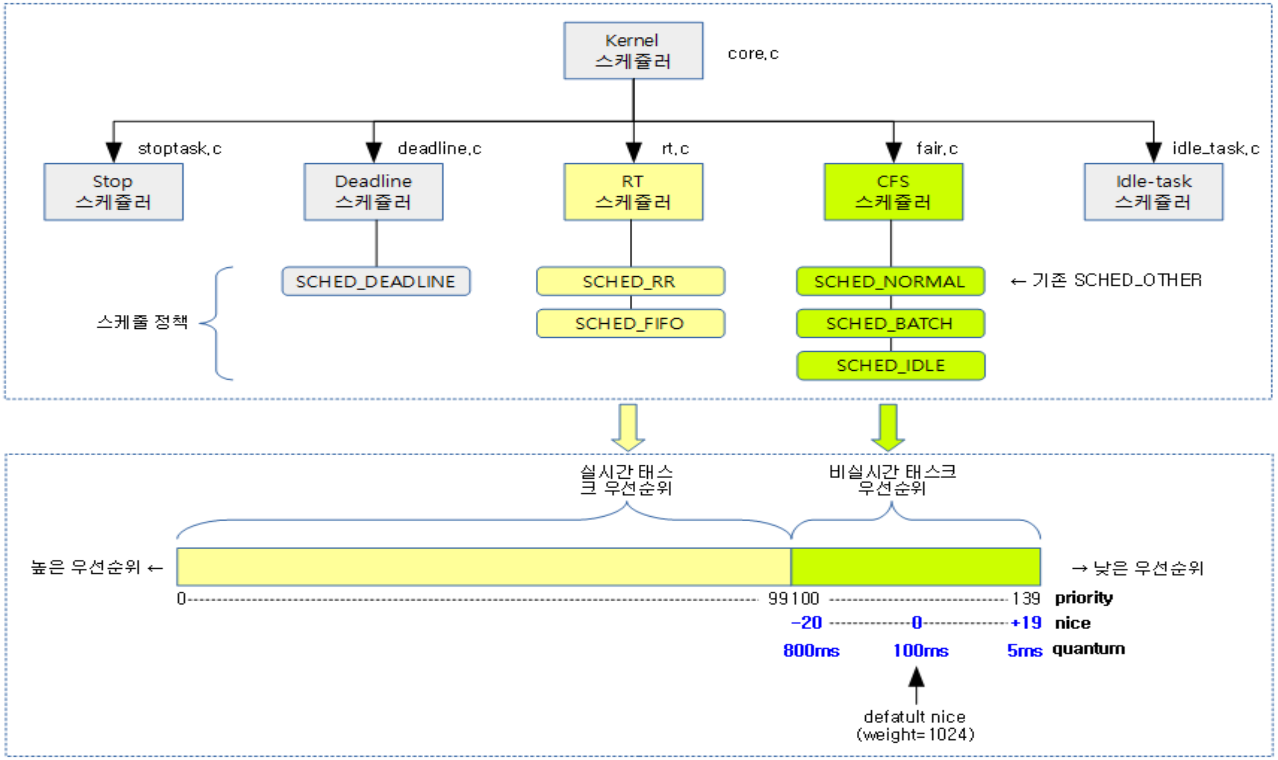
Scheduling, Device driver

오명훈
snt2426@gmail.com

Scheduler Class



- Advantage of scheduling classes
  - Linux supports two real-time scheduling classes
  - The structure of the scheduler enables real-time processes to be integrated into the kernel without any changes in the core scheduler

