Report on Project 2

6/14/2018

516030910306 Hongyi Guo (Henry Guo)

1.include/linux/sched.h

(1) Define WRR scheduler as 6

```
#define SCHED_WRR		6 /* added by henry */
```

(2) Define wrr_info, to store information of CPU, for multi-CPUs.

```
#define MAX_CPUS 8
struct wrr_info {
	int num_cpus;
	int nr_running[MAX_CPUS];
	int total_weight[MAX_CPUS];
};

```

(3) Declare wrr_rq, run queue of wrr scheduler.

```
struct wrr_rq; /* added by henry */
```

(4) Define wrr scheduler entity. Run_list is the list of tasks of this entity. Time slice depends on whether this process is a foreground/background process.

```
/* added by henry */
struct sched_wrr_entity {
	struct list_head run_list;
	unsigned int time_slice;

#ifdef CONFIG_SMP
	int weight;
#endif

#ifdef CONFIG_WRR_GROUP_SCHED
	struct sched_wrr_entity *parent;
	/* rq on which this entity is (to be) queued: */
	struct wrr_rq		*wrr_rq;
	/* rq "owned" by this entity/group: */
	struct wrr_rq		*my_q;
#endif
};
```

(5) Define the time slice for wrr scheduler, 100 for foreground process and 10 for background processes.

```
/*
 * added by henry, used only for SCHED_WRR tasks
 * for foreground task, timeslice is 100 msecs
 * for background task, timeslice is 10 msecs
 */
#define WRR_TIMESLICE_FORE  (100 * HZ / 1000)
#define WRR_TIMESLICE_BACK  (10 * HZ / 1000)

```

(6) Add wrr_priority and wrr scheduler entity in task_struct

```
    unsigned int wrr_priority; /* added by henry */
```

```
    struct sched_wrr_entity wrr; /* added by henry */
```

---

2. kernel/sched/sched.h

(1) Define whether to use multi-queues on priority, and whether to print extra debugging information.

```
#define CONFIG_WRR_PRIO 1
// #define CONFIG_WRR_DEBUG 1
```

(2) If we want to take multi-CPU into consideration, then declare extern struct and spinlock.

```
/* by henry */
#ifdef CONFIG_SMP
extern struct wrr_info my_wrr;
extern raw_spinlock_t wrr_info_locks[MAX_CPUS];
#endif
```

(3) Define priority array, who has up to MAX_RT_PRIO queues. Its bitmap indicates whether the corresponding queue is empty(0) or not(1).

```c
/*
 * by henry
 *
 * This is the priority-queue data structure of the WRR scheduling class:
 */
#ifdef CONFIG_WRR_PRIO
struct wrr_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
    struct list_head queue[MAX_RT_PRIO];
};
#endif
```

(4) Declare runqueue of wrr scheduler.

```c
struct wrr_rq; /* added by henry */
```

(5) Define struct wrr_rq. "wrr_nr_running" is the tasks running on all wrr runqueues.

In priority-based multi-queue mode, "active" is an array of wrr_runqueue's.

In single-queue mode, one list is defined by its head "queue".

If SMP is defined, define wrr_nr_total as the total weight of the tasks in this runqueue.

```c
////////////////////// This is the part to study ///////////////////////

/*
 * added by henry
 * Weighted Round-Robin classes' related field in a runqueue:
 */
struct wrr_rq {
#ifdef CONFIG_WRR_PRIO
    struct wrr_prio_array active;
#else
    struct list_head queue;
#endif
    unsigned int wrr_nr_running;

#ifdef CONFIG_SMP
    unsigned long wrr_nr_migratory;
    unsigned long wrr_nr_total;
    int overloaded;
    struct plist_head pushable_tasks;
#endif
};

```

(6) Define a wrr_rq named wrr in struct rq.

```c
    struct wrr_rq wrr; /* added by henry */
```

(7) Declare extern wrr_sched_class.

```
extern const struct sched_class wrr_sched_class; /* added by henry */
```

(8) Declare extern idle balance function "idle_balance_wrr". Note this is active only when CONFIG_SMP defined.

```
extern void idle_balance_wrr(struct rq *this_rq);
```

(9) Declare extern initialization function "init_sched_wrr_class" defined in wrr.c.

```
extern void init_sched_wrr_class(void); /* by henry */
```

(10) Declare extern runqueue initialization function "init_wrr_rq" defined in wrr.c.

```
extern void init_wrr_rq(struct wrr_rq *wrr_rq, struct rq *rq); /* by henry */
```

