OS Linux

Scheduling in 3.x Kernels

Denny

The Linux Operating system can execute multiple processes simultaneously, although only one process can be actually executed by a processor at an instance. Multiprocessors allows multiple tasks to run in parallel. There additionaly are processes that are sleeping or waiting to be killed. The part of the kernel, which is responsible for granting CPU time to tasks, is called process scheduler.

Location of a Scheduler in Linux

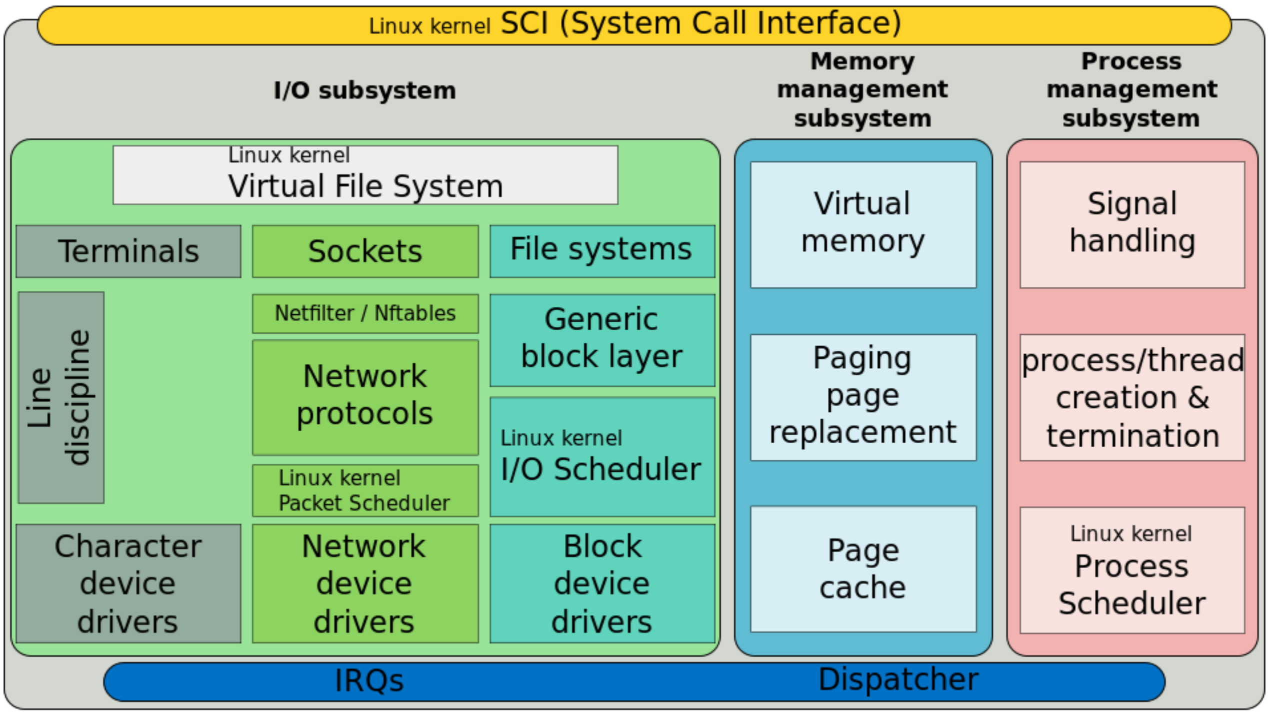


Image courtesy Wikipedia

Priority

Priority shows the importance of a certain task. One of the factors that determine how much running time a task gets and how often it is preempted is priority.

Types of Priority

Static Priority Type:

- Value used by the scheduler to rate the process with respect to the other processes in the system
- For example: static priority of normal non realtime processes is a number ranging from 100 (highest priority) to 139 (lowest priority);
- time quantum duration assigned to the process depends on its priority

Dynamic Priority Type:

- Is the number actually looked up by the scheduler when selecting the new process to run
- Is also a number ranging from 100 (highest priority) to 139 (lowest priority); One example of such a situation is when the system lifts a task's priority to a higher level for a period of time, so it can preempt another high-priority task

Process Terminologies

Active :

Allowed to run processes with unexhausted time quantums

Expired:

Not allowed to run processes sicne they have already used their time slices. They will be run after they are made to wait for a long time .

