

Alternative Scheduling in Linux

Alina Beygelzimer

November 30, 1999

Normal Scheduling under Linux

- `schedule()` function (`kernel/sched.c`)
- Linux supports three scheduling classes:
 - `SCHED_OTHER`
 - `SCHED_FIFO`
 - `SCHED_RR`
- All processes regardless of the scheduling policy are placed on a **single run queue** (doubly linked by `prev_run` and `next_run` components of the task structure)
- Components of the `task_struct` related to scheduling:
 - `long counter;`
 - `long priority;`
 - `unsigned long policy;`
 - `unsigned long rt_priority;`
 - `long need_resched;`
- Several system calls allow the policy and scheduling parameters to be modified (`setpriority`, `nice`, `sched_setparam`, etc.)

The scheduler gets called from

- some system calls (usually indirectly by calling `sleep_on()`)
- `ret_from_sys_call` (if `need_resched` flag is on)

```
static void update_process_times(unsigned long ticks) {  
    ...  
    struct task_struct *p = current;  
    p->counter -= ticks;  
    if (p->counter < 0) {  
        p->counter = 0;  
        p->need_resched = 1;  
    }  
    ...  
}
```

What does `schedule()` do?

- checks to make sure it hasn't been called from an interrupt handler
- does "administrative" work: calls the bottom halves of the interrupt routines and routines registered for the scheduler task queue:

```
if (tq_scheduler)
    run_task_queue(&tq_scheduler);
if (bh_mask & bh_active)
    do_bottom_half();
```

- Determines the fate of the current process:

If the current task belongs to the `SCHED_RR` class and its time slice has expired, it is placed at the end of the run queue.

```
struct task_struct *prev = current;
if (prev->policy == SCHED_RR && !prev->counter) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

If the current task is either finished or blocked and is un interruptible, it is removed from the run queue.

If the current task is blocked but can be reactivated by signals, the scheduler checks to see if any signals have been sent to it. If so, it is made runnable.

- If the current task simply ran out of time, nothing is done to it.

```

switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:
}

```

- Next, the scheduler goes through all the tasks in the run queue and computes the goodness of each task.

```

int goodness (struct task_struct *prev,
              struct task_struct *p)
{
    int weight;
    if (p->policy != SCHED_OTHER) {
        weight = 1000 + p->rt_priority;
    } else {
        weight = p->counter;
        if (weight && (p == prev))
            weight += 1;
    }
    return weight;
}

```

- `repeat_schedule:`

```

next = &init_task;
c = -1000;
while (p = init_task.next_run; p != &init_task) {
    weight = goodness(prev, p);
    if (weight > c) {
        c = weight; next = p;
    }
    p = p->next_run;
}
if (!c) {
    for_each_task(p)
        p->counter = (p->counter / 2) + p->priority;
    goto repeat_schedule;
}

```
- `if (prev != next) {`