---

3. kernel/sched/core.c

(1) Define wrr_info struct to store information of wrr scheduler, only used when CONFIG_SMP is defined.

```
/* by henry */
#ifdef CONFIG_SMP
struct wrr_info my_wrr = {
    .num_cpus = 0,
    .nr_running = { 0 },
    .total_weight = { 0 },
};
#endif
```

(2) Call "INIT_LIST_HEAD" to initialize sched_wrr_entity in "__sched_fork" function.

```
    INIT_LIST_HEAD(&p->wrr.run_list); /* by henry */
```

(3) If CONFIG_SMP defined, call idle balance function of wrr scheduler when no wrr task is running on this rq.

```
    /* by henry */
#ifdef CONFIG_SMP
    if (unlikely(!rq->wrr.wrr_nr_running))
        idle_balance_wrr(rq);
#endif
```

(4) Defined in __sched_setscheduler.

```
    printk("I AM IN __SCHED_SETSCHEDULER\n");
```

Take SCHED_WRR into consideration.

```
	if (policy != SCHED_FIFO && policy != SCHED_RR &&
	    policy != SCHED_NORMAL && policy != SCHED_BATCH &&
	    policy != SCHED_IDLE && policy != SCHED_WRR)
	    return -EINVAL;
}
```

```
if ((rt_policy(policy) || (policy == SCHED_WRR))
    != (param->sched_priority != 0)) {
    printk("recheck 2 failed\n");
    return -EINVAL;
}
```

(5) Defined in "__setscheduler", which do the actual job of priority change. Set "sched_class" of "p" to "wrr_sched_class" if new policy is "SCHED_WRR", and set wrr_priority.

```
/* Actually do priority change: must hold rq lock. */
static void
__setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
{
#ifdef CONFIG_SCHED_DEBUG
    // printk("__setscheduler called\n");
    printk("[%d]\tpriority set to %d\n", p->pid, prio);
#endif
    p->policy = policy;
    p->rt_priority = prio;
    p->wrr_priority = prio; /* by henry */
    p->normal_prio = normal_prio(p);
    /* we are holding p->pi_lock already */
    p->prio = rt_mutex_getprio(p);

    if (policy == SCHED_WRR) { /* by henry */
        p->sched_class = &wrr_sched_class;
#ifdef CONFIG_SCHED_DEBUG
        printk("set task scheduler as wrr\n");
#endif
    }
    else if (rt_prio(p->prio)) {
        p->sched_class = &rt_sched_class;
#ifdef CONFIG_SCHED_DEBUG
        printk("set task scheduler as rt\n");
#endif
    }
    else {
        p->sched_class = &fair_sched_class;
#ifdef CONFIG_SCHED_DEBUG
        printk("set task scheduler as fair\n");
#endif
    }
    set_load_weight(p);
```

(6) A little bit revise in "void __init sched_init(void)", to initialize wrr runqueue at beginning, and calculate total number of possible CPUs.

```
	for_each_possible_cpu(i) {
		struct rq *rq;

		rq = cpu_rq(i);
		raw_spin_lock_init(&rq->lock);
		rq->nr_running = 0;
		rq->calc_load_active = 0;
		rq->calc_load_update = jiffies + LOAD_FREQ;
		init_cfs_rq(&rq->cfs);
		init_rt_rq(&rq->rt, rq);
		init_wrr_rq(&rq->wrr, rq); /*by henry */

#ifdef CONFIG_SMP
		my_wrr.num_cpu++;
#endif
```

---

4. /kernel/sched/rt.c

(1) A little bit revise, add wrr_sched_class into the circular list.

```
const struct sched_class rt_sched_class = {
	.next			= &wrr_sched_class,
```

---

5. kernel/sched/wrr.c

I write this part based on the provided "rt.c".