Data structure

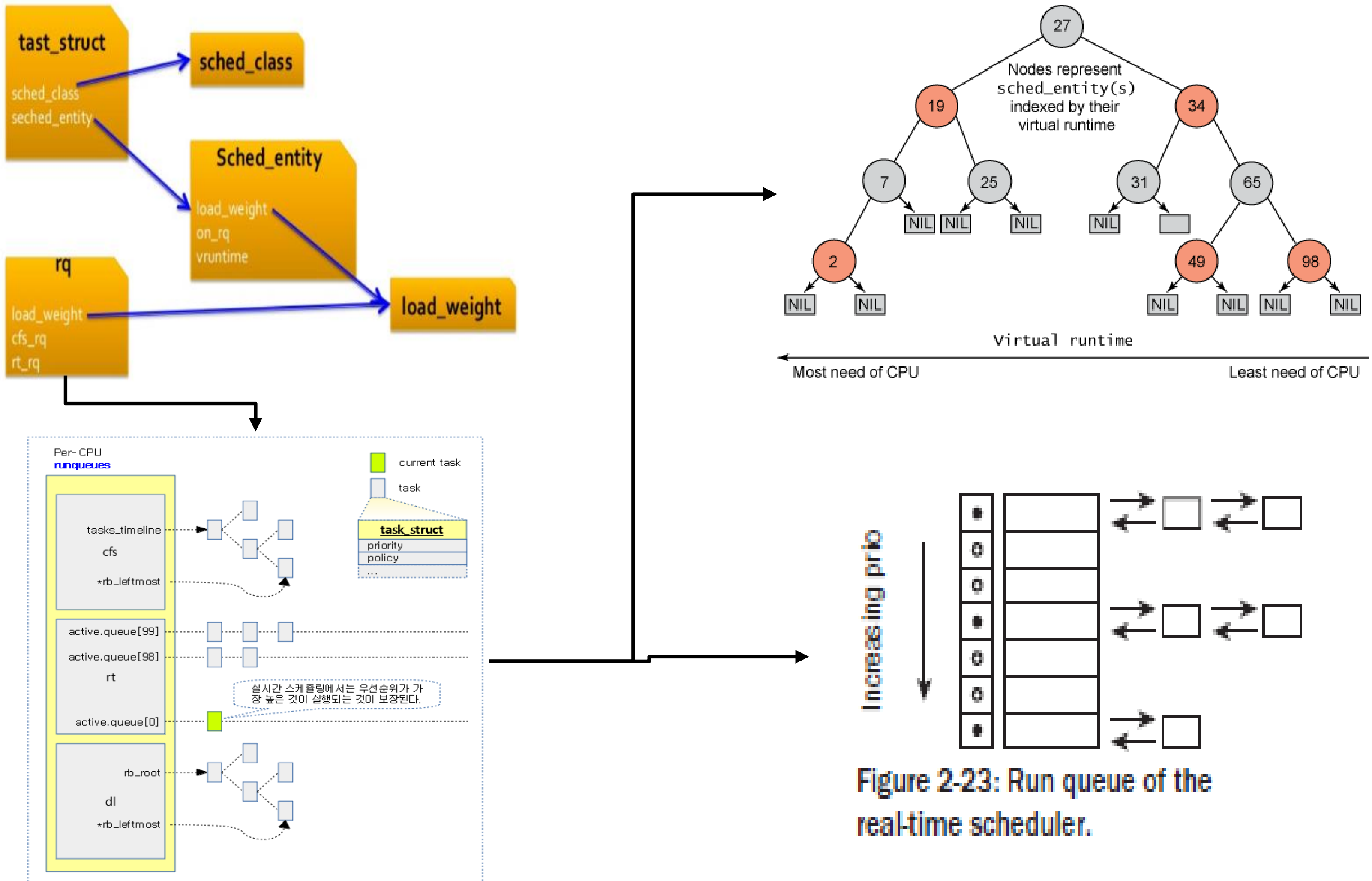
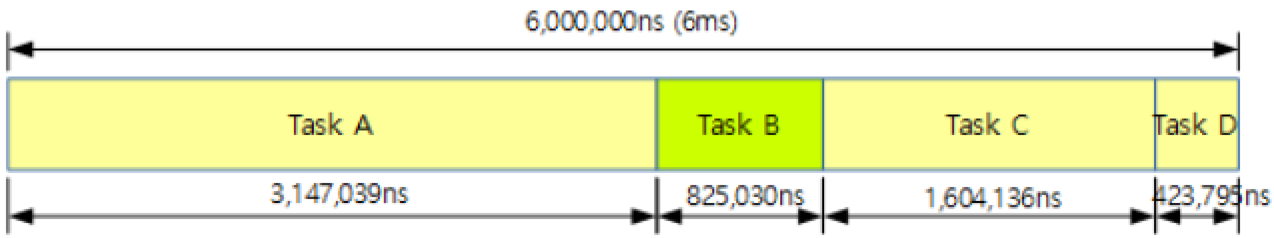


Figure 2-23: Run queue of the real-time scheduler.

