Scheduling

Periodically task switches occur. It is the job of the scheduler to pick the next task to run on a given CPU. The scheduler must do so *fairly*, balancing several needs. Although scheduling algorithms, policies and implementations vary greatly among operating systems and are the subject of many research papers, we can generalize a few broad principles:
• CPU time is a scarce resource for which tasks compete.
• Each task has an "importance", or **static priority**, which can be configured directly or indirectly by the system administrator.
• Tasks should appear as responsive as possible to interactive events. E.g. when a key is pressed or the mouse is moved, the application should respond quickly.
• Real-time tasks which must handle events within a fixed time interval must be given the right-of-way when those events happen.
• Tasks tend to be either **compute-bound** or **I/O-bound**. The former spend most of their time computing and thus have a heavy appetite for user-mode CPU time. The latter spend most of their time waiting for I/O. These classical definitions are often stressed by media applications, e.g. a streaming video server which is both I/O bound with network traffic and compute-bound with compression and decompression algorithms. Compute time among CPU-bound processes should be fairly distributed so that jobs complete in a reasonable time.
• The scheduler system itself should have a low overhead. The context switch is not a good time to be executing complicated, long-winded algorithms.

The UNIX model of priority

In the classic UNIX priority model, the static priority of a process is represented by a **nice value**, ranging from -19 to +20, which weights the schedulers allocation of CPU time among competing tasks. The nice value is by default 0. Positive "nice" values give a task poorer static priority, i.e. we are being "nice" to other processes on the system. Conversely, negative "nice" values give a task better than average static priority. Only the superuser can give a task (including itself) a negative nice value, but any task can give itself a positive value. (In more modern kernels, the privilege to give negative nice values is fine-grained and not necessarily linked to having uid==0).

This counter-intuitive inversion of the sense of priority is historic and ingrained in the UNIX/POSIX standards. In recent Linux kernels, the static priority levels range from 0 to 99, with 0 being the lowest (worst) and 99 being the best. These priority levels are not the same as the nice value.

Static priority levels from 1 to 99 are reserved for **real time tasks**. These tasks are scheduled strictly by static priority, the thinking being that a more important real-time task, such as the control rod algorithm in an nuclear plant, should always run when it has to, even if a less important task (such as the plant's payroll system) has been waiting a really, really long time for some CPU. The static priority level of 0 is used for "normal" (non real-time) tasks, and the nice value is used to establish a static sub-priority within the class of non real-time tasks.

Real-Time Tasks

Real-time tasks under LINUX are assigned to scheduler class `SCHED_RR` or `SCHED_FIFO`. A SCHED_FIFO task will continue to run until it either sleeps, is pre-empted by another real-time task with higher static priority, or voluntarily yields (`sched_yield` system call).

A SCHED_RR task runs in "round-robin" fashion. It is given a certain fixed **Quantum**, or **Time-Slice**, which is configurable on a per-task basis. When it has run for that amount of time, it will yield, allowing another SCHED_RR task at the same priority level to run. SCHED_RR tasks will also be pre-empted by real-time tasks with higher priorities that become ready to run, and will always pre-empty lower priority tasks.

Historically only the superuser could establish real-time tasks. Under modern Linux kernels, this capability is regulated by the rlimit_rtprio resource limit.

Quantum

The amount of time that a task has the CPU before being rescheduled is known as the Quantum or the Time Slice. Scheduling algorithms vary. The time slice could be completely static, determined dynamically when the task is scheduled in and then not re-evaluated until the next time it gets the CPU, or re-evaluated at every scheduler tick.

The Old Linux Scheduler vs the CFS scheduler

The Linux scheduler which was in use until a few years ago was called the O(1) scheduler. It followed the classic UNIX model in using an array of linked lists, the array being indexed by the dynamic priority level, which ranged from 1 to 140, with 140 being the *worst*.

This scheduler is presented as an appendix, as it highlights some interesting design trade-offs. The advantage of the O(1) scheduler was that both selection of the next process and insertion of the process into the run queue array was constant time. Therefore not much CPU was being spent at each time slice tick towards scheduler computation.

