Linux kernel scheduler

Mon, Jan 5, 2015 os (/jkoo/tags/os)

(https://www.facebook.com/sharer.php?u=https%3a%2f%2fhelix979.github.io%2fjkoo%2fpost%2fos-scheduler%2f)

(https://twitter.com/intent/tweet?text=Linux%20kernel%20scheduler&url=https%3a%2f%2fhelix979.github.io%2fjkoo%2fpost%2fos-scheduler%2f)

(https://www.linkedin.com/shareArticle?mini=true&url=https%3a%2f%2fhelix979.github.io%2fjkoo%2fpost%2fos-scheduler%2f&title=Linux%20kernel%20scheduler)

(http://service.weibo.com/share/share.php?url=https%3a%2f%2fhelix979.github.io%2fjkoo%2fpost%2fos-scheduler%2f&title=Linux%20kernel%20scheduler)

(mailto:?subject=Linux%20kernel%20scheduler&body=https%3a%2f%2fhelix979.github.io%2fjkoo%2fpost%2fos-scheduler%2f)

What is the kernel?

The kernel is fundamental part of an operating system (OS) that manages the computer's hardwares, and allows softwares to run and use hardware resources in shared manners. Typically, the hardware resources to take into account are: (1) processors, (2) memory, and (3) input/output (I/O) devices such as keyboard, disk drives, network interface cards, and so on.

Rough distinction between an OS and a kernel is that an OS is the kernel plus some useful utilities and applications such as administration tools and GUIs.

Monolithic kernels and modules

Linux has a monolithic kernel that contains all of the code necessary to perform every kernel related task in a single binary file. However, Linux can extend its functionality by adding modules. Here, the modules are pieces of code that can be loaded and unloaded into the kernel upon demand while a system is up and running.

Process

A process is an instance of a computer program that is being executed. Each process has its own address space, which is protected from being accessed by other processes except through a legitimate means, that is, an inter-process communication mechanism.

The address space is typically partitioned into several regions: text, data, bss, heap and stack. The text segment contains the compiled code of a program, *i.e.*, a set of instructions. The data segment stores initialized global and static variables, and constant variables like strings. The uninitialized global and static variables are located in the bss segment. The heap is the region set aside for dynamic memory allocation. The stack is where local variables are allocated within functions.

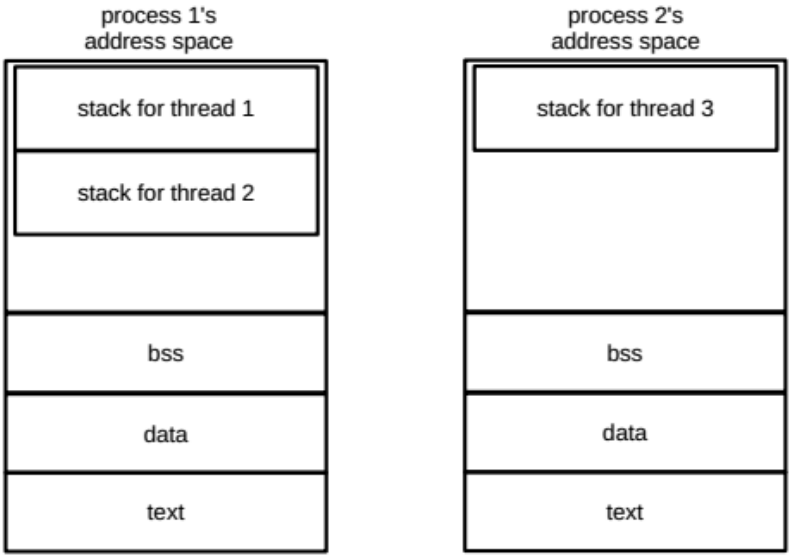

FIGURE 1: Address spaces.
No sharing between processes; threads within a process share text, data, bss.

A thread is also the programmed code in execution. The thread is the smallest unit that can be managed independently by a kernel scheduler. However, unlike a process, the thread runs inside the address space of a process that it belongs to (refer to Figure 1.) Multiple threads can exist within the same process and share memory. Thanks to the shared memory, threads can easily communicate with one another. Although each thread has a separate stack for local variables and function calls, the stacks are allocated from the shared data area in the process' address space.