Virtual time



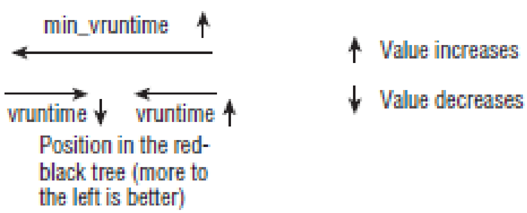
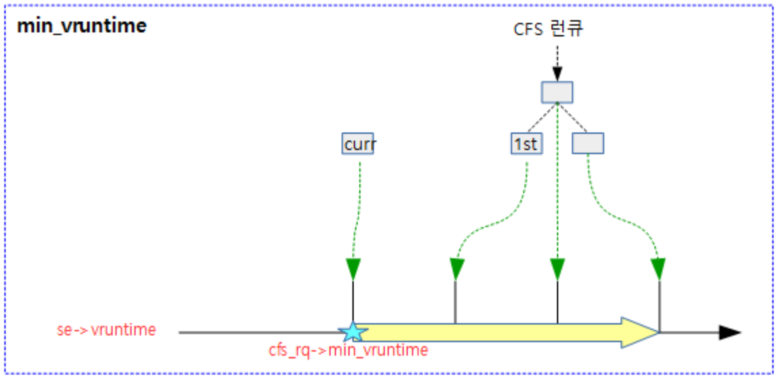


Figure 2-19: Influence of the per-entity and per-queue virtual times on the placement of processes in the red-black tree.



- vruntime
  - No time slice
  - When a different priority is used, the time must be weighted according to the load weight of the process
  - starvation problem, Fairness → min_vruntime
    - cfs 런큐에 새로 진입한 태스크에 min vruntime 값을 사용

CFS Operations

- update_curr

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

많은 스케쥴링 함수에서 수시로 쓰임

  - delta_exec_weighted

```
return mul_u64_u32_shr(delta_exec, fact, shift);
```

=

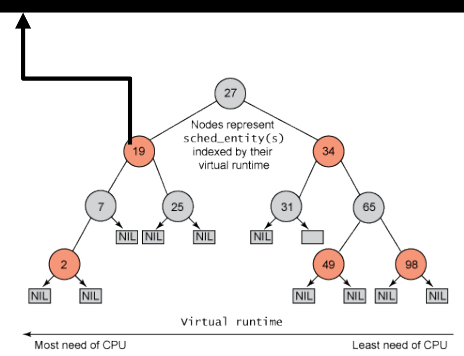
$$\text{delta\_exec\_weighted} = \text{delta\_exec} \times \frac{\text{NICE\_0\_LOAD}}{\text{curr->load.weight}}$$

  - update_min_vruntime

```
u64 vruntime = cfs_rq->min_vruntime;

if (curr) {
    if (curr->on_rq)
        vruntime = curr->vruntime;
    else
        curr = NULL;
}

if (cfs_rq->rb_leftmost) {
    struct sched_entity *se = rb_entry(cfs_rq->rb_leftmost,
                                       struct sched_entity,
                                       run_node);

    if (!curr)
        vruntime = se->vruntime;
    else
        vruntime = min_vruntime(vruntime, se->vruntime);
}

/* ensure we never gain time by being placed backwards. */
cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime);
```

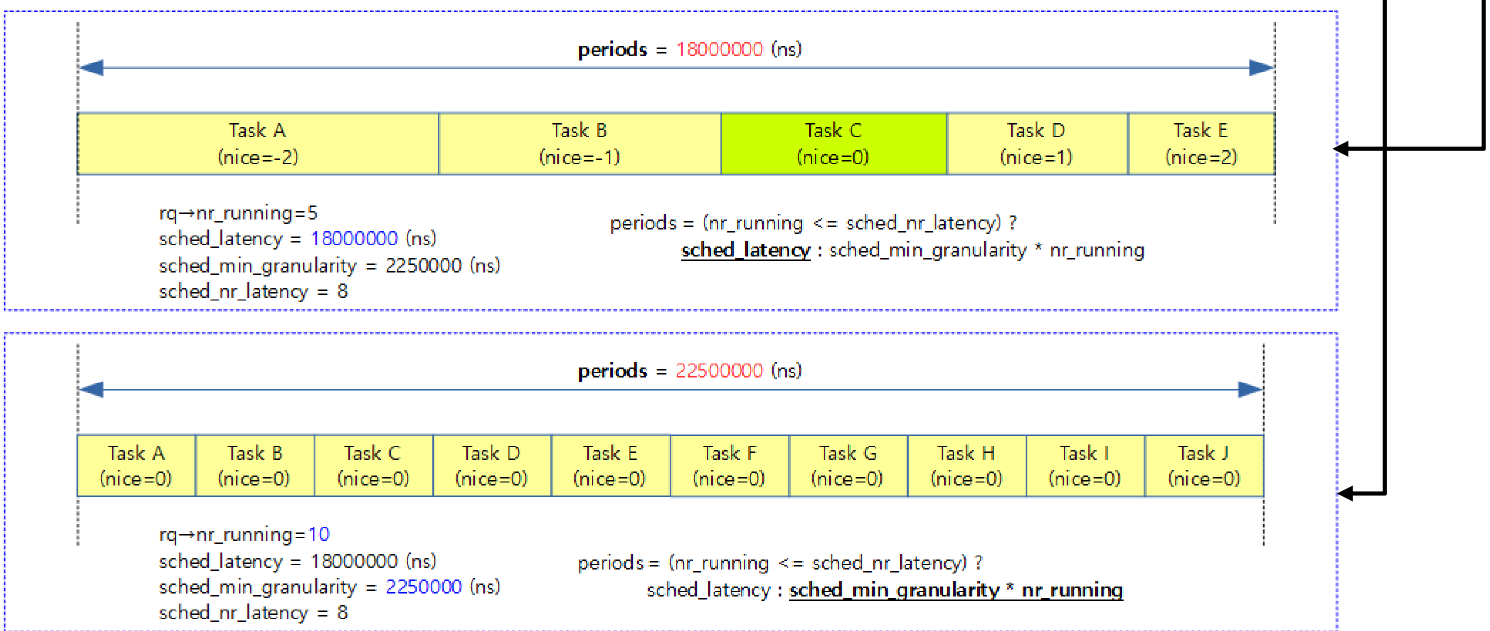
enqueue_entity

```
s64 d = se->vruntime - cfs_rq->min_vruntime;
```