It was found that in order to satisfy all of the needs of scheduling, in particular balancing CPU vs I/O bound processes, a number of "kluges" had crept in, and the O(1) scheduler did not perform well when scaled to large numbers of cpu-bound processes intermixed with interactive processes with a need for quick response time. This lead to the development of what the Linux kernel people call the "Completely Fair Scheduler (CFS)". CFS departs from traditional scheduler practice in that the time slice is not pre-determined when the task is first scheduled, but is re-evaluated at each clock tick. This results in more work at each tick, but with CPU clock rates breaking through the multi-GHz level, while clock ticks remain at 1 msec, the percent overhead for this approach has been less important.

Ideal and practical latency

In the ideal world, if there were 4 runnable processes of the same priority ("nice" level) they should all receive equal distribution of CPU time, and would have the smallest possible **latency**. Latency is defined as how much time elapses from when a task is pre-empted to when it gets the CPU again.

As a practical matter, the latency has a lower bound, because otherwise the system would spend most of its time in task switches, instead of doing useful work. There is a system tunable parameter (it can be set via `sysctl`) called `sched_latency_ns` which defines the target ideal latency period. It is by default 5 msec. So if there were 5 runnable processes, each would get 1msec of CPU and then be pre-empted. This would satisfy the 5msec latency.

However, as the number of runnable processes increases, the cost of making the time slice ever smaller to satisfy the target latency would become prohibitive. Therefore there is a cutoff value of minimum time period which is by default 1msec, which also happens to be the default clock tick time period. So if there are 10 runnable tasks and the target latency is 5msec, the time slice is not 500nsec, but 1msec.

Weighted timeslice

The "nice" value, under the Linux CFS scheduler, is a process scheduling "weight". There are 39 nice steps (-20 to +19). Each nice step represents a 10% relative difference in CPU allocation. (This is a purely Linux interpretation of nice values -- other operating systems may have very different policies).

Let’s say process A has a nice value of 0 and process B has a nice value of 1, and these are the only two runnable processes. Then the weight WA=1.00, and WB=0.800. We define the "load weight" LW as the sum of the weights of all runnable tasks. In this case, LW=1.800. Then the CPU allocation of any task n is w=Wn/LW. For A this is 1.00/1.800=55.5% and for B = 0.800/1.800 = 44.4%. Thus we see that the difference between two tasks separated by one nice level is approximately 10%. This formula is logarithmic. If A had a nice value of -20 and B +19, A would get 99.9% of the CPU and B would get just 0.1%.

If the target latency is L, then the target timeslice period P=L/NR, subject to the lower bounding discussed above. Let’s say L=20 msec and the above tasks A and B are the only runnable tasks so NR=2. Then P=10ms. The weighted timeslice of a task n is Sn=P*Wn/LW. Therefore, A would have a weighted timeslice of 5.55msec, and B would get 4.45msec.

This algorithm extends trivially to any number of runnable processes, and insures that the targeted weighted timeslice of any given process is its "fair share" of the available CPU, considering all of the other runnable processes and their weights. In actual kernel implementation, the weights are coded as integers, rather than the floating-point numbers used above, because the kernel avoids the use of the floating point registers.

Virtual Runtime

Under the CFS scheduler, the figure of merit when comparing runnable processes for scheduling is unfortunately called *virtual runtime*, or vruntime. Like many things in the Linux kernel, this is poorly named, and perhaps would better be called the "weighted actual runtime share".

The idealized allocation of timeslice presented in the previous section can not be realized in practice because 1) pre-emption only happens during a scheduler tick, which has a granularity of (typically) 1 msec, and 2) while a task is running, other tasks awaken, changing the weighted load.

Whenever the scheduler tick occurs, the vruntime of the running process (`current`) is updated: vruntime+= T/w, where T is the amount of time elapsed since the last time the load was examined (typically since the last clock tick) and w is the relative weight of the process (Wn/LW). The higher the relative weight, the less vruntime will be "charged" to the current process. The vruntime is cummulative for the life of the process.

Therefore, the lower the vruntime of a process, the greater is its relative merit for being scheduled now, i.e. its dynamic priority. The process with the lowest vruntime of all the runnable proceses is the one which should be on the CPU.

The CFS scheduler maintains a data structure (it is implemented as a red-black binary tree with caching of the lowest element) to keep all of the runnable tasks in order by vruntime. Retrieval of the "next best" task to run is therefore constant time.