Interactive process:

Interacts with the user hence must have stricter scheduling requirements. They can be active or expired.

Batch Process:

These processes such as compilation, data fetch can have looser constraints.

Real Time processes:

These are associated with a special real-time priority, where the values range from 1 (highest priority) to 99 (lowest priority).

IO-bound

One that depends on constant interrupt from the IO devices.

Processor-bound

A processor-bound task is the one in which the instruction sequence is executed consecutively on the processor until it is either preempted or finished.

Scheduling Policies

Policies basically mean special scheduling decisions for a group of processes such as longer time slices, higher priorities, etc. Following are the scheduling policies currently

• SCHED_NORMAL: the scheduling policy that is used for regular tasks;

• SCHED_BATCH: does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs (CPU-intensive batch processes that are not interactive);

• SCHED_IDLE: Is a very low priority though not an idle task. Some background system threads obey this policy, mainly not to disturb normal way of things;

• SCHED_FIFO and SCHED_RR are for soft real-time processes. Handled by real-time scheduler in and specified by POSIX standard. RR implements round robin method, while SCHED_FIFO uses first in first out queuing mechanism.

CFS

The scheduler used in Linux is called a Completely Fair Scheduler (CFS) which is essentially an O(log n) algorithm that depends on a red-black tree. The Completely Fair Scheduler aims to be fair to all different priority processes by giving appropriate time slices to each process depending on their run time values stored.

CFS uses a time-ordered red black tree to build a "timeline" of future tasks. The elements of the red black tree are the processes or tasks keyed with the value of their run times. The smaller the run time key, the more to the left of the tree a node is. The scheduler always picks the leftmost node as the next task to run. CFS uses priority as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay and vice versa.

A scheduler using CFS does the scheduling during a timer interrupt. During an interrupt, the scheduler examines the current state of affairs and can preempt the current task if it used all its time slice or there is a task with smaller virtual runtime in the tree or even if there is a newly created task.

Internals

The main files concerned :

Linux/kernel/sched.c,

Linux/kernel/sched/fair.c

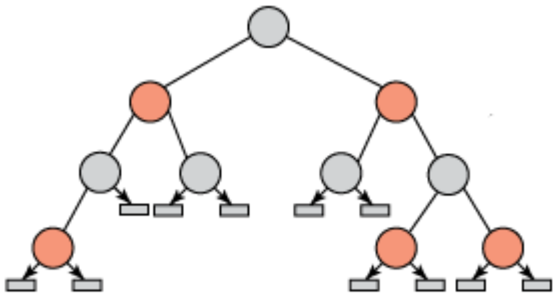
- task_struct

Each process has a task structure associated with it that defines all the properties of the process including details such as its static priority , scheduling class and associated policies, state, cpu associated, time slice etc.(linux/sched.h)

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info thread_info;
#endif
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;