In Linux kernel terms, a thread is often called a task. In the meantime, since by default

Linux kernel creates a process with a single thread by which actual work of the process is done, a thread may look like a process. For that reason, we sometimes refer to a process as a task as well.

Creating a new process: fork() and exec()

Linux kernel generates a new process using `fork()` system call directly followed by `exec()` system call.

When a process invokes `fork()`, a separate address space is created for a new process (called a child process), and all the memory segments of the original process (called a parent process) are copied into there. As a result, both the parent and the child have the exact same content in their own address space. The `fork()` returns the process ID (PID) of a new child process to the parent process, and returns zero to the child process.

When the child process calls `exec()`, all data in the original program is replaced with a running copy of the new program. In other words, `exec()` replaces the current process with a new process of an executable binary file specified in its arguments.

Waiting until a child process terminates

The `wait()` system call allows the parent process to halt execution until its child process finishes. A call to `wait()` returns the PID of the child process on success.

Zombie processes

When a child process terminates, it still exists as an entry in the process table. This entry is required until the parent process reads its child's exit status by calling `wait()` system call, which then removes the entry from the process table.

Thus, the ended child process becomes a meaningless process and just remains until the parent process terminates or it calls `wait()`. The process in this defunct state is called the zombie process.

Copy-on-write

A naive approach to implement `fork()` is that when `fork()` is called, the kernel literally makes the copies of all the data belonging to the parent process, and puts them into the address space for the child process. This is inefficient in that although it takes too much time to duplicate the data, the child may not use any of them in certain cases. For example, if the child issues `exec()` right after `fork()`, all the effort to copy becomes wasteful. Even when the data is indeed used in the child, read-only data can just be shared by having pointers without burdensome copying jobs.

To avoid such inefficiency, the Linux kernel uses what is called the copy-on-write to implement `fork()`. The copy-on-write is a technique by which copying involved in

`fork()` occurs only when either the parent or the child writes. Instead of duplicating all the data upon `fork()`, the parent and the child share the data, marking them to be read-only. When some page (the smallest unit of data for memory management) is modified by any of the two processes, a page fault occurs, which makes each process get a unique copy of the page marked read-write.

PID, TID, PPID, and TGID

The smallest scheduling entity in the Linux kernel is a thread, not a process. Each thread is assigned a unique number for this purpose. In the kernel terms, confusingly enough, this number is called a process ID (PID). The process that threads belong to is accounted for as the thread group ID (TGID).

When a new process starts by invoking `fork()`, it is assigned a new TGID. This newly forked process is created with a single thread, whose PID is the same as the TGID. The parent's TGID is called a parent PID (PPID). If the thread creates another thread, the new thread gets a different PID, but the same TGID is taken over.

In the mean time, some user space applications (*e.g.*, `ps`) have a different (probably better) naming convention: the PID and the TGID in kernel-space view are, respectively, called a thread ID (TID) and a PID in these userland applications. Figure 2 summarizes the relationship among these IDs.