When examined in fine-grain detail, the actual timeslices of tasks will not match their "ideal" computed value. But on the average over a longer period of time, the selection of the next task to run based on lowest vruntime will result in an equitable distribution of CPU time which is approximately equal to that which would have been obtained if it were possible to implement the ideal weighted timeslice. This is depicted below, with tasks A and B as previously described having weights of 1.00 and 0.80. The ideal timeslices of 5.55 and 4.45 msec must be rounded up to 6 and 5, because it is impossible to give a task a timeslice which is a fraction of the tick time. Every time A runs, it gets charged 6*1.0=6.0 of vruntime, but when B runs for 5msec, it gets charged 5/0.80=6.25. The two tasks will alternate running for their respective timeslices, but each time A is being cheated of 0.55 msec and B is getting 0.55 msec bonus time. Eventually, the vruntime catches up with B and it skips one turn.

Current	Slice	VRA	VRB
		0	0
A	6ms	6	0
B	5	6	6.25
A	6	12	6.25
B	5	12	13.5
A	6	18	13.5
B	5	18	19.75
A	6	24	19.75
B	5	24	26
	...		
A	6	570	575.75
B	5	576	582
A	6	582	582
A	6	588	582
B	5	588	588.25

Scheduler interactions with fork

In the CFS scheduler, upon a fork, the child process inherits the vruntime of the parent at the time of the fork. This somewhat mitigates the cheating that could otherwise occur if child processes were given a 0 initial vruntime.

The Scheduler Tick

The periodic interval timer fires at a specific rate (typically once every millisecond). This causes `scheduler_tick()` to be called in an interrupt context.

If the current task has been running for longer than its weight-adjusted target timeslice, then the TIF_NEED_RESCHED flag is set, which makes it a candidate for a context switch when the tick interrupt returns to user mode. Of course, if there are no "better" tasks to run, it will not be switched out.

It is also possible that the distribution of weights has changed since the current task started running, because other tasks have either gone to sleep or woken up. So even if the current task has not yet completed its time slice, its TIF_NEED_RESCHED flag may get set. This will happen if there is another task which has now become "better."

```
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;

        sched_clock_tick();  // Update the system clock

        raw_spin_lock(&rq->lock);
        update_rq_clock(rq);
        update_cpu_load(rq);
          /* Invoke the scheduler class specific tick routine
                   For normal tasks, this lands at entity_tick() below
          */
        curr->sched_class->task_tick(rq, curr, 0);
        raw_spin_unlock(&rq->lock);

}


/* The run queue cfs_rq represents all of the runnable tasks for
         this processor.  The sched_entity is a part of the task_struct
         that tracks scheduler statistics
*/

entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
          /* Update the scheduling stats for curr, including vruntime */
        update_curr(cfs_rq);
          /* If there are other runnable tasks on this runqueue,
                   check if we should be preempted */
        if (cfs_rq->nr_running > 1)
                check_preempt_tick(cfs_rq, curr);
}

check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;

          /* Calculate the "ideal" timeslice as discussed in the text */
        ideal_runtime = sched_slice(cfs_rq, curr);
          /* delta_exec is how long curr has been running */
        delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
        if (delta_exec > ideal_runtime) {
                   /* We have run for more than our ideal timeslice, preempt */
                   /* resched_task basically sets the TIF_NEED_RESCHED flag */
                resched_task(rq_of(cfs_rq)->curr);
                return;
        }
```

```
          /* If the WAKEUP_PREEMPT scheduler feature is not configured, then
                    woken-up tasks never preempt a running task, so if the above
                    check for our timeslice is fine, we just keep running until
                    our timeslice is up
          */
        if (!sched_feat(WAKEUP_PREEMPT))
                return;

          /* If the current task has just started running, it wouldn’t be
                    nice to preempt it now.  Maybe later.
          */
        if (delta_exec < sysctl_sched_min_granularity)
                return;

          /* Otherwise, if there are other runnable tasks in our run queue,
                    AND the "next best" task has a lower vruntime than us,
                    AND it is better by the value of our time slice or more,
                    THEN we get preempted
          */
        if (cfs_rq->nr_running > 1) {
                    //__pick_next_entity picks runnable task with lowest vruntime
                struct sched_entity *se = __pick_next_entity(cfs_rq);
                long delta = curr->vruntime - se->vruntime;

                if (delta > ideal_runtime)
                        resched_task(rq_of(cfs_rq)->curr);
        }
}
```

Interactive performance

The vruntime approach naturally favors a process that has woken up after a long sleep, because its vruntime has not been incremented. When a process W is awoken, its vruntime is compared to the current process C. If VR(W)>VR(C), then the current process is still "better" than the awoken, and it will not be pre-empted. This can happen if the current task has a much better nice value than the awoken task.