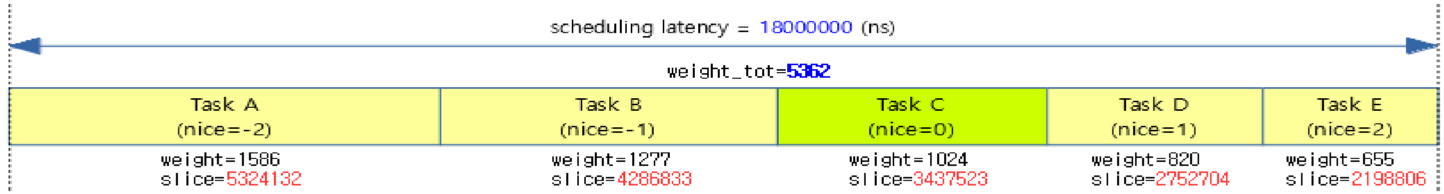
CFS Operations

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



- Latency
  - The interval during which every runnable task should run at least once
  - nr_running와 sched_nr_latency를 비교한 결과에 따라 두 가지 중 하나로 결정

CFS Operations



$$time\ slice = periods \times \frac{weight}{\mathrm{tot\_weight}}$$

kernel/sched_fair.c

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        u64 slice = __sched_period(cfs_rq->nr_running);

        slice *= se->load.weight;
        do_div(slice, cfs_rq->load.weight);

        return slice;
}
```

```
slice = __calc_delta(slice, se->load.weight, load);
```

kernel/sched_fair.c

```
static u64 __sched_vslice(unsigned long rq_weight, unsigned long nr_running)
{
        u64 vslice = __sched_period(nr_running);

        vslice *= NICE_0_LOAD;
        do_div(vslice, rq_weight);

        return vslice;
}

static u64 sched_vslice(struct cfs_rq *cfs_rq)
{
  return __sched_vslice(cfs_rq->load.weight, cfs_rq->nr_running);
}
```

```
static u64 sched_vslice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    return calc_delta_fair(sched_slice(cfs_rq, se), se);
}
```

- Distribution of the time among active processes in one latency period is performed by considering the relative weights of the respective tasks
- 독립된 소스 코드 X
  - Vruntime 메커니즘

CFS Operations

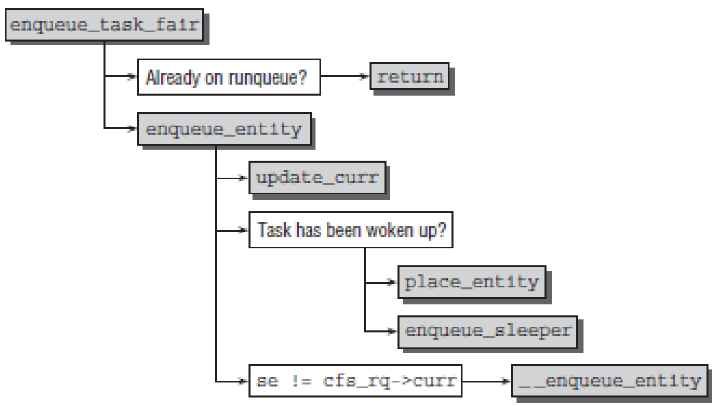


Figure 2-20: Code flow diagram for **enqueue_task_fair**.

