进程调度

张雷

南京大学计算机系

2018-10-30

CPU Scheduler

CPU Scheduler

Lab3: Linux kernel synchronization mechanism

Build a new kernel synchronization mechanism (a new lock) based on device orientation.
Implement the following four system calls.

```
/*
 * Sets current device orientation in the kernel.
 * System call number 326.
 */
int set_orientation(struct dev_orientation *orient);

struct dev_orientation {
    int azimuth; /* rotation around the X-axis (-180<=azimuth<=180)*/
    int pitch;   /* rotation around the Y-axis: -90<=pitch<=90 */
    int roll;    /* rotation around Z-axis: -180<=roll<=180 */
};
```

Lab3: Linux kernel synchronization mechanism

```c
/*
 * Create a new orientation event using the specified orientation range.
 * Return an event_id on success and appropriate error on failure.
 * System call number 327.
 */
int orientevt_create(struct orientation_range *orient);

/*
 * Destroy an orientation event and notify any processes which are
 * currently blocked on the event to leave the event.
 * Return 0 on success and appropriate error on failure.
 * System call number 328.
 */
int orientevt_destroy(int event_id);

/*
 * Block a process until the given event_id is notified. Verify that the
 * event_id is valid.
 * Return 0 on success and appropriate error on failure.
 * System call number 329.
 */
int orientevt_wait(int event_id);
```

Lab3: Test

写两个C程序来测试系统。两个程序分别fork n个child（n>2）。每个child都等待同一个事件，并执行以下操作：

- faceup: 模拟器屏幕朝向你时每隔一秒打印`facing up`
- facedown: 模拟器屏幕背向你时每隔一秒打印`facing down`

同时父进程等待60秒，然后通过关闭打开的方向事件来关闭所有子进程（而不是通过发送信号或其他此类方法）。

```
struct orientation_range r;
int event_id = orientevt_create(&r);

/*child 1,2,3...*/
while(1)
{
    orientevt_wait(event_id);
    printf("facing down!");
    sleep(1);
}

/*parent*/
sleep(60);
orientevt_destroy(event_id); /*reference counting*/
```

Lab3: Linux kernel synchronization mechanism

Sleeping via wait queues: A wait queue is a list of processes waiting for an event to occur, wait queues are represented in the kernel by `wake_queue_head_t`.

- created statically via `DECLARE_WAITQUEUE()`
- created dynamically via `init_waitqueue_head()`

Processes put themselves on a wait queue and mark themselves not runnable. When the event associated with the wait queue occurs, the processes on the queue are awakened.

```
/* 'q' is the wait queue we wish to sleep on */
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        /* handle signal */
    schedule();
}
finish_wait(&q, &wait);
```

Lab3: Linux kernel synchronization mechanism

Sleeping via wait queues

1. Creates a wait queue entry via the macro `DEFINE_WAIT()`.
2. Adds itself to a wait queue via `add_wait_queue()`. This wait queue awakens the process when the condition for which it is waiting occurs. Of course, there needs to be code elsewhere that calls `wake_up()` on the queue when the event actually does occur.
3. Calls `prepare_to_wait()` to change the process state to either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`. This function also adds the task back to the wait queue if necessary, which is needed on subsequent iterations of the loop.
4. If the state is set to `TASK_INTERRUPTIBLE`, a signal wakes the process up. This is called a spurious wake up (a wake-up not caused by the occurrence of the event). So check and handle signals.
5. When the task awakens, it again checks whether the condition is true. If it is, it exits the loop. Otherwise, it again calls `schedule()` and repeats.

CPU Scheduler

CPU scheduler

The CPU scheduler selects from among the processes in ready queue, and allocates the a CPU core to one of them

- Queue may be ordered in various ways

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is nonpreemptive. All other scheduling is preemptive.

