Linux Kernel Development 03

工海系 大四 朱紹勳

	Linux Kernel Dev 3rd	Understand Linux Kernel 3rd	Source Code Example
Process Management	Chapter 3	Chapter 3	
Process Scheduling	Chapter 4	Chapter 7	
Kernel Synchronization	Chapter 9, 10	Chapter 5	

Enter & Exit of System Call

int0x80

- Entering
  - set_system_gate(0x80, &system_call);
  - system_call()

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xffffe000, %ebx
  /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx

  cmpl $NR_syscalls, %eax
  jb nobadsys
  movl $(-ENOSYS), 24(%esp)
  jmp resume_userspace
nobadsys:

call *sys_call_table(0, %eax, 4)
```

- Exiting
  - system_call()

```
movl %eax, 24(%esp)
cli
movl 8(%ebp), %ecx
testw $0xffff, %cx
je restore_all
```

sysenter, sysexit

- Use 3 special registers:

```
cs  <- SYSENTER_CS_MSR
eip <- SYSENTER_EIP_MSR
esp <- SYSENTER_ESP_MSR
ss  <- SYSENTER_CS_MSR + 8
```

- vsyscall:
  - CPU support

```
__kernel_vsyscall:
int $0x80
ret
```

  - CPU not support

```
__kernel_vsyscall:
pushl %ecx
pushl %edx
pushl %ebp
movl %esp, %ebp
sysenter
```

- Entering
  - eax <- #syscall
    __kernel_vsyscall
  - save registers to user stack
  - sysenter_entry:

```
movl -508(%esp), %esp
sti
pushl $(__USER_DS)
pushl %ebp
pushfl
pushl $(__USER_CS)
pushl $SYSENTER_RETURN
movl (%ebp), %ebp
```

- Exiting
  - SYSENTER_RETURN:

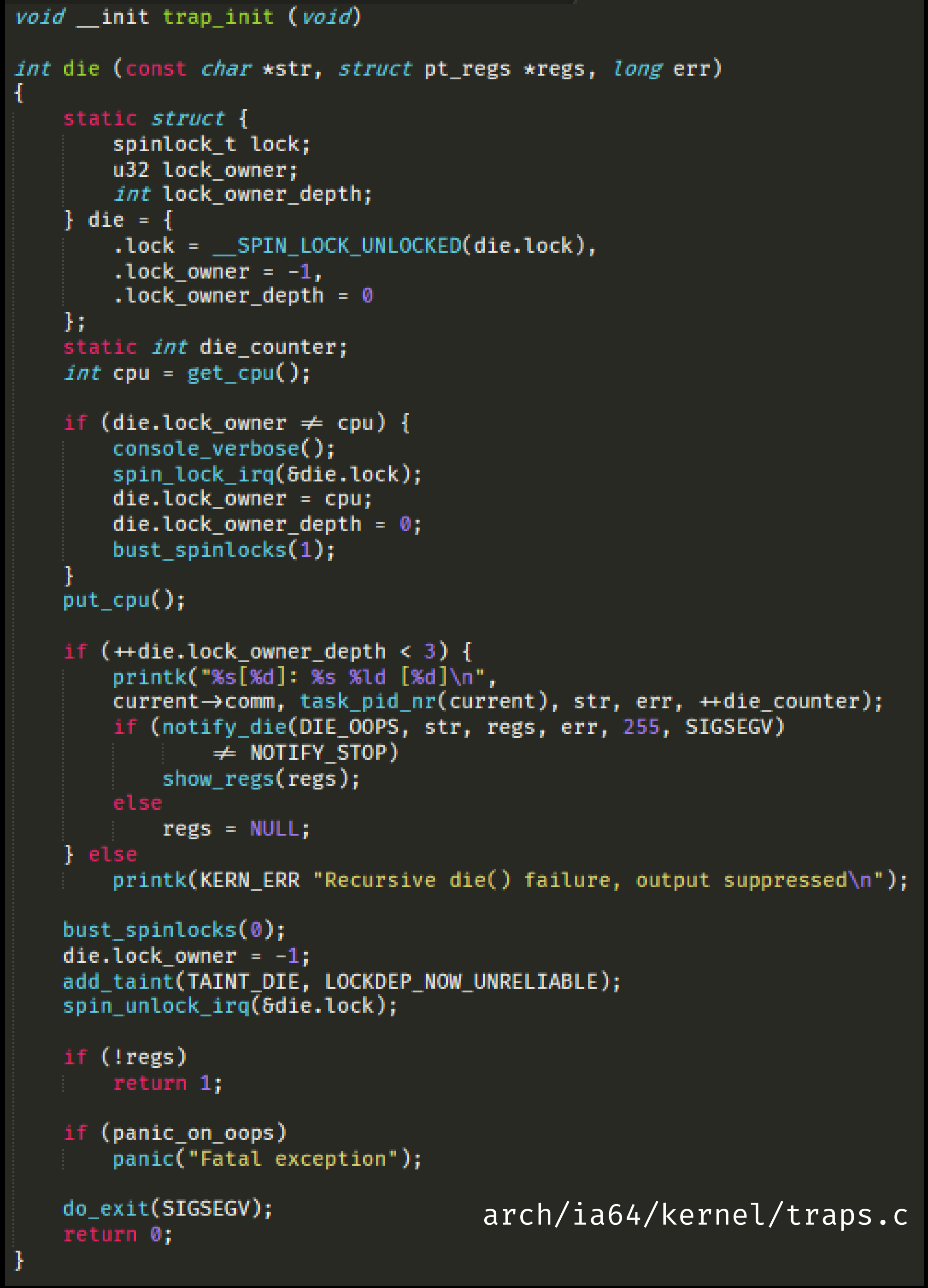
```
popl %ebp
popl %edx
popl %ecx
ret
```

Process Management

Process Termination

- do_group_exit()
  - terminate while thread group. Extension from do_exit()
- do_exit()