The actual algorithm in the kernel is somewhat more sophisticated, because of the need to consider the last CPU that a task ran on before sleeping or getting pre-empted. Depending on how the load is balanced, and the length of time elapsed, it may be better to let a task wait a little longer in order to get back on its last CPU.

Overview of the old Linux scheduler

In the Linux kernel, the `schedule` function is called from other kernel functions to relinquish the CPU. `schedule` selects another task (or possibly the same task) to run on the CPU in question. `schedule` is called either directly, e.g. when a system call blocks, or indirectly when the `TIF_NEED_RESCHED` flag is noticed upon return back to user mode from a system call, page fault or interrupt.

The Linux kernel uses a system of numeric priorities. The lower a task's **dynamic priority**, the more favored it is for selection by `schedule`. The dynamic priority is calculated from a number of factors including the `static priority` (which can be set by the system administrator).

When a task is switched in, it is given a **quantum**, i.e. an allowance of a certain amount of time, before it must relinquish the CPU. A task which calls a blocking system call will yield before the expiration of the quantum. On the other hand, a **compute-bound** task which is performing computations will continue to execute until the quantum expires. In this case the task will be pre-empted, i.e. it will be switched out without ever volunteering for this.

The mechanism for time-slicing and pre-emptive scheduling is the periodic timer interrupt. Each such interrupt is called a *tick*. On Linux/X86 the tick rate is 1000 ticks per second. When a tick happens, the time remaining in the quantum of the current process is decremented. If it reaches 0, and another process is ready to run, the current process is switched out.