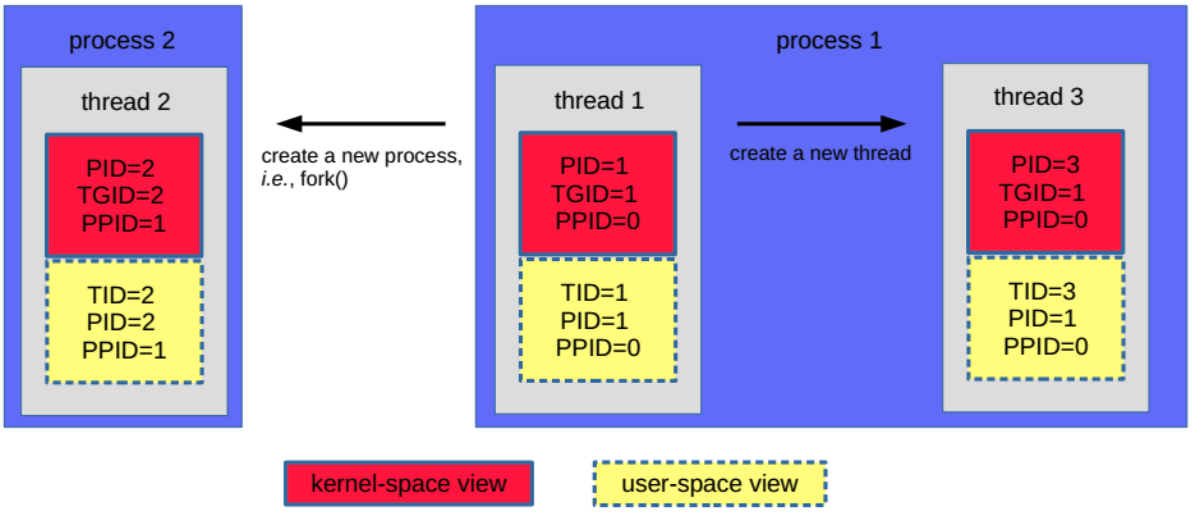

FIGURE 2: Relationship among PID, TID, PPID, and TGID.

Scheduler

What the scheduler does

In modern computer systems, there may be many threads waiting to be served at the same time. Thus, one of the most important jobs of the kernel is to decide which thread to run for how long. The part of the kernel in charge of this business is called the scheduler.

On a single processor system, the scheduler alternates different threads in a time-division manner, which may lead to the illusion of multiple threads running concurrently. On a multi-processor system, the scheduler assigns a thread at each processor so that the threads can be truly concurrent.

Priority

Most of scheduling algorithms are priority-based. A thread is assigned a priority according to its importance and need for processor time. The general idea, which isn't exactly implemented on Linux, is that threads with a higher priority run before those with a lower priority, whereas threads with the same priority are scheduled in a round-robin fashion.

Preemptive scheduling

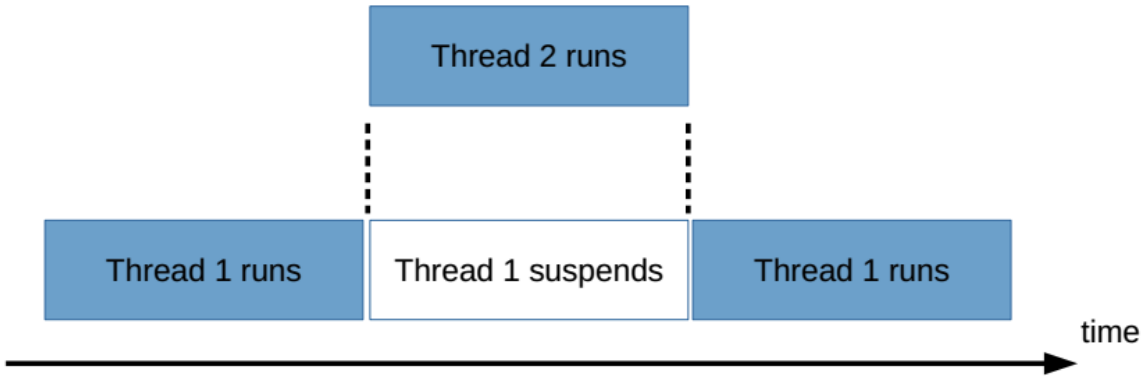

FIGURE 3: Preemption.

Linux kernel features a preemptive scheduling, which means that a thread can stop to execute another thread before it completes as shown in Figure 3. A thread can be preempted by a pending thread that is more important (*e.g.*, of a higher priority). The preempted thread resumes its execution after the preempting thread finishes or blocks.

Context switching

All information that describes the states of the currently running thread is referred to as the "context". The context of a thread are mainly the contents of hardware registers

including the program counter, the address space, and memory map (which will be explained later). Simply put, the context tells up to what point the instructions of the thread is executed, what the outcomes is, and where the content of memory pertinent to the thread exist. These are all you need to know in order to resume the thread at a later time.

When a processor changes a thread to execute from one to another (as a result of scheduling for instance), the context of the old thread is saved somewhere, and the context of the new thread gets loaded. This procedure is called the context switching. The context switching happens fast and frequently enough that the users feel like the threads are running at the same time.