```
1. Set flag(proc desc.) to PF_EXITING
2. Call del_timer_sync()
3. Call exit_mm(), exit_sem(), exit_files(), exit_fs()…
4. exit_notify()
     1.  update family relationship(init)
     2.  SIGCHLD
     3.  if _exit_signal == 1 ? EXIT_DEAD : EXIT_ZOMBIE
5. schedule() // to switch to new process
6. release_task()
   delete proc desc. (before then, we can still access info of dead process)
     1.  __exit_signal()
     2.  wait() wait4(), waitpid()

Parentless Task: exit_notify(),
     forget_original_parent(), find_new_reaper(), ptrace_exit_finish()
```



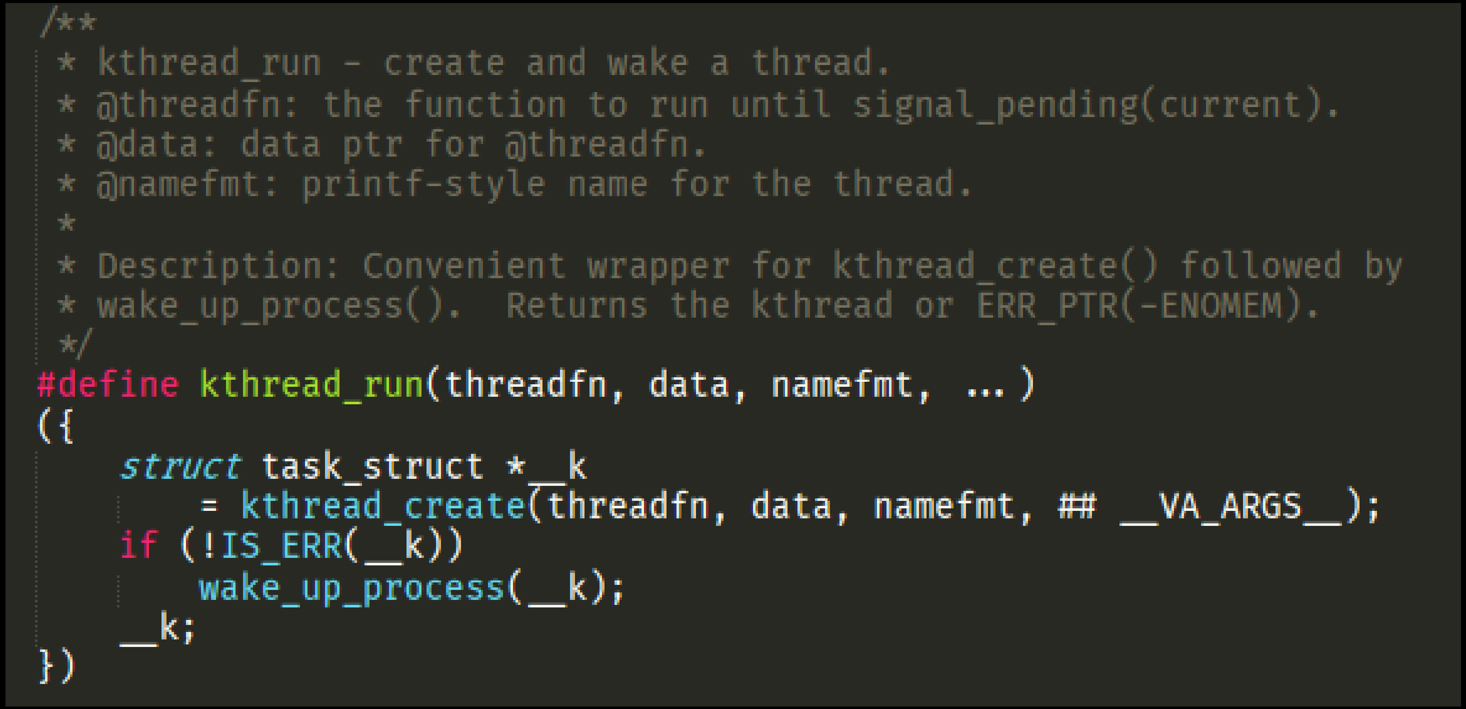
```c
void __init trap_init (void)