调度时机

- 主动调度：直接调用schedule()，如进程退出，或者进入睡眠状态等
- 内核抢占：当进程位于内核空间时，有一个更高优先级的任务出现时，如果当前内核允许抢占，则可以将当前任务（如用户进程执行系统调用）挂起，执行优先级更高的进程
- 用户抢占：内核返回用户空间时（从系统调返回用户空间或从中断处理程序返回用户空间），如果need resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占

Scheduling Criteria

1. CPU utilization – keep the CPU as busy as possible
2. Throughput – # of processes that complete their execution per time unit
3. Turnaround time – amount of time to execute a particular process
4. Waiting time – amount of time a process has been waiting in the ready queue
5. Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling as Classification/Prediction

分类一

- I/O-bound: I/O操作频繁, 花费很多时间等待I/O操作的完成, 不需要太长时间片
- CPU-bound: 计算密集型, 需要大量的CPU时间进行运算

分类二

- 交互式进程：响应时间快，shell，图形界面
- 批处理进程：完成时间要短，编译程序
- 实时进程：响应时间要短、稳定，不被低优先级进程阻塞，视频/音频、自动化控制

分类三

- CPU burst: performing calculations
- I/O burst: waiting for data transfer in or out of the system
- process as distribution over CPU-I/O burst.

CPU Scheduler

Linux Scheduler History

Linux Scheduling Pre-2.6

- Pre-2.6 Linux systems used a scheduler that scaled poorly, requiring O(n) time to schedule tasks where n is the number of runnable threads.

The Linux 2.6 O(1) Scheduler

- For Linux 2.6, Ingo Molnar, the Linux kernel scheduler maintainer, implemented a new O(1) scheduler to address the scalability problems inherent to the earlier approach.
- The O(1) scheduler combines a static and dynamic priority.
  - Static priority: set by the user or system using nice.
  - Dynamic priority: a potential "boost" to the static priority intended to reward interactive threads.

Problems With The O(1) Scheduler: The interactivity boost code got very complex and difficult to understand. It was not clear that it was doing the right thing. Not surprisingly, the scheduler was also difficult to model, useful in order to convince yourself that it is doing "the right thing". (Geoffrey Challen)

Structure of the Linux scheduler

`kernel/sched/core.c`

core underlying scheduler mechanism, includes `schedule()`

Several schedule classes/policies on top of core.c

1. kernel/sched/rt.c: real-time scheduler
2. kernel/sched/fair.c: completely fair scheduling (CFS) class
3. kernel/sched/idle.c: simple, special scheduling class for the per-CPU idle tasks

Add a new scheduling class if you want a new scheduler.

__schedule(): the main scheduler function

```
static void __sched __schedule(bool preempt)
{
    ...
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;
    /*prev is going to be replaced, some accounting and statistics*/
    put_prev_task(rq, prev);
    rq_lock(rq, &rf);
    ...
    next = pick_next_task(rq, prev, &rf);
    ...
    if (likely(prev != next)) {
        ...
        rq->nr_switches++;
        rq->curr = next;
        ...
        /* Also unlocks the rq: */
        rq = context_switch(rq, prev, next, &rf);
    }
    ...
}
```

__sched: the prefix

```
void __sched some_function(...)
{
    ...
    schedule();
    ...
}
```

The purpose of the prefix is to put the compiled code of the function into a special section of the object file, .sched.text. This information enables the kernel to ignore all scheduling-related calls when a stack dump or similar information needs to be shown. Since the scheduler function calls are not part of the regular code flow, they are of no interest in such cases.

pick_next_task

```c
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        const struct sched_class *class;
        struct task_struct *p;

        /* Optimization: we know that if all tasks are in the fair class we can
         * call that function directly, but only if the @prev task wasn't of a
         * higher scheduling class, because otherwise those loose the
         * opportunity to pull in more work from other CPUs.
         */
        if (likely((prev->sched_class == &idle_sched_class ||
                    prev->sched_class == &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_running)) {

                p = fair_sched_class.pick_next_task(rq, prev, rf);
                if (unlikely(p == RETRY_TASK)) goto again;
                /* Assumes fair_sched_class->next == idle_sched_class */
                if (unlikely(!p))
                        p = idle_sched_class.pick_next_task(rq, prev, rf);
                return p;
        }

again:
        for_each_class(class) {
                p = class->pick_next_task(rq, prev, rf);
                if (p) {
                        if (unlikely(p == RETRY_TASK))
                                goto again;
                        return p;
                }
        }
        /* The idle class should always have a runnable task: */
        BUG();
}
```

scheduler_tick: the periodic scheduler

```
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)
static void put_prev_task(struct rq *rq, struct task_struct *prev)
{
    ...
    prev->sched_class->put_prev_task(rq, prev);
}
/*gets called by the timer code, with HZ frequency.*/
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    ...
    /*updates the clock of the run queue*/
    update_rq_clock(rq);
    /*checking if the current thread is running for too long,
     *if it is, setting a flag that indicates __schedule()
     * must be called to replace the running task */
    curr->sched_class->task_tick(rq, curr, 0);
    ...
}
```

