

Scheduling in Linux – Part 3

Scheduler Related Code Walk Through

Disclaimer: Codes shown have in many cases some unimportant / uninteresting lines deleted in the middle (sometimes marked with ... and sometimes not). You are supposed to check the actual code from the sources, not just look at slides.

CFS Implementation in Linux

- Outline of what we will see
 - What are some of the basic data structures involved?
 - Initialization of the scheduling parameters
 - Updating the runtimes
 - Updating the loads
 - Basic flow of the main scheduler
 - What happens on a timer tick?
 - Sleep and wakeup
 - When is the scheduler called?

Some Basic Data Structures

Some scheduling info in task_struct

```
struct task_struct {  
    ...  
    int                prio;  
    int                static_prio;  
    int                normal_prio;  
    unsigned int       rt_priority;  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    struct sched_rt_entity rt;  
    struct sched_dl_entity dl;  
    ...  
}
```

- *static_prio* : the static priority of the process from the nice value
- *prio* : the actual priority of the process used by the scheduler
- *normal_prio* : the priority based on the static priority and the scheduling policy
- *rt_priority* : real time priority (a number between 0 and 99)
- *se, rt, dl* : different scheduling entity structures corresponding to *fair*, *rt*, and *deadline* class. The applicable structure is used depending on the scheduling class of the process

Computing the different priorities

- Suppose the *task_struct* is pointed to by *p*
- Compute *p->normal_prio* from *p->static_prio*
- Compute *p->prio* from *p->normal_prio* (via *effective_prio()*)

```
#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)
```

<https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched/prio.h#L22>

https://elixir.bootlin.com/linux/v5.10.188/A/ident/effective_prio


```
static inline int normal_prio(struct task_struct *p)
{
    return __normal_prio(p->policy, p->rt_priority, PRIO_TO_NICE(p->static_prio));
}
```

```
static inline int __normal_prio(int policy, int rt_prio, int nice)
{
    int prio;
    if (dl_policy(policy))
        prio = MAX_DL_PRIO - 1;
    else if (rt_policy(policy))
        prio = MAX_RT_PRIO - 1 - rt_prio;
    else
        prio = NICE_TO_PRIO(nice);
    return prio;
}
```

```
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    /*
     * If we are RT tasks or we were boosted to RT priority,
     * keep the priority unchanged. Otherwise, update priority
     * to the normal priority:
     */
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```

The `sched_entity` data structure

- <https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched.h#L459>
- Defines the entity being scheduled in CFS
- Each node of the RB tree is a *sched_entity* structure
- This is a `fair class specific` structure, there are separate structures (*sched_entity_rt* etc.) for other classes

<i>struct sched_entity {</i>	
<i>struct load_weight</i>	<i>load;</i>
<i>struct rb_node</i>	<i>run_node;</i>
...	
<i>unsigned int</i>	<i>on_rq;</i>
<i>u64</i>	<i>exec_start;</i>
<i>u64</i>	<i>sum_exec_runtime;</i>
<i>u64</i>	<i>vruntime;</i>
...	
<i>u64</i>	<i>nr_migrations;</i>
<i>struct sched_statistics</i>	<i>statistics;</i>
...	

- *load* : the load of this process (the weight we used in CFS)
- *run_node* : the RB tree node for this process
- *on_rq* : task is on runqueue
- *exec_start* : starting time of the process in the last scheduling tick period
- *sum_exec_runtime* : total runtime of the process
- *vruntime* : virtual runtime
- *nr_migration* : number of times this process is migrated between CPUs
- *statistics* : a structure containing different scheduling stats field

The sched_class data structure

- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/sched.h#L1783>
- Defines generic functions (function pointers) for operations on the runqueue