(1) Overview.

```c
const struct sched_class wrr_sched_class = {
	.next			=	&fair_sched_class,	/* never */

	/* [required] adding/removing a task to/from a priority array */
	.enqueue_task		=	enqueue_task_wrr,	/* required */
	.dequeue_task		=	dequeue_task_wrr,	/* required */

	.yield_task		=	yield_task_wrr,		/* required */
	.check_preempt_curr	=	check_preempt_curr_wrr,	/* required */

	.pick_next_task		=	pick_next_task_wrr,	/* required */
	.put_prev_task		=	put_prev_task_wrr,	/* required */

	.task_fork		=	task_fork_wrr,		/* required */

#ifdef CONFIG_SMP
	.select_task_rq		=	select_task_rq_wrr,	/* never */
	.set_cpus_allowed	=	set_cpus_allowed_wrr,	/* never */
	.rq_online		=	rq_online_wrr,		/* never */
	.rq_offline		=	rq_offline_wrr,		/* never */
	.pre_schedule		=	pre_schedule_wrr,	/* never */
	.post_schedule		=	post_schedule_wrr,	/* never */
	.task_woken		=	task_woken_wrr,		/* never */
#endif

	.switched_from		=	switched_from_wrr,	/* never */

	.set_curr_task		=	set_curr_task_wrr,	/* required */
	.task_tick		=	task_tick_wrr,		/* required */

	.get_rr_interval	=	get_rr_interval_wrr,	/* required */

	.prio_changed		=	prio_changed_wrr,	/* never */
	.switched_to		=	switched_to_wrr,	/* required */
};
```

(2) init_wrr_rq

To initialize wrr runqueue, we need to initialize every runqueue for each priority. Totally MAX_RT_PRIO+1 runqueues. Note that we need to set the last bit of array's bitmap into 1,for it's the delimiter for bitsearch.

```c
void init_wrr_rq(struct wrr_rq *wrr_rq, struct rq *rq)
{
#ifdef CONFIG_WRR_PRIO
    printk("init_wrr_rq called!\n");

    struct wrr_prio_array *array;
    int i;

    array = &wrr_rq->active;
    for (i = 0; i < MAX_RT_PRIO; i++) {
        INIT_LIST_HEAD(array->queue + i);
        __clear_bit(i, array->bitmap);
    }
    /* delimiter for bitsearch: */
    __set_bit(MAX_RT_PRIO, array->bitmap);
    wrr_rq->wrr_nr_running = 0;

#else /* !CONFIG_WRR_PRIO */

    INIT_LIST_HEAD(&wrr_rq->queue);
    wrr_rq->wrr_nr_running = 0;

#endif /* CONFIG_WRR_PRIO */
}
```

(3) pick_next_task_wrr

Use sched_find_first_bit function to find the first "1" in the bitmap, which is the non-empty queue with the highest priority.

```c
static struct task_struct *pick_next_task_wrr(struct rq *rq)
{
    struct task_struct *p;
    struct wrr_rq *wrr_rq;
    struct sched_wrr_entity *next = NULL;

    wrr_rq = &rq->wrr;

#ifdef CONFIG_WRR_PRIO

    struct wrr_prio_array *array = &wrr_rq->active;
    struct list_head *queue;
    int idx;

    if (!wrr_rq->wrr_nr_running)
        return NULL;

    idx = sched_find_first_bit(array->bitmap);

    BUG_ON(idx >= MAX_RT_PRIO);

    queue = array->queue + idx;
    next = list_entry(queue->next, struct sched_wrr_entity, run_list);

    BUG_ON(!next);
```

(4) enqueue_task_wrr

Enqueue the task into the corresponding queue of its priority, and set the same bit to 1 int the bitmap, indicating this queue is non-empty.

```
    struct sched_wrr_entity *wrr_se;
    struct wrr_rq *wrr_rq;

    wrr_se = &p->wrr;
    wrr_rq = &rq->wrr;

#ifdef CONFIG_WRR_PRIO
    int prio = wrr_se_prio(wrr_se);
    struct wrr_prio_array *array = &wrr_rq->active;
    struct list_head *queue = array->queue + prio;

    if (wrr_se == NULL)
        return;

    if (flags & ENQUEUE_HEAD) {
        list_add(&wrr_se->run_list, queue);
    }
    else {
        list_add_tail(&wrr_se->run_list, queue);
    }

    __set_bit(prio, array->bitmap);

    WARN_ON(!rt_prio(prio));

#else /* !CONFIG_WRR_PRIO */
```

```
    wrr_rq->wrr_nr_running++;
    inc_nr_running(rq);
```

If using multi-CPU, total weight should be updated after enqueuing.

```
#ifdef CONFIG_SMP
    cpu = cpu_of(rq);
    raw_spin_lock(&wrr_info_locks[cpu]);
    my_wrr.nr_running[cpu]++;
    my_wrr.total_weight[cpu] += wrr_se->weight;
    raw_spin_unlock(&wrr_info_locks[cpu]);
#endif
```

(5) dequeue_task_wrr

Basically the same with enqueue, but for checking whether the corresponding queue is empty, and if so, we need to clear the bit in the bitmap.