Scheduling class

kernel/sched/sched.h

defines the scheduling class. A scheduling class is a set of function pointers, defined through `struct sched_class`. Scheduling classes allow for implementing these algorithms in a modular way.

```
struct sched_class {
    const struct sched_class *next;
    ...
    /*put sth on the run queue*/
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /*take sth off the run queue*/
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /*key schedule func. to select the next task*/
    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);
    ...
};
```

Scheduling class II

New scheduling class has to implement and set these function pointers.

```
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,
    ...
};
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,
    ...
};
```

Run queues

单个运行队列

- Linux2.4 O(n)调度器，适用于单CPU系统
- 多CPU系统存在并发访问，需要同步/lock, lock contetion

多个运行队列

- per-CPU run queues，没有race condition（禁止抢占）

```
DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
struct rq {
    unsigned long nr_running;
    ...
    struct cfs_rq cfs; /*class specific run queues*/
    struct rt_rq rt;
    struct task_struct *curr, *idle, *stop;
    ...
    int cpu;
    u64 clock;
    ...
};
```

每个sched_class使用自己的运行队列，新的sched_class也需要在struct rq中添加自己的运行队列struct xxx_rq xxx。

Scheduling Entities

A scheduling entity could be a task, or a group of tasks.

```
struct sched_entity {
    /* for load-balancing */
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 vruntime;
    u64 prev_sum_exec_runtime;
    u64 nr_migrations;
    ...
};
```

```
struct task_struct {
    ...
    int prio,
    static_prio,
    normal_prio;
    unsigned int rt_priority;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
    unsigned int policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice;
    ...
}
```

CPU Scheduler

idle_sched_class

the class set 12 function pointers out of 23 function pointers: not all function pointers defined in sched.h are required.

```
const struct sched_class idle_sched_class = {
        /* .next is NULL */
        /* no enqueue/yield_task for idle tasks */

        /* dequeue is not valid, we print a debug message there: */
        .dequeue_task                   = dequeue_task_idle,

        .pick_next_task                   = pick_next_task_idle,
        .put_prev_task                   = put_prev_task_idle,

#ifdef CONFIG_SMP
        .select_task_rq                   = select_task_rq_idle,
        .set_cpus_allowed        = set_cpus_allowed_common,
#endif

        .set_curr_task          = set_curr_task_idle,
        .task_tick                = task_tick_idle,
        ...
};
```

kernel/sched/idle.c

```
static struct task_struct *
pick_next_task_idle(struct rq *rq,
                    struct task_struct *prev,
                    struct rq_flags *rf)
{
        put_prev_task(rq, prev);
        update_idle_core(rq);
        schedstat_inc(rq->sched_goidle);

        return rq->idle; /*always there*/
}

/* It is not legal to dequeque the idle task - print a warning message*/
static void
dequeue_task_idle(struct rq *rq, struct task_struct *p, int flags)
{
        raw_spin_unlock_irq(&rq->lock);
        printk(KERN_ERR "bad: scheduling from the idle thread!\n");
        dump_stack();
        raw_spin_lock_irq(&rq->lock);
}

static void put_prev_task_idle(struct rq *rq, struct task_struct *prev)
{
}
```

CPU Scheduler

The RT scheduler

If a real-time process exists in the system and is runnable, it will always be selected by the scheduler — unless there is another real-time process with a higher priority.
The RT scheduler supports the following scheduling policies:

- Round robin processes : `SCHED_RR` have a time slice whose value is reduced when they run if they are normal processes. Once all time quantums have expired, the value is reset to the initial value, but the process is placed at the end of the queue. This ensures that if there are several `SCHED_RR` processes with the same priority, they are always executed in turn.
- First-in, first-out processes: `SCHED_FIFO` do not have a time slice and are permitted to run as long as they want once they have been selected.