```
struct sched_class {  
  
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);  
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);  
    void (*yield_task)   (struct rq *rq);  
    ...  
    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);  
    struct task_struct *(*pick_next_task)(struct rq *rq);  
    void (*task_fork)(struct task_struct *p);  
    void (*task_dead)(struct task_struct *p);  
    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);  
    ...  
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio);  
    void (*update_curr)(struct rq *rq);  
    ...  
}
```

- *enqueue_task* : called when a task becomes runnable
- *dequeue_task* : called when a task is no longer runnable
- *yield_task* : called when a task wants to give up the CPU voluntarily (but is still runnable)
- *check_preempt_curr* : checks if a runnable task should preempt the currently running task or not
- *pick_next_task* : choose the next task to run
- *task_fork, task_dead* : called to inform the scheduler that a new task is spawned or dead
- *task_tick* : called on a timer interrupt
- *prio_changed*: called when the priority of a process is changed
- *update_curr* : updates the runtime statistics

- Pointed to from the *task_struct* structure, assigned the correct scheduler class variable on initialization based on scheduling class
- The scheduler class variable has the actual functions
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/fair.c#L11529>

```
const struct sched_class fair_sched_class
    __section(" fair_sched_class") = {
    .enqueue_task          = enqueue_task_fair,
    .dequeue_task          = dequeue_task_fair,
    .yield_task            = yield_task_fair,
    .check_preempt_curr    = check_preempt_wakeup,
    .pick_next_task        = __pick_next_task_fair,
    .task_tick             = task_tick_fair,
    .task_fork             = task_fork_fair,
    .prio_changed          = prio_changed_fair
    .update_curr           = update_curr_fair
    ...
}
```

- The scheduler classes are themselves organized in an array by a linker script
- The order is very important, used in code in many places to ascertain priority
- <https://elixir.bootlin.com/linux/v5.10.188/source/include/asm-generic/vmlinux.lds.h#L128>

```
#define SCHED_DATA \
    STRUCT_ALIGN(); \
    __begin_sched_classes = .; \
    *(__idle_sched_class) \
    *(__fair_sched_class) \
    *(__rt_sched_class) \
    *(__dl_sched_class) \
    *(__stop_sched_class) \
    __end_sched_classes = .;
```

The runqueue

- Each CPU has its own runqueue
- The runqueue is a generic structure, has pointers to class-specific runqueues
 - <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/sched.h#L897>

```
struct rq {  
    raw_spinlock_t    lock;  
    unsigned int      nr_running;  
    ...  
    struct cfs_rq      cfs;  
    struct rt_rq       rt;  
    struct dl_rq       dl;  
    ...  
    struct task_struct __rcu *curr;  
    struct task_struct *idle;  
    int                cpu;  
    ...  
}
```

- *lock* : spinlock for locking the runqueue
- *nr_running*: number of processes on this queue, over all scheduling classes
- *cfs, rt, dl* : class specific queues for fair class, rt class, and deadline class
- *curr* : pointer to currently running process
- *idle* : pointer to the idle process
- *cpu* : cpu of this runqueue

Some of the fields in the CFS runqueue

```
struct cfs_rq {  
    struct load_weight    load;  
    unsigned int          nr_running;  
    unsigned int          h_nr_running;  
    ...  
    u64                  min_vruntime;  
    struct sched_entity  *curr;  
    ...  
}
```

- *load* : the load of all the processes in the runqueue
- *nr_running* : no. of processes in runqueue that will share the CPU
- *h_nr_running* : no. of processes in the runqueue
- *min_vruntime* : minimum vruntime in the queue
- *curr* : current running process

Initializations of scheduling parameters

Initializations

- *kernel_clone()* (in `kernel/fork.c`) calls *copy_process()* (in `kernel/fork.c`), which calls *sched_fork()* (in `kernel/sched/core.c`) that initializes most scheduling parameters
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L3244>

```
static void  __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    p->on_rq                = 0;
    p->se.on_rq              = 0;
    p->se.exec_start         = 0;
    p->se.sum_exec_runtime   = 0;
    p->se.prev_sum_exec_runtime = 0;
    p->se.nr_migrations      = 0;
    p->se.vruntime           = 0;
    ...
}
```