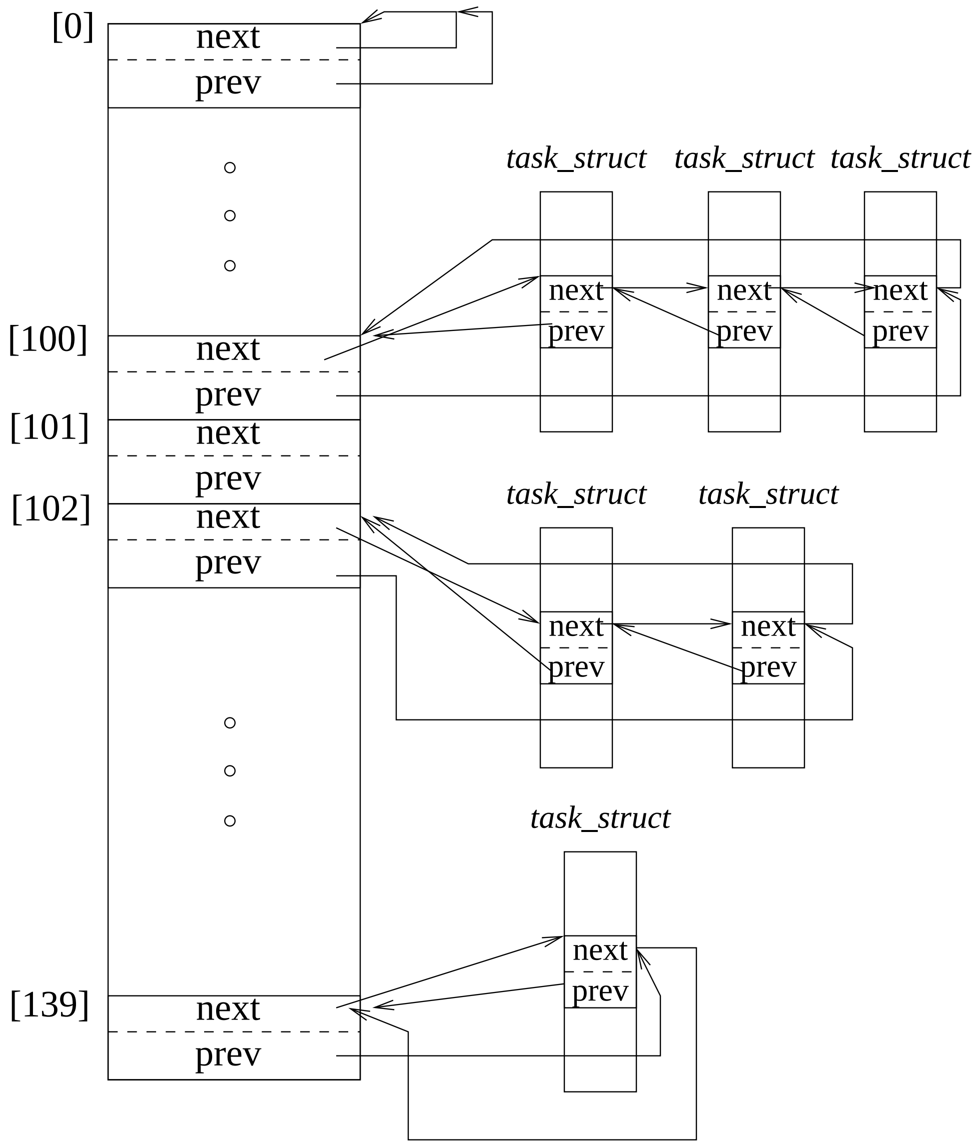
Run Queues

The Linux kernel keeps runnable tasks in data structures called *runqueues*. There is one run queue for each processor in the system. Each runqueue contains an array of circular linked lists. There are 139 list anchors, i.e. one for each of the possible dynamic priority levels. The runqueue actually contains two such priority arrays, one called the *active* and one called the *expired*. As we shall see, the scheduler uses these dual arrays to improve interactive response.

If a process is runnable, then it is on exactly one runqueue. When a process is sleeping, it is not on any runqueues. Normally, when processes wake up, they are placed on the runqueue of the CPU which last executed them. This is to improve cache performance. However, processes are periodically migrated among runqueues of a multiprocessor system to keep the load more or less balanced.

When the scheduler looks for the "best" task to run next, it visits the runqueue in dynamic priority order, starting with the "best" priority level (which in Linux is the lowest numerically). At the first non-empty list slot, the scheduler picks the task at the head of the lists of tasks at that priority. Since tasks are placed at the rear of the list (for a given priority) when they are switched out, if the dynamic priorities were to never change and always be equal to the static priorities, the scheduler would at least implement "**Round-Robin**" scheduling.

Because searching for the first non-empty list requires a large number of memory accesses, and because it is important to make the scheduling decision quickly with minimal overhead, the Linux kernel actually adds a bitmask which represents the emptiness of the priority buckets. Thus the scheduler needs only to fetch the bitmask and (on the X86 architecture) execute an instruction which finds the bit number of the first bit in a word which is set.



Priority formula

The dynamic priority ranges from 1 to 139, with 1 being the "best" or "most preferred" value. However, priority values 1 through 99 are reserved for "real-time" processes. Conventional processes which are ordinarily experienced by end-users have priority values ranging from 100 to 139. This corresponds to the traditional UNIX use of 40 priority levels.

The *static priority* can be adjusted by the process. The default value is 120, i.e. right in the middle. Any process can give itself worse static priority by using the `nice` system call, but only processes running as root can improve the static priority of themselves or others (improve == make it numerically lower). The difference between the static priority and 120 is the "nice value". Starting a process with a positive nice value is a "nice" thing to do...it volunteers to make the process less of a CPU hog.

The *dynamic priority* is periodically recalculated as follows:

```
dyn_prio = Max( 100, Min(static_prio-bonus+5,139))
```

`bonus` ranges from 0 to 10 and is a measure of the average sleep time of the process. A bigger bonus means a smaller (better) dynamic priority. The longer the average sleep time, the bigger the bonus. In the current version of the kernel, proceses with sleep averages of less than 100ms get 0 bonus, anything with 1000ms or more gets a bonus of 10, and values in between are linearly interpolated.