Thread states

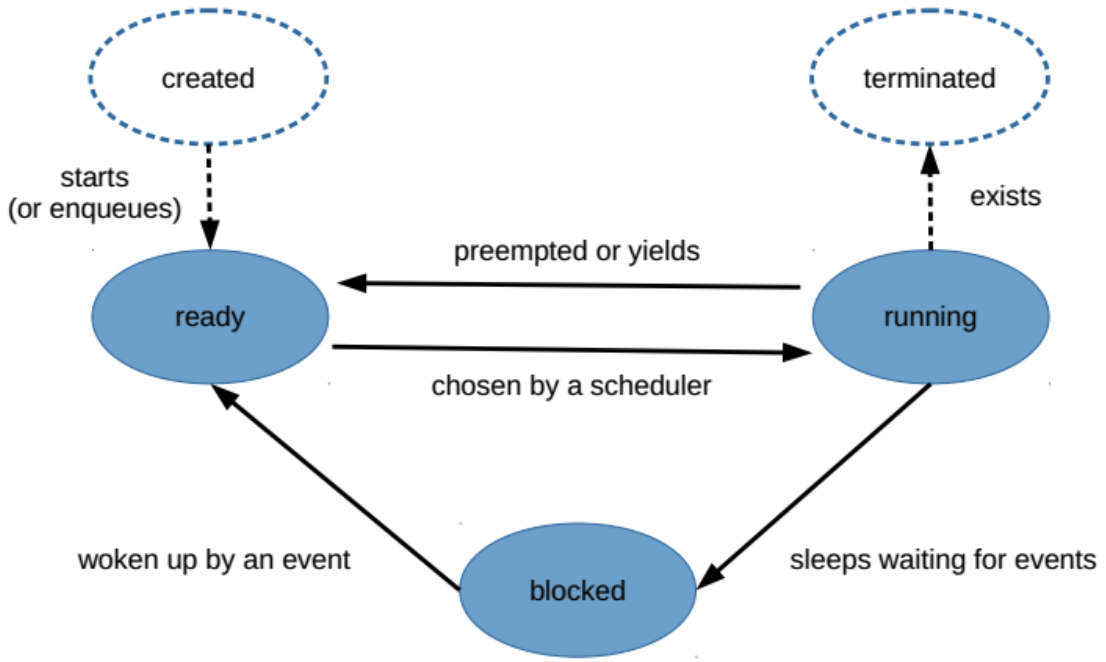

FIGURE 4: Thread states.

The scheduler needs to know which threads are runnable at a given time so that it can choose a right one to run next. For that reason, each thread maintains its current state. In general terms, a thread may stay in one out of the following states (see also Figure 4).

- Ready: The thread is ready to run, but not allowed to, because all processors are busy executing other threads. The ready thread is awaiting execution until the scheduler chooses itself to run next.
- Running: Instructions of the thread are being executed on a processor.

- Blocked: The thread is blocked waiting for some external event such as I/O or a signal. The blocked thread is not a candidate to schedule, *i.e.*, it cannot run.

I/O-bound vs. CPU-bound

Threads (or processes) can be classified into two major types: I/O-bound and CPU-bound.

The I/O-bound threads are mostly waiting for arrivals of inputs (*e.g.*, keyboard strokes) or the completion of outputs (*e.g.*, writing into disks). In general, these threads do not stay running for very long, and block themselves voluntarily to wait for I/O events. What matters with the I/O-bound threads is that they need to be processed in quick, since otherwise users may feel that the system has no good responsiveness. Therefore, a common rule is that a scheduler puts more urgency into serving the I/O-bound threads.

The CPU-bound threads are ones that spend much of their time in doing calculations (*e.g.*, compiling a program). Since there are not many I/O events involved, they tend to run as long as the scheduler allows. Typically, users do not expect the system to be responsive while the CPU-bound threads are running. Thus, the CPU-bound threads are picked to run by a scheduler less frequently. However, once chosen, they holds a CPU for a longer time.

Real-time vs. non-real-time

Another criterion that categorizes threads is if they are real-time threads or not.

The real-time threads are ones that should be processed with a strict time constraint, often referred to as a deadline. The operational correctness of a real-time thread depends not only on computation results, but also on whether the results are produced before the deadline. Therefore, the scheduler, in general, takes care of the real-time threads with a high priority.