```

int sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    __sched_fork(clone_flags, p);
    ...
    p->prio=current->normal_prio;
    ...
    if (unlikely(p->sched_reset_on_fork)) {
        if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
            p->policy=SCHED_NORMAL; p->static_prio=NICE_TO_PRIO(0); p->rt_priority=0;
        } else if (PRIO_TO_NICE(p->static_prio) < 0) p->static_prio=NICE_TO_PRIO(0);
        p->prio=p->normal_prio=p->static_prio;
        set_load_weight(p);
        p->sched_reset_on_fork=0;
    }
    if (dl_prio(p->prio)) return -EAGAIN;
    else if (rt_prio(p->prio)) p->sched_class= &rt_sched_class;
    else p->sched_class=&fair_sched_class;
    ...

```

Updating runtimes

Updating the runtime

- Done by the *update_curr_fair()* function, which calls the *update_curr()* function
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/fair.c#L852>
- Called periodically on scheduler tick or on sleep/wakeup

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

```
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)  
{  
    if (unlikely(se->load.weight != NICE_0_LOAD))  
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);  
    return delta;  
}
```

```
static void update_min_vruntime(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    struct rb_node *leftmost = rb_first_cached(&cfs_rq->tasks_timeline);
    u64 vruntime = cfs_rq->min_vruntime;
    if (curr) {
        if (curr->on_rq)
            vruntime = curr->vruntime;
        else
            curr = NULL;
    }
}
```



```
if (leftmost) { /* non-empty tree */
    struct sched_entity *se;
    se = rb_entry(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);
```

Updating the load

Updating the load

- Needs to be done when
 - The priority changes
 - Done by the *set_load_weight()* function
 - *set_load_weight()* calls *reweight_task()* which computes the task's new weight and calls *reweight_entity()*
 - *reweight_entity()* assigns the task's weight to its *sched_entity* structure and updates the request queue's total load
 - Note that here both the task's and the runqueue's load changes
 - When a task is added or deleted from the queue
 - Only the runqueue's load changes (total load of all tasks in it)

On changing priority

```
static void set_load_weight(struct task_struct *p)
{
    bool update_load = !(READ_ONCE(p->state) & TASK_NEW);
    int prio = p->static_prio - MAX_RT_PRIO;
    struct load_weight *load = &p->se.load;

    ...
    /* SCHED_OTHER tasks have to update their load when changing their weight */
    if (update_load && p->sched_class == &fair_sched_class) {
        reweight_task(p, prio);
    } else {
        load->weight = scale_load(sched_prio_to_weight[prio]);
        ...
    }
}
```

```
static void reweight_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, unsigned long weight)
{
    if (se->on_rq) {
        if (cfs_rq->curr == se) update_curr(cfs_rq);
        update_load_sub(&cfs_rq->load, se->load.weight);
    }
    update_load_set(&se->load, weight);
    if (se->on_rq) update_load_add(&cfs_rq->load, se->load.weight);
}
```

On adding/deleting tasks

- *enqueue_task_fair()* calls *enqueue_entity()*
- *enqueue_entity()* calls *account_entity_enqueue()*
- *account_entity_enqueue()* calls *update_load_add()* to actually add the weight to the runqueue's load

Scheduler Flow

Basic scheduler flow

- Entry point is the generic `_schedule()` function
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L4430>

- General Flow

- Disable interrupts (*local_irq_disable()*)
- Lock the runqueue (*rq_lock()*)
- If current task is not in TASK_RUNNING state
 - If it has a signal pending (*signal_pending_state()*), change state to TASK_RUNNING
 - Else dequeue it
- Choose the next task to run (*pick_next_task()*) and context switch if needed (if different from current task)
- Unlock the run queue

Picking the next task

- Done by the *pick_next_task()* routine
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L4351>
- Simply goes through all scheduler classes in order to pick the highest priority task available
- Makes some interesting optimizations based on the fact that most often all tasks belong to the fair scheduling class
- Calls the scheduler class specific routine that actually picks the next task

```
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;
```

```
    if (likely(prev->sched_class <= &fair_sched_class &&
```

```
                rq->nr_running == rq->cfs.h_nr_running)) {
```

```
        p = pick_next_task_fair(rq, prev, rf);
```

```
        if (unlikely(p == RETRY_TASK))
```

```
            goto restart;
```

```
        if (!p) {
```

```
            put_prev_task(rq, prev);
```

```
            p = pick_next_task_idle(rq);
```

```
        }
```

```
        return p;
```

```
}
```

...

restart:

```
    put_prev_task_balance(rq, prev, rf);  
    for_each_class(class) {  
        p = class->pick_next_task(rq);  
        if (p)  
            return p;
```

- The CFS specific function *pick_next_task_fair()* actually picks the CFS task
- <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/fair.c#L7255>

```

struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

again:
    if (!sched_fair_runnable(rq))
        goto idle;

    ...
    if (prev)
        put_prev_task(rq, prev);
    do {
        se = pick_next_entity(cfs_rq, NULL);
        set_next_entity(cfs_rq, se);
        ...
    } while (cfs_rq);

    p = task_of(se);

```

What happens on timer tick

Time management

- The timer periodically interrupts
 - called a *scheduler tick*
- The timer interrupt handler calls *update_process_times()*
- *update_process_times()* calls *scheduler_tick()*
- *scheduler_tick()* calls the current process's *task_tick()* function, which for fair class, is *task_tick_fair()*
- *task_tick_fair()* calls *entity_tick()*

```
static void entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
    update_curr(cfs_rq);

    ...

    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);

    ...
}
```

- Already seen *update_curr()*
 - Updates *vruntime* and *min_vruntime*
- *check_preempt_tick()*
 - Checks if preemption is needed
 - Basically, compute the timeslice the current process should get
 - If the process has already run for longer than this, reschedule it
 - Otherwise, if the process should not run as per its new *vruntime*, reschedule it

```
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));
        ...
        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));
```

```
}
```

- *resched_curr()* actually does not preempt the current process (does not call the scheduler), it just sets a flag (`TIF_NEED_RESCHED`) indicating the task needs to be rescheduled
- *sched_slice()* calculates the timeslice

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

```
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;
}
```

Sleep and Wakeup

Sleep and Wakeup

- Processes wait on different wait queues
- Wait queues
 - Linked list of wait queue entries

Basic data structures for wait queues

```
struct wait_queue_entry {  
    unsigned int          flags;  
    void                  *private;  
    wait_queue_func_t    func;  
    struct list_head     entry;  
};
```

```
struct wait_queue_head {  
    spinlock_t           lock;  
    struct list_head     head;  
};
```

- *flags*: different values, the two of interest to us are
 - WQ_FLAG_EXCLUSIVE
 - WQ_FLAG_BOOKMARK
- *private*: points to the task that is waiting
- *func*: the function to be called on wake up
 - There is a *default_wake_function()* also, which through other functions, calls *activate_task()*, which enqueues the task back into the run queue

Creating a wait queue

```
#define __WAITQUEUE_INITIALIZER(name, tsk) {  
    .private = tsk,  
    .func      = default_wake_function,  
    .entry      = { NULL, NULL } }
```

```
#define DECLARE_WAITQUEUE(name, tsk)  
    struct wait_queue_entry name = __WAITQUEUE_INITIALIZER(name, tsk)
```

Initializing a wait queue and an entry

Head can be initialized by the *init_waitqueue_head()* function

Initializing an entry:

```
static inline void init_waitqueue_entry(struct wait_queue_entry *wq_entry, struct  
task_struct *p)  
{  
    wq_entry->flags                = 0;  
    wq_entry->private              = p;  
    wq_entry->func                 = default_wake_function;  
}
```

Adding/Deleting from wait queues

```
void add_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry  
*wq_entry)  
{  
    unsigned long flags;  
    wq_entry->flags &= ~WQ_FLAG_EXCLUSIVE;  
    spin_lock_irqsave(&wq_head->lock, flags);  
    __add_wait_queue(wq_head, wq_entry);  
    spin_unlock_irqrestore(&wq_head->lock, flags);  
}
```

```
void add_wait_queue_exclusive(struct wait_queue_head *wq_head, struct  
wait_queue_entry *wq_entry)
```