And when using multi-CPU, we need to decrease the weight of this CPU.

```c
    struct sched_wrr_entity *wrr_se = &p->wrr;
    struct wrr_rq *wrr_rq = &rq->wrr;

#ifdef CONFIG_WRR_PRIO
    int prio = wrr_se_prio(wrr_se);
    struct wrr_prio_array *array = &wrr_rq->active;
#endif

    if (wrr_se == NULL)
        return;

    list_del_init(&wrr_se->run_list); /* delete and initialize node */

#ifdef CONFIG_WRR_PRIO
    if (list_empty(array->queue + prio))
        __clear_bit(prio, array->bitmap);
    WARN_ON(!rt_prio(prio));
#endif

    WARN_ON(!wrr_rq->wrr_nr_running);
    wrr_rq->wrr_nr_running--;
    dec_nr_running(rq);

#ifdef CONFIG_SMP
    cpu = cpu_of(rq);
    raw_spin_lock(&wrr_info_locks[cpu]);
    my_wrr.nr_running[cpu]--;
    my_wrr.total_weight[cpu] -= wrr_se->weight;
    raw_spin_unlock(&wrr_info_locks[cpu]);
#endif
```

(6) requeue_task_wrr

Move the task into the end of this queue, usually called by “yield_task_wrr”.

Use “list_move” and “list_move_tail” function defined in list.h.

```c
    struct sched_wrr_entity *wrr_se;
    struct wrr_rq *wrr_rq;

    wrr_se = &p->wrr;
    wrr_rq = &rq->wrr;

#ifdef CONFIG_WRR_PRIO
    int prio = wrr_se_prio(wrr_se);
    struct wrr_prio_array *array = &wrr_rq->active;
    struct list_head *queue = array->queue + prio;

    if (wrr_se == NULL)
        return;
    if (head)
        list_move(&wrr_se->run_list, queue);
    else
        list_move_tail(&wrr_se->run_list, queue);
#else
```

(7) yield_task_wrr

To let current task on rq give out its time slice, we just move it to the end of the queue.

```
static void yield_task_wrr(struct rq *rq)
{
	requeue_task_wrr(rq, rq->curr, 0);
}
```

(8) task_tick_wrr

Decrease the time_slice of task p. If it run out of time slice, use "cgroup_path" to decide which time slice should be given to it.

```
/* for group path */
#ifndef PATH_MAX
#define PATH_MAX 4096
#endif
```

```
static char group_path[PATH_MAX];
static void task_tick_wrr(struct rq *rq, struct task_struct *p, int queued)
{
	struct sched_wrr_entity *wrr_se;
	wrr_se = &p->wrr;

	if (wrr_se == NULL) {
		return;
	}
	watchdog(rq, p);

	printk("[%d]\tprio is %d\n", p->pid, p->wrr_priority);
	printk("[%d]\t%d time slice left\n", p->pid, p->wrr.time_slice);

	if (--wrr_se->time_slice) {
		return;
	}

	cgroup_path(task_group(p)->css.cgroup, group_path, PATH_MAX);

	if (strcmp(group_path, "/") == 0) {
		wrr_se->time_slice = WRR_TIMESLICE_FORE;
#ifdef CONFIG_WRR_DEBUG
		printk("[%d]\ttime slice is set as FOREground\n", p->pid);
#endif
	} else {
		wrr_se->time_slice = WRR_TIMESLICE_BACK;
#ifdef CONFIG_WRR_DEBUG
		printk("[%d]\ttime slice is set as BACKground\n", p->pid);
#endif
	}
```

```
#ifdef CONFIG_WRR_PRIO
    /* decrease the priority */
    if (p->wrr_priority < MAX_RT_PRIO-1) {
        dequeue_task_wrr(rq, p, 1);
        p->wrr_priority++;
        enqueue_task_wrr(rq, p, 1);
    }
#endif

    if (wrr_se->run_list.prev != wrr_se->run_list.next) {
        requeue_task_wrr(rq, p, 0);
        set_tsk_need_resched(p);
        return;
    }
}
```

Each time task p run out of time slice, we degrade its priority by plus its wrr_priority by 1, and insert it into new queue.

