



Max. latency: 1.4ms

Max. latency in FIASCO: <30?s

The old scheduler

- Features to keep
  - Good interactivity under high load
  - Good performance with few runnable tasks
  - Fairness
  - Support of priorities
  - SMP
    - efficiency
      - No idling cpu with runnable tasks in the system
    - affinity
      - Goodness takes last running process into account

Implementation of the old scheduler

- Time divided into epochs
- Each task gets a quantum per epoch
  - Based on static priority
  - Quantum grows if not exhausted in previous epochss
    - Interactivity boost
- Scheduler selects task with highest goodness
  - Calculation of goodness of all runnable processes must be done for each scheduling decision
    - Cache pollution on different CPU
  - all CPUs fetch tasks from one global queue
    - Contention
    - Automatic load balancing

Insufficiencies

- Duration of scheduling grows with number of processes
  - Iteration over all runnable processes to find maximal goodness
- Missing SMP scalability
  - Only one global runqueue
  - Random bouncing
    - Processes with expired quantum are marked unrunnable until all processes of the epoch finished
- No fixed cpu affinity

O(1) scheduler

- Runqueue per CPU
  - Two priority-sorted arrays (active, expired)
    - Transfer exhausted task from active to expired array
    - Switch arrays if all tasks have expired
  - 64bit bitfield for efficient lookup of highest available priority with runnable threads
    - No goodness calculation necessary

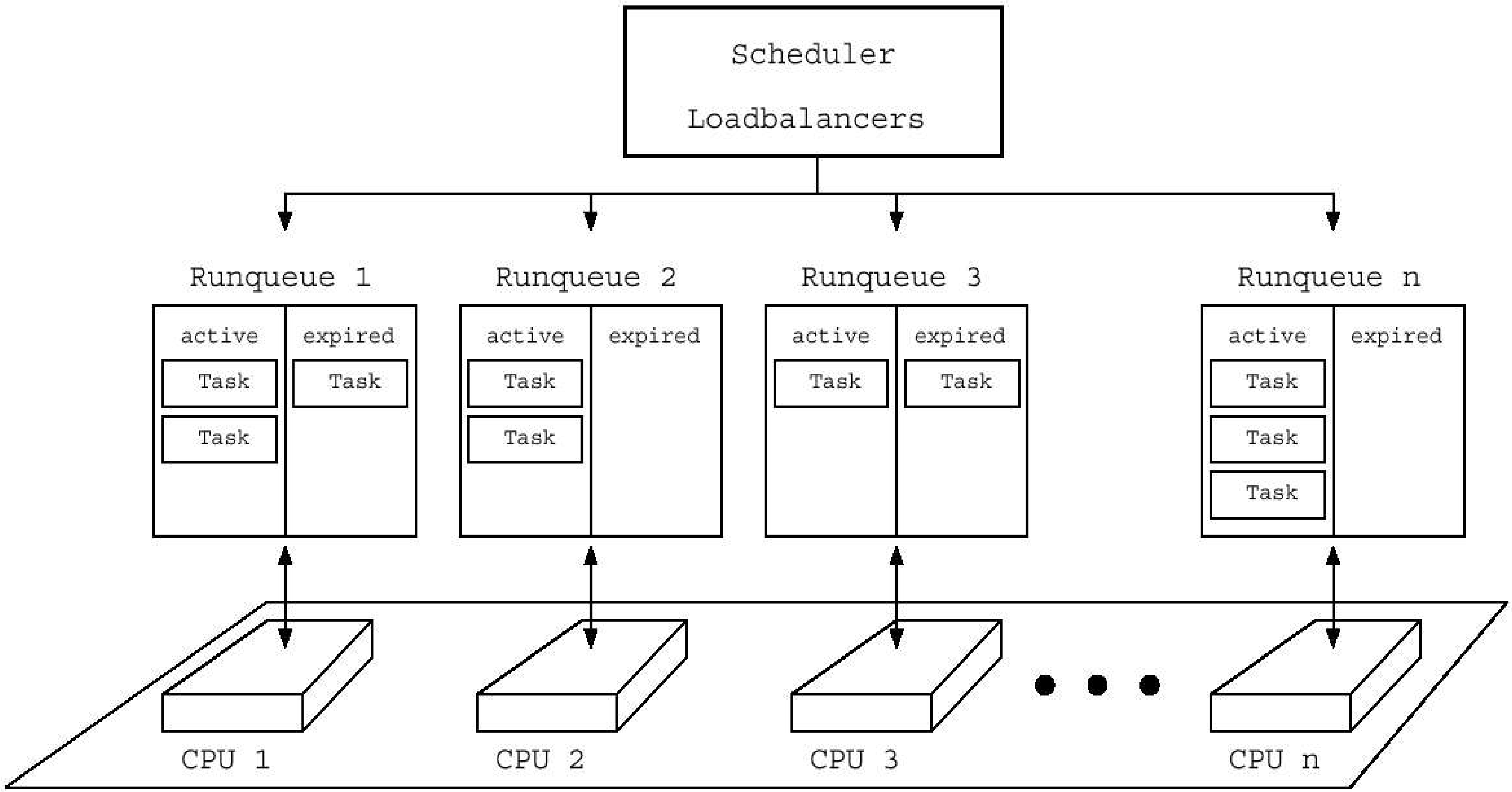
Handling interactivity

- Depending on the sleeping behavior a classification of interactive /non-interactive task is done
  - Empirical based on “good interactive feeling”
- Priority change [-5, +5]
- Interactive tasks are not transferred into expired array, but scheduled again
  - Lose interactivity classification if not sleeping anymore

Load balancing

- No automatic load balancing due to global queue any longer
- Load-balancing kernel thread per CPU
  - Activation depending on load situation
    - Immediately if idle
    - Every 250ms if running tasks are available
  - Tries to fetch tasks from heavily loaded other CPUs
    - From expired array
    - If runnable on destination CPU (affinity is user defined)
    - Avoid task with hot cache working set

O(1) Scheduler



Performance

- 20% better in chatserver benchmark
- Significant more context switches
  - Important for highly threaded systems
  - 300% more on 2 way system
  - 60 times more on a 8 way system
- Better fork() performance
  - 25% - 100% gain
  - Runs childs before parents
    - Saves copy-on-write when execing immediately