The RT scheduler: priority

```
<sched/prio.h>

#define MAX_NICE        19
#define MIN_NICE        -20
#define NICE_WIDTH        (MAX_NICE - MIN_NICE + 1)

#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO                MAX_USER_RT_PRIO
#define MAX_PRIO                (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO                (MAX_RT_PRIO + NICE_WIDTH / 2)

/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)        ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio)        ((prio) - DEFAULT_PRIO)
```

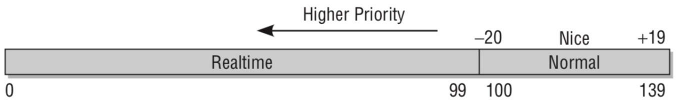

Figure 1: Kernel priority scale

The RT scheduler data structures

The RT scheduler: 250 lines of code compared to 1,100 for CFS.

```
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;
    ...
}
```

`active.queue[x]`: all real-time threads whose priority is x.

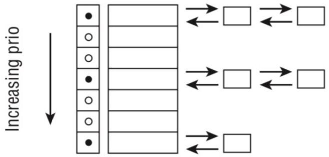

Figure 2: Run queue of the real-time scheduler.

SCHED_FIFO and SCHED_RR difference

```
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);
        update_rt_rq_load_avg(rq_clock_task(rq), rq, 1);
        watchdog(rq, p);

        /* RR tasks need a special form of timeslice management.
         * FIFO tasks have no timeslices, can run until stop or yield, nothing to do
         */
        if (p->policy != SCHED_RR)
                return;

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

        /* Requeue SCHED_RR threads to the end of queue if time slice becomes zero
         */
        p->rt.time_slice = sched_rr_timeslice;
        for_each_sched_rt_entity(rt_se) {
                if (rt_se->run_list.prev != rt_se->run_list.next) {
                        requeue_task_rt(rq, p, 0);
                        resched_curr(rq);
                        return;
                }
        }
}
```

pick_next_task_rt

```
static struct task_struct *
pick_next_task_rt(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        struct task_struct *p;
        struct rt_rq *rt_rq = &rq->rt;

        /* We may dequeue prev's rt_rq in put_prev_task().
         * So, we update time before rt_nr_running check.
         */
        if (prev->sched_class == &rt_sched_class)
                update_curr_rt(rq);

        if (!rt_rq->rt_queued)
                return NULL;

        put_prev_task(rq, prev);

        p = _pick_next_task_rt(rq);

        /* The running task is never eligible for pushing */
        dequeue_pushable_task(rq, p);
        rt_queue_push_tasks(rq);

        return p;
}
```

pick_next_task_rt

```
static struct task_struct *_pick_next_task_rt(struct rq *rq)
{
        struct sched_rt_entity *rt_se;
        struct task_struct *p;
        struct rt_rq *rt_rq  = &rq->rt;

        do {
                rt_se = pick_next_rt_entity(rq, rt_rq);
                BUG_ON(!rt_se);
                rt_rq = group_rt_rq(rt_se);
        } while (rt_rq);
        p = rt_task_of(rt_se);
        p->se.exec_start = rq_clock_task(rq);
        return p;
}
static struct sched_rt_entity *pick_next_rt_entity(struct rq *rq,
                                                   struct rt_rq *rt_rq)
{
        struct rt_prio_array *array = &rt_rq->active;
        struct sched_rt_entity *next = NULL;
        struct list_head *queue;
        int idx;

        idx = sched_find_first_bit(array->bitmap);
        BUG_ON(idx >= MAX_RT_PRIO);

        queue = array->queue + idx;
        next = list_entry(queue->next, struct sched_rt_entity, run_list);
        return next;
}
```

CPU Scheduler

Completely Fair Scheduler (CFS)

Completely Fair Scheduler

- introduced and maintained by Ingo Molnar in 2.6.23
- based on RSDL by Con Kolivas(an Australian anaesthetist)

CFS scheduling algorithm

Pick the task with the smallest `vruntime`.

`vruntime`

the virtual runtime of a process, or the actual runtime (the amount of time spent running) normalized (or weighted) by the number of runnable processes.