```
u64 vruntime = cfs_rq->min_vruntime;

/*
 * The 'current' period is already promised to the current tasks,
 * however the extra weight of the new task will slow them down a
 * little, place the new task so that it fits in the slot that
 * stays open at the end.
 */
if (initial && sched_feat(START_DEBIT))
    vruntime += sched_vslice(cfs_rq, se);

/* sleeps up to a single latency don't count. */
if (!initial) {
    unsigned long thresh = sysctl_sched_latency;

    /*
     * Halve their sleep time's effect, to allow
     * for a gentler effect of sleepers:
     */
    if (sched_feat(GENTLE_FAIR_SLEEPERS))
        thresh >>= 1;

    vruntime -= thresh;
}

/* ensure we never gain time by being placed backwards. */
se->vruntime = max_vruntime(se->vruntime, vruntime);
```

- RQ enqueue
  - Wake up
    - If the process has been sleeping before, the virtual run time of the process is first adjusted in place_entity
    - by subtracting sysctl_sched_latency, it is ensured that the newly awoken process will only run after the current latency period has been finished

CFS Operations

```c
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);
    }

    if (static_branch_unlikely(&sched_numa_balancing))
        task_tick_numa(rq, curr);
}
```

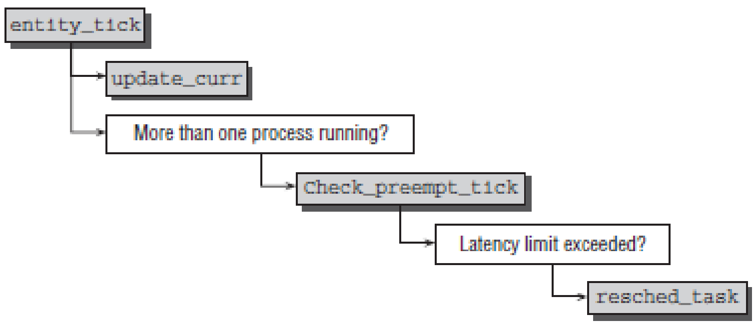


Figure 2-22: Code flow diagram for **entity_tick**.

```c
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));
```

```c
if (delta_exec < sysctl_sched_min_granularity)
    return;
```

```c
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));
```

- Tick
  - If the task has been running for longer than the desired time interval, a reschedule is requested with resched_task

CFS Operations

```c
    find_matching_se(&se, &pse);
    update_curr(cfs_rq_of(se));
    BUG_ON(!pse);
    if (wakeup_preempt_entity(se, pse) == 1) {
        /*
         * Bias pick_next to pick the sched entity that is
         * triggering this preemption.
         */
        if (!next_buddy_marked)
            set_next_buddy(pse);
        goto preempt;
    }

    return;

preempt:
    resched_curr(rq);
```

```c
static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
    s64 gran, vdiff = curr->vruntime - se->vruntime;

    if (vdiff <= 0)
        return -1;

    gran = wakeup_gran(curr, se);
    if (vdiff > gran)
        return 1;

    return 0;
}
```

```c
static unsigned long
wakeup_gran(struct sched_entity *curr, struct sched_entity *se)
{
    unsigned long gran = sysctl_sched_wakeup_granularity;