#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
        unsigned int cpu;       /* current CPU */
```

- runque structure

In process scheduling runqueues are the central data structure which holds all the tasks in a runnable state. Each CPU will have an associated run queue. Every run queue there is will have some processes in them linked together as a list. No process will be lying in two runqueues. In CFS model the runqueue is not structured as the usual linear queue rather ,as discussed, a Red Blue Tree



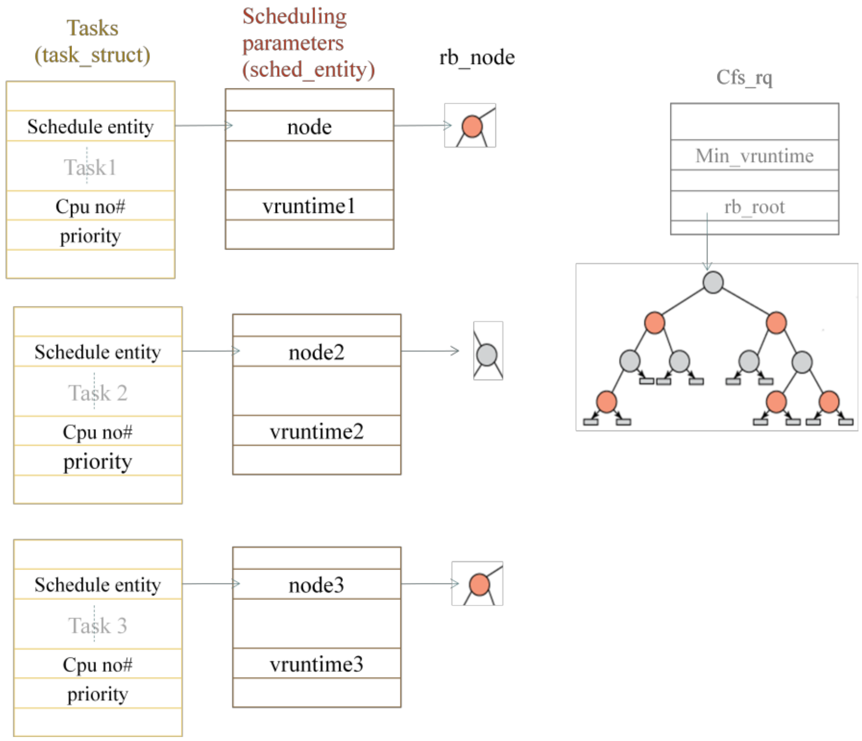
The rbtree data structure is represented by struct cfs_rq.

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
        struct load_weight load;
        unsigned long nr_running;

        u64 exec_clock;
        u64 min_vruntime;

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;

        struct list_head tasks;
        struct list_head *balance_iterator;


        struct sched_entity *curr, *next, *last, *skip;

        unsigned int nr_spread_over;

#ifdef CONFIG_FAIR_GROUP_SCHED
        struct rq *rq;  /* cpu runqueue to which this cfs_rq is attached */


        int on_list;
        struct list_head leaf_cfs_rq_list;
        struct task_group *tg;  /* group that "owns" this runqueue */

```

- sched_entity

For each task the scheduling parameters are summarized as a structure which is named *sched_entity*. Details such as node position, timestamp started, current and previous run times, load weight etc are contained in this structure.

As shown in the following figure the associations between tasks , scheduling entities and the rbtree is as below



Scheduling Function

The schedule function will be either called right away if a process need to be blocked immediately if suppose some resource is not available. Also it is called when a process runs out of its time slice. The following are the steps for scheduling a task.

When the linux timer interrupts , the scheduler_tick () is called. During each time of the scheduler tick , the run time is subtracted for the current process in the runqueue. This is taken care of by the scheduler_tick() function's update_rq_clock() call. The initial time slice can be set during the time of process creation or can be changed dynamically as discussed earlier.

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;

        sched_clock_tick();

        raw_spin_lock(&rq->lock);
        update_rq_clock(rq);
        curr->sched_class->task_tick(rq, curr, 0);
        cpu_load_update_active(rq);
        calc_global_load_tick(rq);
        raw_spin_unlock(&rq->lock);

        perf_event_task_tick();

#ifdef CONFIG_SMP
        rq->idle_balance = idle_cpu(cpu);
        trigger_load_balance(rq);
#endif
        rq_last_tick_reset(rq);
```

The schedule() function is the core area of the Linux scheduling process.(kernel/sched.c)

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;
```

The current CPU, the runqueue of the current CPU and the current process from the runqueue are identified initially as can be ssen in the figure.

```
cpu = smp_processor_id();
rq = cpu_rq(cpu);
rcu_note_context_switch(cpu);
prev = rq->curr;
```

Next the current process is put back to the rbtree with a call to put_prev_task() . Later the next task is taken from the rbtree with the call to pick_next_task().[function internally calls pick_next_task_fair() which is defined in sched/fair.c]

```

        put_prev_task(rq, prev);
        next = pick_next_task(rq);
        clear_tsk_need_resched(prev);
        rq->skip_clock_update = 0;

```

In essense a task with the smallest virtual run time is to be selected next. For which this function simply picks the left-most task from the red-black tree and returns the associated *sched_entity* .

```
                rq->curr = next;
                ++*switch_count;

                context_switch(rq, prev, next); /* unlocks the rq */
                /*
                 * The context switch have flipped the stack from under us
                 * and restored the local variables which were saved when
                 * this task called schedule() in the past. prev == current
                 * is still correct, but it can be moved to another cpu/rq.
                 */
```

Up next the context_switch() function is called for restoring the stack to the version that was earlier saved by this new task when it got preempted earlier.