CFS data structures

```
const struct sched_class fair_sched_class = {
    .next           = &idle_sched_class,
    ...
    .enqueue_task = enqueue_task_fair,
    .dequeue_task = dequeue_task_fair,
    .pick_next_task     = pick_next_task_fair,
    .put_prev_task      = put_prev_task_fair,
    ...
    .task_tick      = task_tick_fair,
    .check_preempt_curr = check_preempt_wakeup,
    ...
};
struct cfs_rq {
    struct load_weight load;
    unsigned long nr_running;
    u64 min_vruntime;
    struct rb_root tasks_timeline; /*red black tree sorted by vruntime*
    struct rb_node *rb_leftmost;
    struct sched_entity *curr;
}
```

CFS time accounting

```
/*
* scheduler tick hitting a task of our scheduling class:
*/
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }
}

static void entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
    /*
    * Update run-time statistics of the 'current'.
    */
    update_curr(cfs_rq);
    /*
    * Update share accounting for long-running entities.
    */
    update_entity_shares_tick(cfs_rq);

    if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
        check_preempt_tick(cfs_rq, curr);
}
```

CFS time accounting

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;
        struct sched_entity *se;
        s64 delta;

        ideal_runtime = sched_slice(cfs_rq, curr);
        delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
        if (delta_exec > ideal_runtime) {
                resched_curr(rq_of(cfs_rq));
                /* The current task ran long enough, ensure it doesn't get
                 * re-elected due to buddy favours.
                 */
                clear_buddies(cfs_rq, curr);
                return;
        }
        if (delta_exec < sysctl_sched_min_granularity)
                return;

        se = __pick_first_entity(cfs_rq);
        delta = curr->vruntime - se->vruntime;

        if (delta < 0)
                return;
        if (delta > ideal_runtime)
                resched_curr(rq_of(cfs_rq));
}
```

CFS time accounting

Update the current task's runtime statistics.

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_clock_task(rq_of(cfs_rq));
        u64 delta_exec;

        if (unlikely(!curr))
                return;

        delta_exec = now - curr->exec_start;
        curr->exec_start = now;

        schedstat_set(curr->statistics.exec_max,
                      max(delta_exec, curr->statistics.exec_max));

        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq, exec_clock, delta_exec);

        curr->vruntime += calc_delta_fair(delta_exec, curr);
        update_min_vruntime(cfs_rq);

        if (entity_is_task(curr)) {
                struct task_struct *curtask = task_of(curr);
                trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
                cpuacct_charge(curtask, delta_exec);
                account_group_exec_runtime(curtask, delta_exec);
        }
        account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

CFS: pick the next task

```
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;
}

static struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
    struct rb_node *left = cfs_rq->rb_leftmost;
    if (!left)
        return NULL;
    return rb_entry(left, struct sched_entity, run_node);
}
```

CFS: adding process to the tree

```c
static void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /*
    * Update the normalized vruntime before updating min_vruntime
    * through callig update_curr().
    */
    if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
        se->vruntime += cfs_rq->min_vruntime;
    /*
    * Update run-time statistics of the 'current'.
    */
    update_curr(cfs_rq);
    update_cfs_load(cfs_rq, 0);
    account_entity_enqueue(cfs_rq, se);
    update_cfs_shares(cfs_rq);
    if (flags & ENQUEUE_WAKEUP) {
        place_entity(cfs_rq, se, 0);
        enqueue_sleeper(cfs_rq, se);
    }
    update_stats_enqueue(cfs_rq, se);
    check_spread(cfs_rq, se);
    if (se != cfs_rq->curr)
        __enqueue_entity(cfs_rq, se);
    se->on_rq = 1;
    if (cfs_rq->nr_running == 1)
        list_add_leaf_cfs_rq(cfs_rq);
}
```

CFS: adding process to the tree

```
/* Enqueue an entity into the rb-tree: */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        struct rb_node **link = &cfs_rq->tasks_timeline.rb_root.rb_node;
        struct rb_node *parent = NULL;
        struct sched_entity *entry;
        bool leftmost = true;

        /*
         * Find the right place in the rbtree:
         */
        while (*link) {
                parent = *link;
                entry = rb_entry(parent, struct sched_entity, run_node);
                /*
                 * We dont care about collisions. Nodes with
                 * the same key stay together.
                 */
                if (entity_before(se, entry)) {
                        link = &parent->rb_left;
                } else {
                        link = &parent->rb_right;
                        leftmost = false;
                }
        }

        rb_link_node(&se->run_node, parent, link);
        rb_insert_color_cached(&se->run_node,
                               &cfs_rq->tasks_timeline, leftmost);
}
```

CFS: remove process from the tree

```
static void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
        /*Update run-time statistics of the 'current'.*/
        update_curr(cfs_rq);

        update_stats_dequeue(cfs_rq, se, flags);
        clear_buddies(cfs_rq, se);

        if (se != cfs_rq->curr)
                __dequeue_entity(cfs_rq, se);
        se->on_rq = 0;
        account_entity_dequeue(cfs_rq, se);