    /*
     * Since its curr running now, convert the gran from real-time
     * to virtual-time in his units.
     *
     * By using 'se' instead of 'curr' we penalize light tasks, so
     * they get preempted easier. That is, if 'se' < 'curr' then
     * the resulting gran will be larger, therefore penalizing the
     * lighter, if otoh 'se' > 'curr' then the resulting gran will
     * be smaller, again penalizing the lighter task.
     *
     * This is especially important for buddies when the leftmost
     * task is higher priority than the buddy.
     */
    return calc_delta_fair(gran, se);
}
```

- Preemption
  - the kernel uses check_preempt_curr to see if the new task can preempt the currently running one
  - If the new task is a real-time task, rescheduling is immediately requested because real-time tasks always preempt CFS tasks
  - The minimum is kept in sysctl_sched_wakeup_granularity

CFS Operations

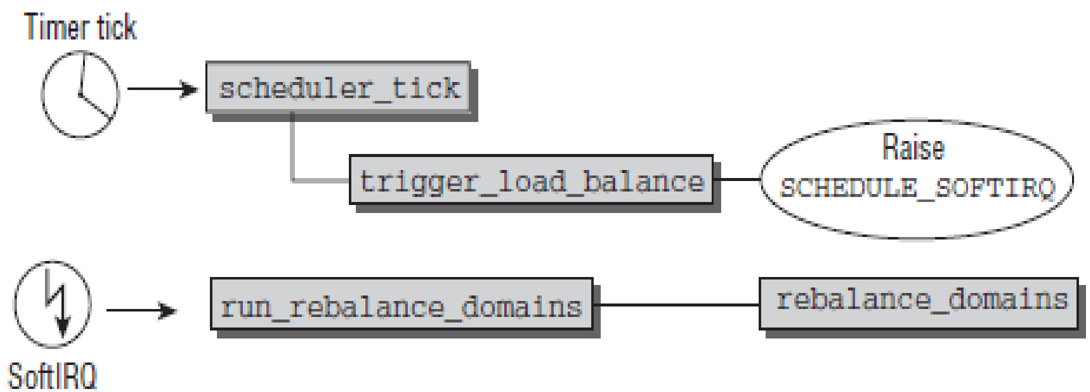