Real-time threads can be further classified into hard real-time or soft real-time ones by the consequence of missing a deadline. Hard real-time threads require all deadlines to be met with no exception; otherwise, a system may fall into catastrophic failure. For the soft real-time threads, missing a deadline results in degraded quality for the intended service, but a system can still go on.

Non-real-time threads are not associated with any deadlines. They could be human-interactive threads or batch threads. Here, the batch threads are ones that process a large amount of data without manual intervention. For the batch threads, a fast response time is not critical, and so they can be scheduled to run as resources allow.

Core Scheduler

Scheduling classes

One may say that the Linux kernel scheduler consists of mainly two different scheduling algorithms, which are what are called the real-time scheduler and the completely fair scheduler. Scheduling classes allow for implementing these algorithms in a modular way. In detail, a scheduling class is a set of function pointers, defined through `struct sched_class`.

```
struct sched_class {
    const struct sched_class *next;
    ...
    struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);
    ...
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    ...
};
```

Each scheduling algorithm gets an instance of `struct sched_class` and connects the function pointers with their corresponding implementations.

The `rt_sched_class` implements so-called real-time (RT) scheduler.

```
const struct sched_class rt_sched_class = {
    .next           = &fair_sched_class,
    ...
    .pick_next_task     = pick_next_task_rt,
    .put_prev_task      = put_prev_task_rt,
    ...
    .task_tick      = task_tick_rt,
    ...
};
```

As its name implies, the RT scheduler targets to deals with the real-time threads. The RT scheduler assigns a priority to every thread to schedule, and processes the threads in order of their priorities. The RT scheduler is proved good enough by many people's experience, but there is no guarantee that all deadlines are met. Namely, the RT scheduler in the Linux kernel only addresses the needs of threads with soft real-time requirements.

The completely fair scheduler (CFS) is implemented by the `fair_sched_class`.

```
const struct sched_class fair_sched_class = {
    .next           = &idle_sched_class,
    ...
    .pick_next_task     = pick_next_task_fair,
    .put_prev_task      = put_prev_task_fair,
    ...
    .task_tick      = task_tick_fair,
    ...
};
```

The CFS also assigns a priority to a thread. However, unlike in the RT scheduler, this priority does not directly mean the order of being processed. Rather, it decides how long a thread can occupies a processor compared to others. In other words, the priority in CFS determines the proportion of processor time that a thread can use. Threads with a high priority can hold a processor longer than threads with a low priority. Meanwhile, the CFS may allow a long-waited low-priority thread to run even though there are high-priority threads ready. This is because each thread is guaranteed to use its own fraction of processor time for a certain time interval according to its priority, which is why the term "fair" comes in the name of this scheduling algorithm.

The core logics of the kernel scheduler iterate over scheduler classes in order of their priority: `rt_sched_class` processed prior to `fair_sched_class`. That way, codes in the `rt_sched_class` does not need to interact with codes in the `fair_sched_class`.

Scheduling policies

When created, each thread gets assigned a scheduling policy that is in turn treated by a specific scheduling algorithm. Different scheduling policies may result in different outcomes even with the same scheduling algorithm.

The RT scheduler supports the following two scheduling policies:

- `SCHED_RR` : Threads of this type run one by one for a pre-defined time interval in their turn (round robin).
- `SCHED_FIFO` : Threads of this type run until done once selected (first-in/first-out).

The scheduling polices dealt with by the CFS include:

- `SCHED_BATCH` : This policy handles the threads that have a batch-characteristic, *i.e.*, CPU-bounded and non-interactive. Threads of this type never preempt non-idle threads.
- `SCHED_NORMAL` : Normal threads fall into this type.

Run queues

The core scheduler manages ready threads by enqueueing them into a run queue, which is implemented by `struct rq`.

```
struct rq {
    ...
    struct cfs_rq cfs;
    struct rt_rq rt;
    ...
    struct task_struct *curr, *idle, *stop;
    ...
    u64 clock;
    ...
};
```

Each CPU has its own run queue, *i.e.*, there are as many run queues as the number of CPUs in a system. A ready thread can belong to a single run queue at a time, since it is impossible that multiple CPUs process the same thread simultaneously. Here comes the need of load balancing among CPUs in multi-core systems. Without a special effort to balance load, threads may wait in a specific CPU's run queue, while other CPUs have nothing in their run queue, which, of course, means performance degradation.