```
{
```

```
    unsigned long flags;
```

```
    wq_entry->flags |= WQ_FLAG_EXCLUSIVE;
```

```
    spin_lock_irqsave(&wq_head->lock, flags);
```

```
    __add_wait_queue_entry_tail(wq_head, wq_entry);
```

```
    spin_unlock_irqrestore(&wq_head->lock, flags);
```

```
}
```

```
void remove_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry  
*wq_entry);
```

- Entries with `WQ_FLAG_EXCLUSIVE` are always added to the end
- Thus the queue will always have a sequence of non-exclusive entries followed by a sequence of exclusive entries
- Matters in the way we wake up processes

- Basic steps
 - Declare a wait queue
 - On trying to sleep
 - Create a wait queue entry and initialize it properly
 - Add to wait queue and set state properly (*prepare_to_wait()*)
 - Call scheduler
 - After being woken, set task state to runnable again and remove entry from wait queue (*finish_wait()*)

void

*prepare_to_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry, int state)*

{

unsigned long flags;

...

spin_lock_irqsave(&wq_head->lock, flags);

if (list_empty(&wq_entry->entry))

__add_wait_queue(wq_head, wq_entry);

set_current_state(state);

spin_unlock_irqrestore(&wq_head->lock, flags);

}

```
void finish_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry)
{
    unsigned long flags;
    __set_current_state(TASK_RUNNING);
    if (!list_empty_careful(&wq_entry->entry)) {
        spin_lock_irqsave(&wq_head->lock, flags);
        list_del_init(&wq_entry->entry);
        spin_unlock_irqrestore(&wq_head->lock, flags);
    }
}
```

An example: helper function for socket buffer allocation

```
static long sock_wait_for_wmem(struct sock *sk, long timeo)
{
    DEFINE_WAIT(wait); ...
    for (;;) {
        if (!timeo) break;
        if (signal_pending(current)) break;
        ...
        prepare_to_wait(sk_sleep(sk), &wait, TASK_INTERRUPTIBLE);
        ...
        if (sk->sk_shutdown & SEND_SHUTDOWN) break;
        if (sk->sk_err) break;
        timeo = schedule_timeout(timeo);
    }
    finish_wait(sk_sleep(sk), &wait);
    return timeo;
}
```

Waking Up Tasks

- Main entry point: *wake_up()*
- Calls finally *__wake_up_common_lock()* which calls *__wake_up_common()*
- *__wake_up_common()*
 - Walks the queue to wake up the waiting processes, subject to a maximum
 - The exact number depends on wait flags and a count
 - For each process woken, calls the corresponding function in its *wait_queue_entry*
 - If a maximum number is reached and there are still entries to wake up, bookmark and return, *__wake_up_common_lock* will call it again immediately in a loop

```
static void __wake_up_common_lock(struct wait_queue_head *wq_head, unsigned int mode,
                                int nr_exclusive, int wake_flags, void *key)
{
    unsigned long flags; wait_queue_entry_t bookmark;
    bookmark.flags = 0; bookmark.private = NULL; bookmark.func = NULL;
    INIT_LIST_HEAD(&bookmark.entry);
    do {
        spin_lock_irqsave(&wq_head->lock, flags);
        nr_exclusive = __wake_up_common(wq_head, mode, nr_exclusive,
wake_flags, key, &bookmark);
        spin_unlock_irqrestore(&wq_head->lock, flags);
    } while (bookmark.flags & WQ_FLAG_BOOKMARK);
}
```

```
static int __wake_up_common(struct wait_queue_head *wq_head, unsigned int mode,
                           int nr_exclusive, int wake_flags, void *key,
                           wait_queue_entry_t *bookmark)
{
    wait_queue_entry_t *curr, *next;
    int cnt = 0;
    lockdep_assert_held(&wq_head->lock);
    if (bookmark && (bookmark->flags & WQ_FLAG_BOOKMARK)) {
        curr = list_next_entry(bookmark, entry);
        list_del(&bookmark->entry);
        bookmark->flags = 0;
    } else
        curr = list_first_entry(&wq_head->head, wait_queue_entry_t, entry);
    if (&curr->entry == &wq_head->head)
        return nr_exclusive;
}
```