```

    get_mmu_context(next);
    switch_to(prev, next);
}

```

See

```

include/asm-*/mmu_context.h:get_mmu_context()
include/asm-*/system.h:switch_to()

```

Scheduling real-time tasks. Why Linux can't handle hard real-time tasks and why simple fixes don't work

- Linux is a fair time-sharing system. It is *designed* to optimize the *average* performance and to try to give every process a fair share of compute time. A process can be preempted at an unpredictable moment and forced to wait for its share.
- Linux VMS also adds unpredictability: pages belonging to any process can be swapped out to disk at any time (although now you can lock pages in memory...)
- Linux processes are heavyweight.
- Kernel processes are non-preemptive. Real-time processes cannot get scheduled while the kernel works on behalf of even the least important process.
- Linux kernel uses disabling interrupts as a means of synchronization. A real-time interrupt may be delayed until the current process, no matter how unimportant, finishes its critical section.

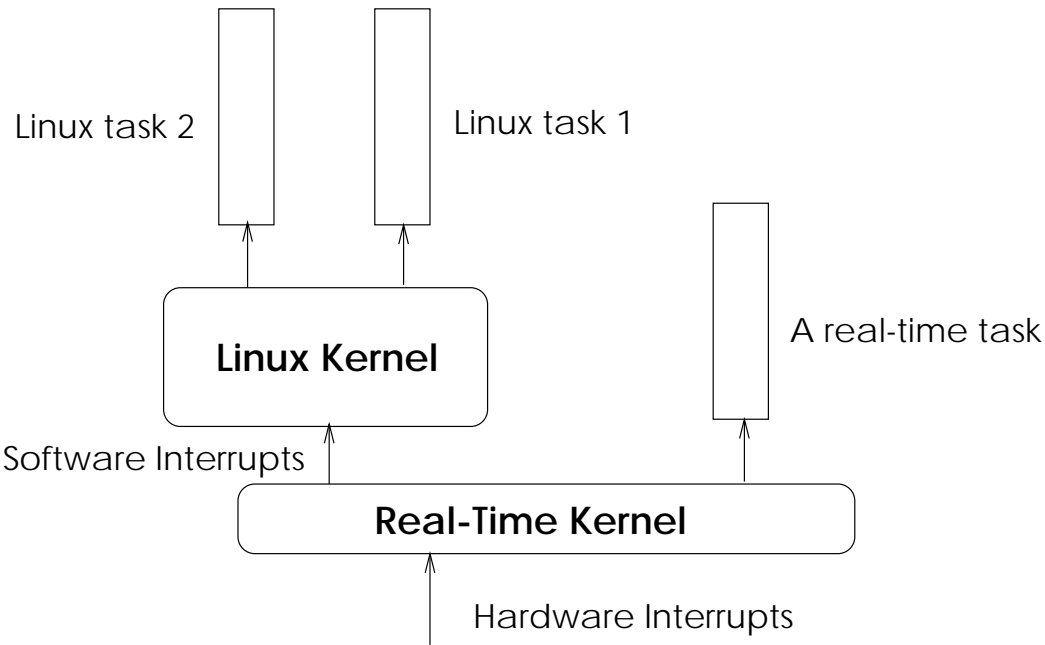
Distortions in scheduling due to blocked interrupts

Category	min	mean	max
Console (TTY) code	0.34	0.60	8.79
Network device driver	0.33	0.37	3.26
Disk service driver and buffer management code	0.35	34.68	407.62
Scheduling	0.33	1.11	43.66
Timer handling	0.35	2.33	52.98
Memory management	0.34	0.60	10.31
Other	0.33	0.39	82.61

Duration (in microseconds) for which interrupts are disabled in the Linux Kernel*

* Courtesy of KURT developers (University of Kansas)

RT-Linux (Yodaiken, Barabanov)



Alternative scheduling algorithms

- Priority scheduling
- Fair-share scheduling
- Real-time scheduling
 - Static table-driven scheduling
 - Priority-driven preemptive scheduling (Liu and Layland, 73)
 - * Rate-monotonic scheduling (static)
 - * Statistical rate-monotonic scheduling (Atlas and Bestavros, 98)
 - * Earliest deadline scheduling (dynamic)
 - Dynamic planning-based scheduling
- Proportional-share scheduling (Waldspurger, 94)
 - Lottery scheduling (randomized, absolute and relative error $O(\sqrt{n_a})$)
 - Multi-winner lottery scheduling (hybrid)
 - Stride scheduling (deterministic, absolute error $O(n_c)$, relative error 1)
 - Hierarchical stride scheduling (deterministic, absolute error $O(\log n_c)$, relative error 1)
- Microeconomic scheduling

Proportional-share scheduling. Framework

- Ticket abstraction
- Ticket transfers
provide a convenient solution to priority inversion problem. Unlike priority inheritance, transfers from multiple clients are additive.
- Ticket inflation and deflation
maybe allowed only within trust boundaries that can be defined using ticket currencies.
- Ticket currencies

Lottery Scheduling

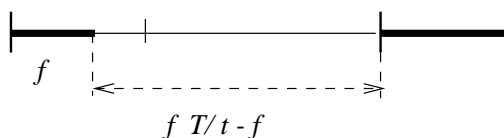
Each allocation is determined by holding a *lottery* that randomly selects a winning ticket. The resource is granted to the client holding the winning ticket. This effectively allocates resources to competing clients *proportionally* to the number of tickets they hold.

Nonuniform Quanta

When a client consumes a fraction f of its allocated quantum, its resource share is adjusted by $1/f$ until the client starts its next quantum.

Proof. Consider a client that owns t of the T tickets. Client is given extra $t/f - t$ *compensation tickets* for using a fraction f of its allocated quantum. This changes its overall ticket value to t/f .

Without this compensation, the client's expected waiting time would be $T/t - 1$. With compensation, it becomes $(T + t/f - t)/(t/f) - 1 = fT/t - f$. The client's expected resource usage is f quanta over a time period consisting of $f + (fT/t - f) = fT/t$ quanta. Thus, the client receives a resource share of $f/(fT/t) = t/T$, as desired. \square



Performance of Lottery Scheduling

$p = t/T$. After n_a allocations, the expected number of wins w for client c is $E[w] = n_a p$ with variance $\sigma_w^2 = n_a p(1 - p)$. The coefficient of variation $\sigma_w/E[w] = \sqrt{(1 - p)/np}$. Thus, the expected throughput error for a client is $O(\sqrt{n_a})$. Client's throughput is proportional to its ticket allocation with accuracy that steadily improves when error is measured as a percentage of n_a . Nevertheless, *the absolute value of the error is unbounded*.

The response time has a geometric distribution. The expected number of lotteries l that a client must wait before completing its first win is $E[l] = 1/p$ with variance $\sigma_l^2 = (1 - p)/p^2$. The coefficient of variation $\sigma_l/E[l] = \sqrt{1 - p}$. Thus, the response time variability depends only on the client's relative share of tickets.

Stride Scheduling

Idea: compute the time interval (*stride*) that a client must wait between successive allocations. The client with the smallest stride will be scheduled most frequently. The client's *stride* is inversely proportional to *tickets*. The client's *pass* represents the time for the client's next selection. The client with the minimum *pass* is selected, and its *pass* is advanced by its *stride*.

Performance of Stride Scheduling

For skewed ticket distributions it is possible for a client to have $O(n_c)$ absolute error. Nevertheless, stride scheduling is considerably more accurate than lottery scheduling since the error does not grow with the number of allocations.

Hierarchical Stride Scheduling Apply stride scheduling recursively. Individual clients are combined into groups. An allocation is performed by invoking the normal stride scheduling algorithm first among groups, and then among individual clients within groups. This allows to reduce the error to $O(\log n_c)$.