A run queue includes `struct cfs_rq cfs` and `struct rt_rq rt`, which are sub-run queues for the CFS and the RT scheduler, respectively. Enqueueing a thread into a run queue eventually means enqueueing it into either of these sub-run queues depending on the scheduler class of the thread. The thread that is currently running on a CPU is pointed by `struct task_struct *curr` defined in the run queue of the CPU. The per-run queue variable `clock` is used to store the latest time at which the corresponding CPU reads a clock source.

The main body: `__schedule()`

The function `__schedule()` is the main body of the core scheduler. What it does includes putting the previously running thread into a run queue, picking a new thread to run next, and lastly switching context between the two threads.

```
static void __sched __schedule(void)
{
    ...
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;
    ...
    put_prev_task(rq, prev);
    ...
    next = pick_next_task(rq);
    ...
    if (likely(prev != next)) {
        ...
        rq->curr = next;
        ...
        context_switch(rq, prev, next);
        ...
    }
    ...
}
```

Most work in `__schedule()` is delegated to the scheduling classes. For example, when `put_prev_task()` is invoked in `__schedule()`, actual work is done by the function registered to the function pointer `put_prev_task` of the scheduling class that the previously running task belongs to.

```
static void put_prev_task(struct rq *rq, struct task_struct *prev)
{
    ...
    prev->sched_class->put_prev_task(rq, prev);
}
```

As shown in `rt_sched_class` and `fair_sched_class`, this is `put_prev_task_rt()` for the RT scheduler and `put_prev_task_fair()` for the CFS. A similar thing applies when `pick_next_task()` is invoked.

```
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)

static inline struct task_struct *pick_next_task(struct rq *rq)
{
    const struct sched_class *class;
    struct task_struct *p;
    ...
    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }
    ...
}
```

The function `pick_next_task()` seeks the next-running thread using the function pointer `pick_next_task` of a scheduling class by which `pick_next_task_fair()` and `pick_next_task_rt()` are called for the CFS and the RT scheduler, respectively. Note that by the `for_each_class(class)` loop, scheduling classes are processed one by one in order of their priority, by which `fair_sched_class` can be taken care of only if there is nothing to do with the `rt_sched_class`.

The function `__schedule()` is invoked at many places of the kernel, where there is a need to reschedule threads. One of such cases is after interrupt handling, since by an interrupt, some thread (*e.g.*, a high-priority RT thread) may need to run immediately. Another case is when someone calls it explicitly. For example, a system call `sched_yield()` that causes the calling thread to relinquish the CPU is implemented using `__schedule()`.

Periodic accounting: `scheduler_tick()`

The function `scheduler_tick()` is periodically called by the kernel with the frequency `HZ`, which is the tick rate of the system timer defined on system boot.

```
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    ...
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    ...
}
```

The first thing among what `scheduler_tick()` does is updating clocks invoking `update_rq_clock()`. The `update_rq_clock()` reads a clock source and updates the `clock` of the run queue, which the scheduler's time accounting is based on. The second thing is checking if the current thread is running for too long, and if it is, setting a flag that indicates that `__schedule()` must be called to replace the running task with another. This done by calling `task_tick` in a scheduler class. Again, actual work is delegated to a scheduler-class-specific function pointed by this function pointer.

Completely Fair Scheduler (CFS)

Nice values: priorities in the CFS

nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916

nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
nice	0	1	2	3	4	5	6	7	8	9
weight	1024	820	655	526	423	335	272	215	172	137
nice	10	11	12	13	14	15	16	17	18	19
weight	110	87	70	56	45	36	29	23	18	15

FIGURE 5: Nice-to-weight conversion.

The priority managed by the CFS is called the nice value in particular, which ranges between -20 and 19. Lower values mean higher priorities (*i.e.*, -20 for the highest priority and 19 for the lowest priority). The default nice value is 0 unless otherwise inherited from a parent process. Each nice value has a corresponding weight value, which predefined as Figure 5. Note that there is an inverse relationship between weight values and nice values. The weight value determines how large proportion of CPU time a thread gets compared to other threads. Refer to "time slice" for more detail.