(9) switched_to_wrr

When a task is switched to wrr, we distribute an initial time slice to it based on it being a foreground or background process.

```
static void switched_to_wrr(struct rq *rq, struct task_struct *p)
{
    int foreground;
    cgroup_path(task_group(p)->css.cgroup, group_path, PATH_MAX);
    foreground = (strcmp(group_path, "/") == 0);

    printk("group=%s\n", group_path);
    printk("Switched to a %s WRR entity, pid=%d, proc=%s\n",
        foreground ? "foreground" : "background", p->pid, p->comm);

    p->wrr.time_slice = foreground ? WRR_TIMESLICE_FORE : WRR_TIMESLICE_BACK;

    if (p->prio < rq->curr->prio)
        resched_task(rq->curr);
}
```

(10) get_rr_interval_wrr

```
static unsigned int get_rr_interval_wrr(struct rq *rq, struct task_struct *task)
{
    return task->wrr.time_slice;
}
```

(11) some other useful inline function.

```
static inline struct task_struct *wrr_task_of(struct sched_wrr_entity *wrr_se)
{
    return container_of(wrr_se, struct task_struct, wrr);
}

static inline int wrr_se_prio(struct sched_wrr_entity *wrr_se)
{
    return wrr_task_of(wrr_se)->wrr_priority;
}
```

(12) pick_pullable_task_wrr (when using multi-CPUs)

Check the first non-empty queue, if it has a process not running currently, then we're done, and this task can be pulled and insert to target CPU.

Otherwise, we check the next non-empty queue and on and on and on…

```
#ifdef CONFIG_SMP
static struct task_struct *pick_pullable_task_wrr(struct rq *src_rq,
                                                  int this_cpu)
{
    struct task_struct *p;
    struct list_head *queue;
    struct wrr_prio_array *array;
    struct sched_wrr_entity *pos;
    int idx;
    int flag; /* done or not */

    array = &src_rq->active;
    idx = sched_find_first_bit(array->bitmap);

    flag = 0;
    p = NULL;

    while (idx < MAX_RT_PRIO) {
        queue = array->queue + idx;
        list_for_each_entry(pos, queue, run_list) {
            p = wrr_task_of(pos);
            if (p == src_rq->curr)
                continue;
            if (cpumask_test_cpu(this_cpu, &p->cpus_allowed))
                flag = 1;
        }

        if (!flag)
            idx = find_next_bit(array->bitmap, MAX_RT_PRIO, idx+1);
        else
            break;
    }

    return p;
}
```

(13) idle_balance_wrr

Find the pullable task on a CPU which has more than 1 task. And move it to the new CPU where rq is on.

```
void idle_balance_wrr(struct rq *this_rq)
{
	int this_cpu = this_rq->cpu, cpu;
	struct task_struct *p;
	struct rq *src_rq;

	for_each_possible_cpu(cpu) {
		if (this_cpu == cpu)
			continue;

		src_rq = cpu_rq(cpu);
		double_lock_balance(this_rq, src_rq);

		if (src_rq->wrr.wrr_nr_running <= 1)
			goto skip;

		/* pick a task from src_rq */
		p = pick_pullable_task_wrr(src_rq, this_cpu);

		if (p) {
			if (p == src_rq->curr)
				goto skip;
			WARN_ON(!p->on_rq);

			deactivate_task(src_rq, p, 0);
			set_task_cpu(p, this_cpu);
			activate_task(this_rq, p, 0);
			double_unlock_balance(this_rq, src_rq);
			return;
		}
skip:
		double_unlock_balance(this_rq, src_rq);
	}
}
```

(14) select_task_rq_wrr

Find the CPU with the minimum weight, so, we can transfer tasks to this CPU.

```c
static int
select_task_rq_wrr(struct task_struct *p, int sd_flag, int flags)
{
    int cpu;
    int min_cpu = task_cpu(p);
    int min_weight = INT_MAX;

    if (p->nr_cpus_allowed == 1)
        goto out;

    /* For anything but wake ups, just return the task_cpu */
    if (sd_flag != SD_BALANCE_WAKE && sd_flag != SD_BALANCE_FORK)
        goto out;

    /* search for the cpu with minimized load weight */
    for_each_possible_cpu(cpu) {
        if (my_wrr.total_weight[cpu] < min_weight) {
            min_weight = my_wrr.total_weight[cpu];
            min_cpu = cpu;
        }
    }
    return min_cpu;
out:
    return min_cpu;
}

#else /* !CONFIG_SMP */
```