        /* Normalize after update_curr(); which will also have moved
         * min_vruntime if @se is the one holding it back. But before doing
         * update_min_vruntime() again, which will discount @se's position and
         * can move min_vruntime forward still more.
         */
        if (!(flags & DEQUEUE_SLEEP))
                se->vruntime -= cfs_rq->min_vruntime;

        if ((flags & (DEQUEUE_SAVE | DEQUEUE_MOVE)) == DEQUEUE_SAVE)
                update_min_vruntime(cfs_rq);
}
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        rb_erase_cached(&se->run_node, &cfs_rq->tasks_timeline);
}
```

sleeping and waking

Sleeping via wait queues: A wait queue is a list of processes waiting for an event to occur, wait queues are represented in the kernel by `wake_queue_head_t`.

- created statically via `DECLARE_WAITQUEUE()`
- created dynamically via `init_waitqueue_head()`

Processes put themselves on a wait queue and mark themselves not runnable. When the event associated with the wait queue occurs, the processes on the queue are awakened.

```
/* 'q' is the wait queue we wish to sleep on */
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        /* handle signal */
    schedule();
}
finish_wait(&q, &wait);
```

sleeping and waking

Sleeping via wait queues

1. Creates a wait queue entry via the macro `DEFINE_WAIT()`.
2. Adds itself to a wait queue via `add_wait_queue()`. This wait queue awakens the process when the condition for which it is waiting occurs. Of course, there needs to be code elsewhere that calls `wake_up()` on the queue when the event actually does occur.
3. Calls `prepare_to_wait()` to change the process state to either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`. This function also adds the task back to the wait queue if necessary, which is needed on subsequent iterations of the loop.
4. If the state is set to `TASK_INTERRUPTIBLE`, a signal wakes the process up. This is called a spurious wake up (a wake-up not caused by the occurrence of the event). So check and handle signals.
5. When the task awakens, it again checks whether the condition is true. If it is, it exits the loop. Otherwise, it again calls `schedule()` and repeats.

sleeping example

```
static void snd_usbmidi_output_drain(struct snd_rawmidi_substream *substream)
{
        struct usbmidi_out_port *port = substream->runtime->private_data;
        struct snd_usb_midi_out_endpoint *ep = port->ep;
        unsigned int drain_urbs;
        DEFINE_WAIT(wait);
        long timeout = msecs_to_jiffies(50);

        if (ep->umidi->disconnected)
                return;
        /*
         * The substream buffer is empty, but some data might still be in the
         * currently active URBs, so we have to wait for those to complete.
         */
        spin_lock_irq(&ep->buffer_lock);
        drain_urbs = ep->active_urbs;
        if (drain_urbs) {
                ep->drain_urbs |= drain_urbs;
                do {
                        prepare_to_wait(&ep->drain_wait, &wait,
                                        TASK_UNINTERRUPTIBLE);
                        spin_unlock_irq(&ep->buffer_lock);
                        timeout = schedule_timeout(timeout);
                        spin_lock_irq(&ep->buffer_lock);
                        drain_urbs &= ep->drain_urbs;
                } while (drain_urbs && timeout);
                finish_wait(&ep->drain_wait, &wait);
        }
        spin_unlock_irq(&ep->buffer_lock);
}
```

scheduler related system calls

```
nice();                        /*Sets a process's nice value*/
sched_setscheduler();          /*Sets a process's scheduling policy*/
sched_getscheduler();          /*Gets a process's scheduling policy*/
sched_setparam();              /*Sets a process's real-time priority*/
sched_getparam();              /*Gets a process's real-time priority*/
sched_get_priority_max();      /*Gets the maximum real-time priority*/
sched_get_priority_min();      /*Gets the minimum real-time priority*/
sched_rr_get_interval();       /*Gets a process's timeslice value*/
sched_setaffinity();          /*Sets a process's processor affinity*/
sched_getaffinity();          /*Gets a process's processor affinity*/
sched_yield();                 /*Temporarily yields the processor*/
```

CPU Scheduler

总结

Linux schedulers:

- Real-Time Scheduler:
  - SCHED_RR
  - SCHED_FIFO
- Completely Fair Scheduler:
  - SCHED_BATCH: 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
- Idle Scheduler:
  - SCHED_IDLE
- ...

Thanks!

基础实验楼乙126