Time slice

The CFS sets what is called a time slice that is an interval for which a thread is allowed to run without being preempted. The time slice for a thread is proportional to the weight of the thread divided by the total weight of all threads in a run queue. Therefore, the thread that has a relatively high priority is likely to run longer than the other ready threads.

The function `__scheduler_tick` that is periodically called by a timer interrupt invokes `task_tick` of the scheduling class of the current thread. In the CFS, this function pointer eventually executes the function `check_preempt_tick()`.

```
static void check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    ...
    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime) {
        resched_task(rq_of(cfs_rq)->curr);
        ...
        return;
    }
    ...
}
```

The `check_preempt_tick()` is responsible for checking if the current thread is running any longer than its time slice. If that is the case, `check_preempt_tick()` calls `resched_task()` that marks that `__schedule()` should be executed now to change the running thread. The function `sched_slice()` is the one that calculates the time slice for the currently running thread.

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);
    ...
    cfs_rq = cfs_rq_of(se);
    load = &cfs_rq->load;
    ...
    slice = calc_delta_mine(slice, se->load.weight, load);
    ...
    return slice;
}
```

The Linux kernel sets a scheduling period during which all ready threads are guaranteed to run at least once. The function `__sched_period` updates the scheduling period considering the number of ready threads.

```
static u64 __sched_period(unsigned long nr_running)
{
    u64 period = sysctl_sched_latency;
    unsigned long nr_latency = sched_nr_latency;
    if (unlikely(nr_running > nr_latency)) {
        period = sysctl_sched_min_granularity;
        period *= nr_running;
    }
    return period;
}
```

By default, the kernel targets to serve `sched_nr_latency` threads for `sysctl_sched_latency` ms, assuming that a thread is supposed to run at a time for at least `sysctl_min_granularity` ms, which is defined as follows:

$$\text{sysctl\_min\_granularity} = \frac{\text{sysctl\_sched\_latency}}{\text{sched\_nr\_latency}}$$

That is, the default scheduling period is `sysctl_sched_latency` ms. However, if there are more than `sched_nr_latency` threads in a run queue, the scheduling period is set to `sysctl_min_granularity` ms multiplied by the number of ready threads.

The updated scheduling period is scaled by the function `calc_delta_mine()` that finalizes the time slice for the currently running thread in the following way:

$$\begin{aligned}&\text{time slice of the current thread}\\ &= (\text{scheduling period}) * \left(\frac{\text{weight of the current thread}}{\text{sum of the weights of all threads}}\right).\end{aligned}$$

Virtual runtime

Time accounting in the CFS is done by using the so-called virtual runtime. For a given

thread, its virtual runtime is defined as follows:

$$\text{virtual runtime} = (\text{actual runtime}) * 1024/\text{weight}.$$

Since a weight is proportional to a priority, the virtual runtime of a high priority thread goes slower than that of a low priority thread, when the actual runtime is the same. Note that in the above, 1024 is the weight value for the nice 0. Thus, the virtual runtime for the thread of nice 0 is equal to its actual runtime.

All updates to the virtual runtime are performed in `update_curr()`.

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_of(cfs_rq)->clock_task;
    unsigned long delta_exec;
    ...
    delta_exec = (unsigned long)(now - curr->exec_start);
    ...
    __update_curr(cfs_rq, curr, delta_exec);
    curr->exec_start = now;
    ...
}
```

`clock_task` is used to get a timestamp `now` at the moment when `update_curr()` is invoked. The `clock_task` returns `rq->clock` minus time stolen by handling IRQs. `curr->exec_start` holds the timestamp that was made when the current thread updated its virtual runtime most recently. Thus, the difference between `now` and `curr->exec_start` is the actual runtime elapsed since the last update to the virtual runtime of the current thread. This actual time increment is fed into `__update_curr()` where conversion to virtual time increment is done by `calc_delta_fair()` below.

```
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
          unsigned long delta_exec)
{
    ...
    delta_exec_weighted = calc_delta_fair(delta_exec, curr);
    curr->vruntime += delta_exec_weighted;
    ...
}
```