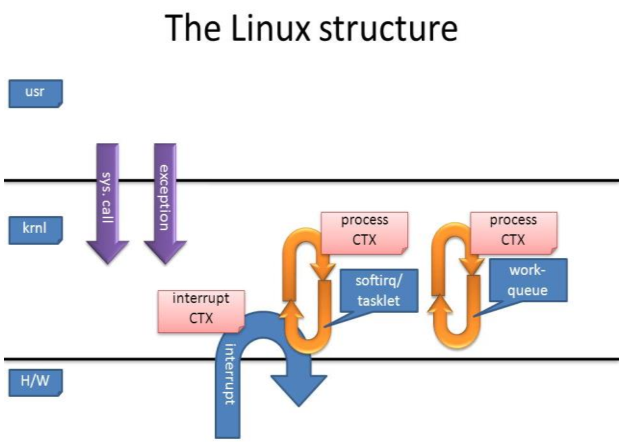


Figure 2-25: Time flow for initiation of load balancing on SMP systems.



- SMP
  - CPU affinity
  - Load Balancing
    - the scheduler_tick periodic scheduler function invokes the trigger_load_balance function on completion of the tasks required for all systems as described above
    - all run queues are organized in *scheduling domains*.

CFS Operations

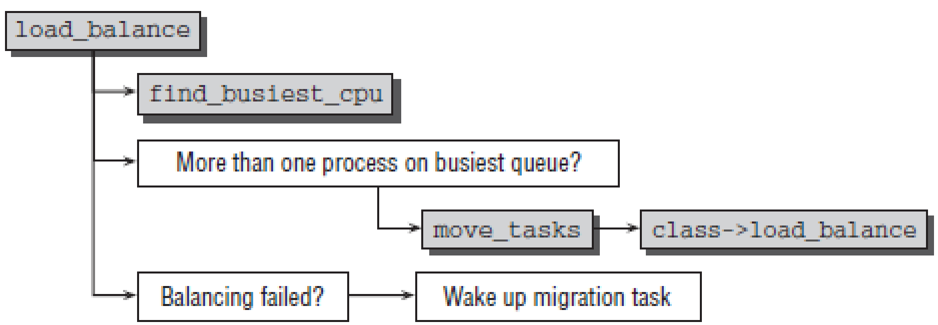


Figure 2-26: Code flow diagram for `load_balance`.

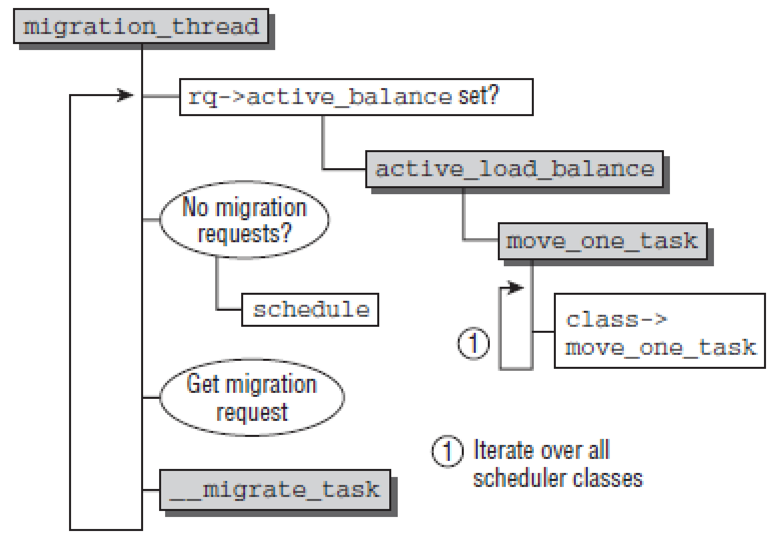
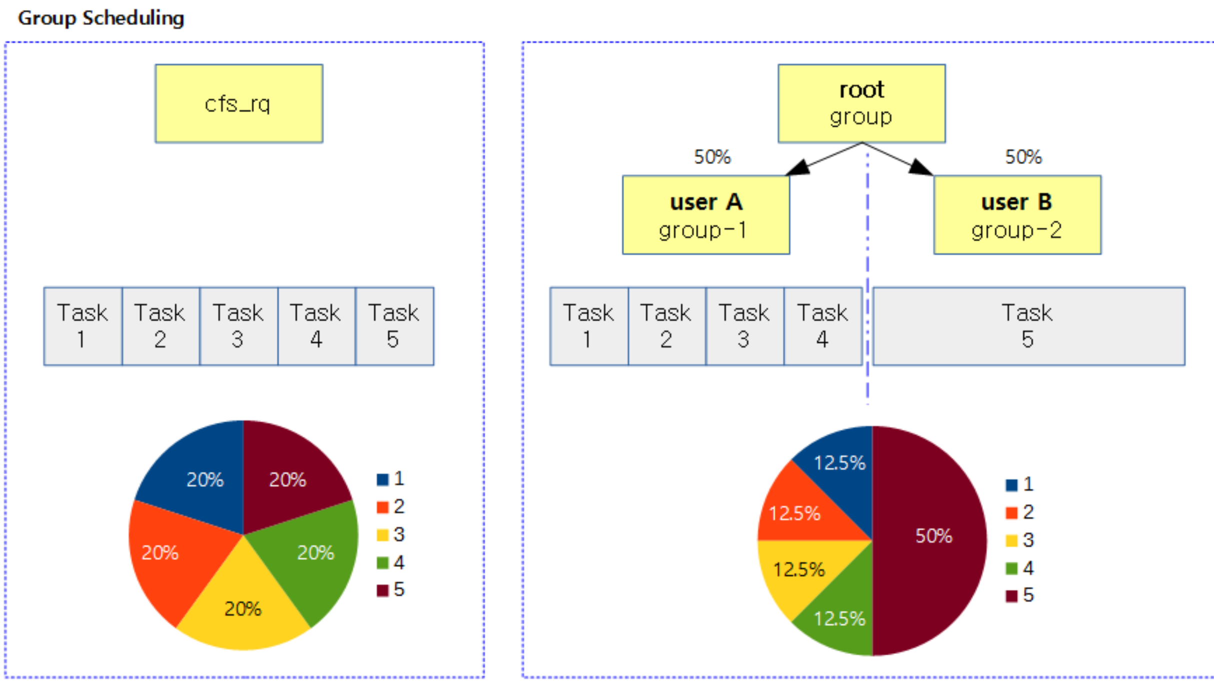


Figure 2-27: Code flow diagram for `migration_thread`.

- SMP
  - move_tasks function, in turn, invokes the scheduler-class specific load_balance method
  - Migration thread
    - This is handled in a kernel thread that executes migration_thread
    - The function tries to move one task from the current run queue to the run queue of the CPU that initiated the request for active balancing

CFS Operations