The effect of the bonus is to reward interactive or I/O bound processes which are not hogging the CPU. Furthermore, if the expression

```
bonus-5 >= static_prio/4-28
```

is true, then the process has met the threshold for being considered an INTERACTIVE process. This will create an additional benefit as discussed below in conjunction with the active vs expire run queue. Note that processes with numerically lower static priority can become INTERACTIVE more easily. A process which is not INTERACTIVE is considered a BATCH process (i.e. CPU-bound).

Quantum

When a process is selected to run and switched in, it is given an allocation of a certain number of ticks, which is known as the *quantum*. Linux uses a variable-quantum scheduling algorithm:

```
if (static_prio < 120)
          quantum=(140-static_prio)*20;
else
          quantum=(140-static_prio)*5;
```

Therefore, more favored (lower static_prio) processes get a longer quantum.

Active and Expired

The scheduler maintains two priority arrays, active and expired. Consider a state where all runnable processes are on the active list. One by one, as they exhaust their time quantum but remain runnable, they are moved to the expired priorty array, using the current dynamic priority at the time of the quantum exhaustion. The scheduler, when looking for the next process to run, always looks at the active list first. The effect of this decision is to prevent a high priority process from monopolizing the CPU. After it has had its turn, lower priority processes must be allowed run once.

This behavior is modified by the INTERACTIVE vs BATCH classification scheme above. While BATCH processes always get moved to the expired array, INTERACTIVE processes get put back on the active array, UNLESS one of the following is true:
• An expired process has been waiting to be scheduled for a really long time.
• OR any of the expired processes has a better static priority than the INTERACTIVE process under consideration.

One of these two conditions will eventually become true, and all of the processes in the run queue will wind up in the expired part. When this happens, and the active part is empty, then the expired part becomes the active part, and the new expired part becomes empty.

Idle

What happens when there are NO runnable processes in the system, e.g. a quiescent system waiting for input? Then the scheduler will not be able to select a runnable process. The Linux kernel uses a special process called, for historical reasons, `swapper`, as the idle task. It is selected when there is absolutely nothing else to do. On multi-processor systems, there is one idle task for each processor.

When power management features are enabled, idle processors may take action to reduce power consumption, such as slowing the CPU clock speed, or powering down disk drives.

scheduler_tick

The `scheduler_tick` function is called in an interrupt context by the periodic timer interrupt handler. It performs the following operations (in pseudo-code)

```
scheduler_tick()
{
        if (current == idle process)
        {
                if (local runqueue contains a runnable process)
                        set TIF_NEED_RESCHED
        }
        else
        {
                if (--current->time_slice<=0)
```

```
                    {
                              remove current from this_rq_>active
                              set TIF_NEED_RESCHED
                              recalculate current->prio //dynamic prio
                              set current->time_slice based on current->static_prio
                              if (TASK_INTERACTIVE(current))
                              {
                                        if ((process on this_rq->expired has
                                                  been waiting more than
                                                  1000*(number_runnable+1)) ||
                                            (an expired process has better static_prio
                                                  than current))
                                                  put current on this_rq->expired
                                        else
                                                  put current on this_rq->active
                              }
                              else
                                        put current on this_rq->expired
                    }
          }
          rebalance run queues based on multiprocessor load
}
```

Inserting a task into the run queue at wakeup

When a task is woken up, this is done by the kernel function `try_to_wake_up`:
• Identify target runqueue. On multi-processor systems, it will usually be the queue of the last CPU to run the process. However, if a different CPU is idle, or has drastically lower workload than the last CPU, it will be migrated.
• Recalculate the average sleep time and dynamic priority.
• Place the woken-up task on the active part of the target run queue.
• If the woken-up task now has a better priority than the task currently executing on the target CPU, set the TIF_NEED_RESCHED flag of the latter task. On multi-processor systems, it may be necessary to send an inter-processor interrupt if the new target CPU is different from the CPU executing try_to_wake_up.

Scheduler interactions with fork

When a new task is created with fork or clone, the Linux kernel gives it an initial quantum which is half of the parent's remaining time. The parent's remaining time is then divided by two. In this way, allocation of CPU resources remains fair and any one user can not monopolize the CPU by use of a "fork bomb".