Putting a running thread back into a runqueue

In order to change the running thread, the previously-running thread should be first back into a runqueue. For this matter, the CFS uses `put_prev_task_fair()` that in turn calls `put_prev_entity()`.

```
static void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
{
    if (prev->on_rq)
        update_curr(cfs_rq);
    ...
    if (prev->on_rq) {
        ...
        __enqueue_entity(cfs_rq, prev);
        ...
    }
    cfs_rq->curr = NULL;
}
```

Using `prev->on_rq`, the `put_prev_entity()` checks if the thread is already on a run queue, in which case nothing should be done. Otherwise, the running thread needs to update its virtual runtime and enqueue into the `cfs_rq`. The `cfs_rq` is implemented with a red-black (RB) tree, where threads are sorted according to their virtual runtime.

Choosing the thread to run next

Choosing the next thread to run in the CFS is the business of `pick_next_task_fair()`, whose main body is implemented by a sub-routine `pick_next_entity()`.

```
static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq)
{
    struct sched_entity *se = __pick_first_entity(cfs_rq);
    struct sched_entity *left = se;
    if (cfs_rq->skip == se) {
        struct sched_entity *second = __pick_next_entity(se);
        if (second && wakeup_preempt_entity(second, left) < 1)
            se = second;
    }
    if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
        se = cfs_rq->last;

    if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
        se = cfs_rq->next;
    ...
    return se;
}
```

Here, `wakeup_preempt_entity()` is the means to balance fairness in terms of virtual time among threads. Specifically, what `wakeup_preempt_entity(se1, se2)` does is to compare the virtual times of `se1` and `se2`, and return -1 if `se1` has run shorter than `se2`, 0 if `se1` has long than `se2` but not long enough, and 1 if thread `1` has run long enough. Keeping things fair between threads using this function, the thread to run next is picked in the following order.

1. Pick the thread that has the smallest virtual runtime.

2. Pick the “next” thread that woke last but failed to preempt on wake-up, since it may need to run in a hurry.
3. Pick the “last” thread that ran last for cache locality.
4. Do not run the “skip” process, if something else is available.

Real-Time (RT) Scheduler

The run queue of the RT scheduler

The RT scheduler’s run queue, represented by `struct rt_rq`, is mainly implemented with an array, each element of which is the head of a linked list that manages the threads of a particular priority.

```
struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1);
    struct list_head queue[MAX_RT_PRIO];
};

struct rt_rq {
    struct rt_prio_array active;
    ...
    int curr;
    ...
}
```

All real-time threads whose priority is `x` are inserted into a linked list headed by `active.queue[x]`. When there exists at least one thread in `active.queue[x]`, the `x`-th bit of `active.bitmap` is set.

Execution and scheduling polices

A newly queued thread is always placed at the end of each list of a corresponding priority in the run queue. The first task on the list of the highest priority available is taken out to run.

There are two scheduling polices applied for the RT scheduler, which are `SCHED_FIFO` and `SCHED_RR`. The difference between the two becomes distinct in `task_tick_rt()`.

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
    struct sched_rt_entity *rt_se = &p->rt;

    update_curr_rt(rq);

    watchdog(rq, p);

    /*
     * RR tasks need a special form of timeslice management.
     * FIFO tasks have no timeslices.
     */
    if (p->policy != SCHED_RR)
        return;

    if (--p->rt.time_slice)
        return;

    p->rt.time_slice = sched_rr_timeslice;

    /*
     * Requeue to the end of queue if we (and all of our ancestors) are the
     * only element on the queue
     */
    for_each_sched_rt_entity(rt_se) {
        if (rt_se->run_list.prev != rt_se->run_list.next) {
            requeue_task_rt(rq, p, 0);
            set_tsk_need_resched(p);
            return;
        }
    }
}
```

The threads with `SCHED_FIFO` can run until they stop or yield. There is nothing to be done every tick interrupt. The `SCHED_RR` threads are given a time slice, which is decremented by 1 on the tick interrupt. When this time slice becomes zero, `SCHED_RR` threads are enqueued again.

Python in half an hour → (https://helix979.github.io/jkoo/post/python-tutorial/)