```

list_for_each_entry_safe_from(curr, next, &wq_head->head, entry) {
    unsigned flags = curr->flags;    int ret;
    if (flags & WQ_FLAG_BOOKMARK) continue;
    ret = curr->func(curr, mode, wake_flags, key);
    if (ret < 0) break;
    if (ret && (flags & WQ_FLAG_EXCLUSIVE) &&!--nr_exclusive) break;

    if (bookmark && (++cnt > WAITQUEUE_WALK_BREAK_CNT) && (&next->entry !=
&wq_head->head)) {
        bookmark->flags = WQ_FLAG_BOOKMARK;
        list_add_tail(&bookmark->entry, &next->entry);
        break;
    }
}

return nr_exclusive;
}

```

Example: Releasing a socket

```
void release_sock(struct sock *sk)
{
    spin_lock_bh(&sk->sk_lock.slock);
    if (sk->sk_prot->release_cb)
        sk->sk_prot->release_cb(sk);
    sock_release_ownership(sk);
    if (waitqueue_active(&sk->sk_lock.wq))
        wake_up(&sk->sk_lock.wq);
    spin_unlock_bh(&sk->sk_lock.slock);
}
```


The idle process

- Function executed *do_idle()*
- Entry function at CPU startup (*cpu_startup_entry*)

```
void cpu_startup_entry(enum cpuhp_state state)  
{  
    arch_cpu_idle_prepare();  
    cpuhp_online_idle(state);  
    while (1)  
        do_idle();  
}
```

```
static void do_idle(void)  
{  
    ...  
    while (!need_resched()) {  
        ...  
    }  
    ...  
    schedule_idle();  
    ...
```

Where all is the scheduler called from

When is the scheduler called?

- On process exit
 - Called by the function *do_task_dead()* (called by *do_exit()*) when a process exits
 - <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L4565>
- For scheduling the idle process
 - Called by *schedule_idle()* (from *do_idle()*) for scheduling the idle task
 - <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L4654>
- Called from wait and wake up functions
 - Too many to list, from too many drivers, file systems, other places

- On process preemption, from *preempt_schedule()* and related functions
 - <https://elixir.bootlin.com/linux/v5.10.188/source/kernel/sched/core.c#L4723>
 - Checks the need-to-reschedule flag set earlier after a kernel task finishes

Scheduling Real Time Tasks

Scheduling Real Time Tasks

- Two policies
 - SCHED_FIFO
 - SCHED_RR
- *sched_fork()* initializes class based on priority value
- Runqueue are a set of priority arrays, 1 per real time priority
- Bitmap of size MAX_RT_PRIO, one per real time priority

```
struct rt_prio_array {  
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /*  
    include 1 bit for delimiter */  
    struct list_head queue[MAX_RT_PRIO];  
};
```

```
struct rt_prio_array {  
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for  
    delimiter */  
    struct list_head queue[MAX_RT_PRIO];  
};
```


- Scheduling is very similar to what we saw earlier
 - *pick_next_task_rt()* eventually calls *pick_next_task_rt_entity()* which finds the first non-empty priority queue (from highest to lowest priority) using the bitmap, then chooses the first task in it to run
 - SCHED_FIFO: no timeslice, a task runs till completion
 - SCHED_RR: fixed timeslice, round-robin within the same priority level

```
static struct sched_rt_entity *pick_next_rt_entity(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);
    queue = array->queue + idx;
    if (SCHED_WARN_ON(list_empty(queue)))
        return NULL;
    next = list_entry(queue->next, struct sched_rt_entity, run_list);
    return next;
}
```

- On periodic scheduler tick
 - *scheduler_tick()* calls *task_tick()*, which calls *task_tick_rt()*
- *task_tick_rt()* action
 - If SCHED_FIFO, no effect, as they have no timeslice
 - If SCHED_RR and timeslice is not over, return
 - If SCHED_RR and timeslice over, recompute timeslice and add to end of queue, call *resched_curr()* to schedule next task
 - Timeslices are same for all priorities, given by RR_TIMESLICE

```
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);
    ...
    if (p->policy != SCHED_RR) return;
    if (--p->rt.time_slice) return;
    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) {
            enqueue_task_rt(rq, p, 0);
            resched_curr(rq);
            return;
        }
    }
}
```