```
#ifdef CONFIG_FAIR_GROUP_SCHED
    int                 depth;
    struct sched_entity     *parent;
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq           *cfs_rq;
    /* rq "owned" by this entity/group: */
    struct cfs_rq           *my_q;
#endif
```

- Group Scheduling
  - Granting identical shares of the available CPU time to each user
  - Multiple hierarchies

CFS Operations

```
static __always_inline int preempt_count(void)
{
    return READ_ONCE(current_thread_info()->preempt_count);
}
```

```
void print_vma_addr(char *prefix, unsigned long ip)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;

    /*
     * Do not print if we are in atomic
     * contexts (in exception stacks, etc.):
     */
    if (preempt_count())
        return;
```

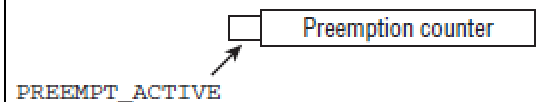


Figure 2-30: The per-process preemption counter.

- Kernel preemption
  - This allows not only userspace applications but also the kernel to be interrupted
  - If preempt_count is zero, the kernel can be interrupted, otherwise not
    - PREEMPT_ACTIVE

Device driver

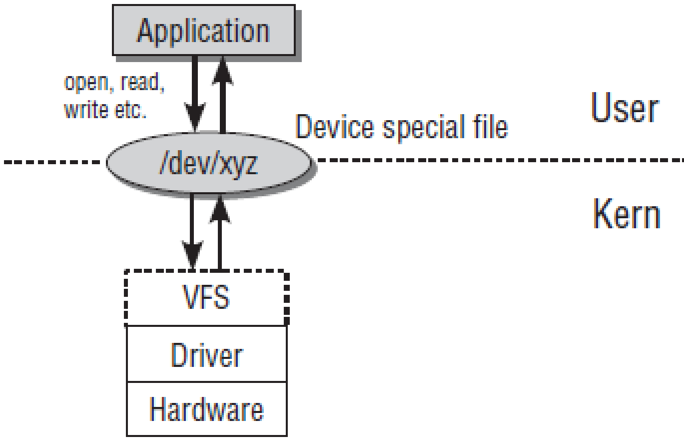
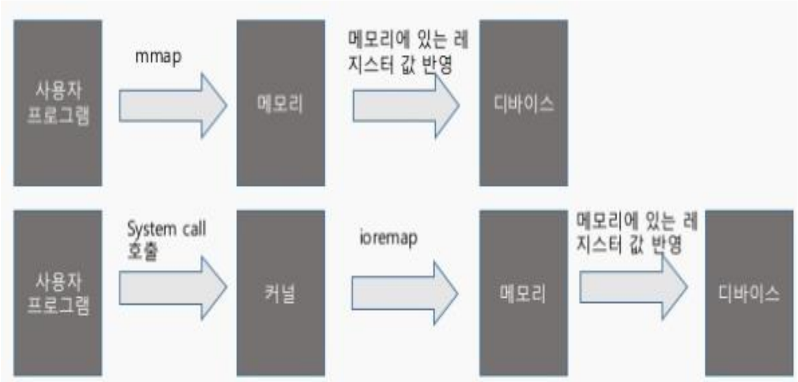


Figure 6-1: Layer model for addressing peripherals.



- Device access
  - Access to each individual device is performed via abstraction layers arranged hierarchically
  - Communication
    - I/O port – outb, outw commands
    - I/O Memory Mapping – ioremap, iounmap functions
    - Polling, Interrupts

Device driver

```
brw-rw----  1 root disk    8,   0 1월  18 06:01 sda
brw-rw----  1 root disk    8,   1 1월  18 06:02 sda1
brw-rw----  1 root disk    8,   2 1월  18 06:02 sda2
brw-rw----  1 root disk    8,   5 1월  18 06:02 sda5
brw-rw----  1 root disk    8,  16 1월  18 06:01 sdb
brw-rw----  1 root disk    8,  17 1월  18 06:02 sdb1
```

```
crw-rw-rw-  1 root tty     5,   0 1월  18 06:01 tty
crw--w----  1 root tty     4,   0 1월  18 06:01 tty0
crw--w----  1 root tty     4,   1 1월  18 06:02 tty1
crw--w----  1 root tty     4,  10 1월  18 06:01 tty10
crw--w----  1 root tty     4,  11 1월  18 06:01 tty11
crw--w----  1 root tty     4,  12 1月  18 06:01 tty12
```

```
#define UNNAMED_MAJOR           0
#define MEM_MAJOR           1
#define RAMDISK_MAJOR           1
#define FLOPPY_MAJOR            2
#define PTY_MASTER_MAJOR        2
#define IDE0_MAJOR          3
#define HD_MAJOR            IDE0_MAJOR
#define PTY_SLAVE_MAJOR         3
#define TTY_MAJOR           4
#define TTYAUX_MAJOR            5
#define LP_MAJOR            6
#define VCS_MAJOR           7
#define LOOP_MAJOR          7
#define SCSI_DISK0_MAJOR        8
#define SCSI_TAPE_MAJOR         9
```

- Device Files
  - Character, Block, Other Devices
  - Major, Minor number
    - identify the matching driver. The reason why two numbers are used is because of the general structure of a device driver
    - same type that are managed by a single device driver
    - devices of the same category can be combined so that they can be inserted logically into the kernel's data structures

Q&A