int die (const char *str, struct pt_regs *regs, long err)
{
    static struct {
        spinlock_t lock;
        u32 lock_owner;
        int lock_owner_depth;
    } die = {
        .lock = __SPIN_LOCK_UNLOCKED(die.lock),
        .lock_owner = -1,
        .lock_owner_depth = 0
    };
    static int die_counter;
    int cpu = get_cpu();

    if (die.lock_owner != cpu) {
        console_verbose();
        spin_lock_irq(&die.lock);
        die.lock_owner = cpu;
        die.lock_owner_depth = 0;
        bust_spinlocks(1);
    }
    put_cpu();

    if (++die.lock_owner_depth < 3) {
        printk("%s[%d]: %s %ld [%d]\n",
        current->comm, task_pid_nr(current), str, err, ++die_counter);
        if (notify_die(DIE_OOPS, str, regs, err, 255, SIGSEGV)
                != NOTIFY_STOP)
            show_regs(regs);
        else
            regs = NULL;
    } else
        printk(KERN_ERR "Recursive die() failure, output suppressed\n");

    bust_spinlocks(0);
    die.lock_owner = -1;
    add_taint(TAINT_DIE, LOCKDEP_NOW_UNRELIABLE);
    spin_unlock_irq(&die.lock);

    if (!regs)
        return 1;

    if (panic_on_oops)
        panic("Fatal exception");

    do_exit(SIGSEGV);
    return 0;
}
```

arch/ia64/kernel/traps.c

Thread

- Actually a light-weighted (specialized) process
- Thread Creation
  - `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0); // clone(SIGCHLD, 0);`
  - `flags are defined in <linux/sched.h>`

- Kernel Thread
  - Perform some operations in the background(in kernel)
  - declared in `<linux/kthread.h>`
  - `struct task_struct *kthread_create(int (*threadfn)(void *data), void *data, const char namefmt[], ...)`
  - `struct task_struct *kthread_run(int (*threadfn)(void *data), void *data, const char namefmt[], ...)`
  - ksoftirqd is an example



```c
/**
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
 * @namefmt: printf-style name for the thread.
 *
 * Description: Convenient wrapper for kthread_create() followed by
 * wake_up_process().  Returns the kthread or ERR_PTR(-ENOMEM).
 */
#define kthread_run(threadfn, data, namefmt, ...)
({
    struct task_struct *__k
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__);
    if (!IS_ERR(__k))
        wake_up_process(__k);
    __k;
})
```

Process Scheduling

Intro of Linux Process Scheduling

- preemptive & non-preemptive(yield)
- History of Linux Process Scheduling Algorithm:
  1. Simple (scan the process linked list and calculate the priority)
  2. O(1) algorithm: ideal for larger server workload
  3. Rotating Staircase Deadline Scheduler
     (Known as Completely Fair Scheduler, CFS)
- I/O-Bound Vs. Processor-Bound
- Interactive processes Vs. Batch processes Vs. Real-time processes
  - Performance bottleneck is limited by IO Request or Processor Computing
  - Trade off between two goals: low latency & high throughput

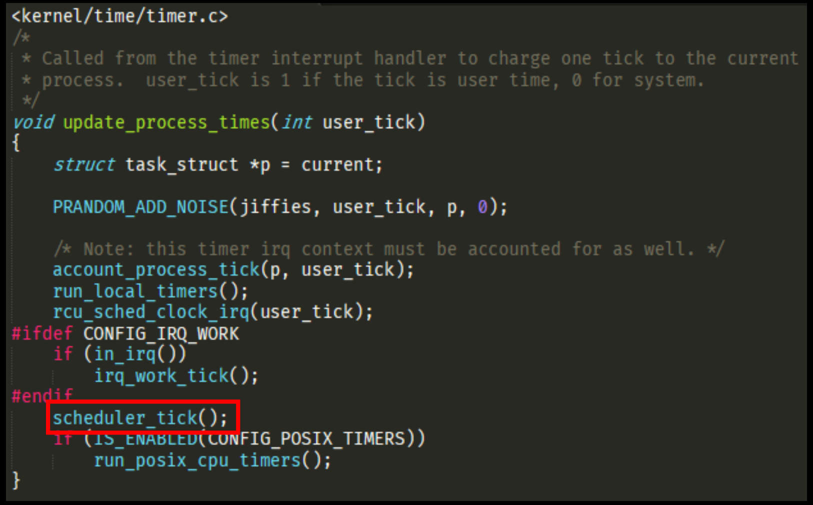
Process Priority & Time slice

- The approaches of priority determination:
  1. Real Time Priority(0~99) & Nice Value(100 ~ 139)
  2. Time Slice
     1. Short → waste time on process overhead, I/O-bound prefer
     2. Long → poor interactive performance, CPU-bound prefer

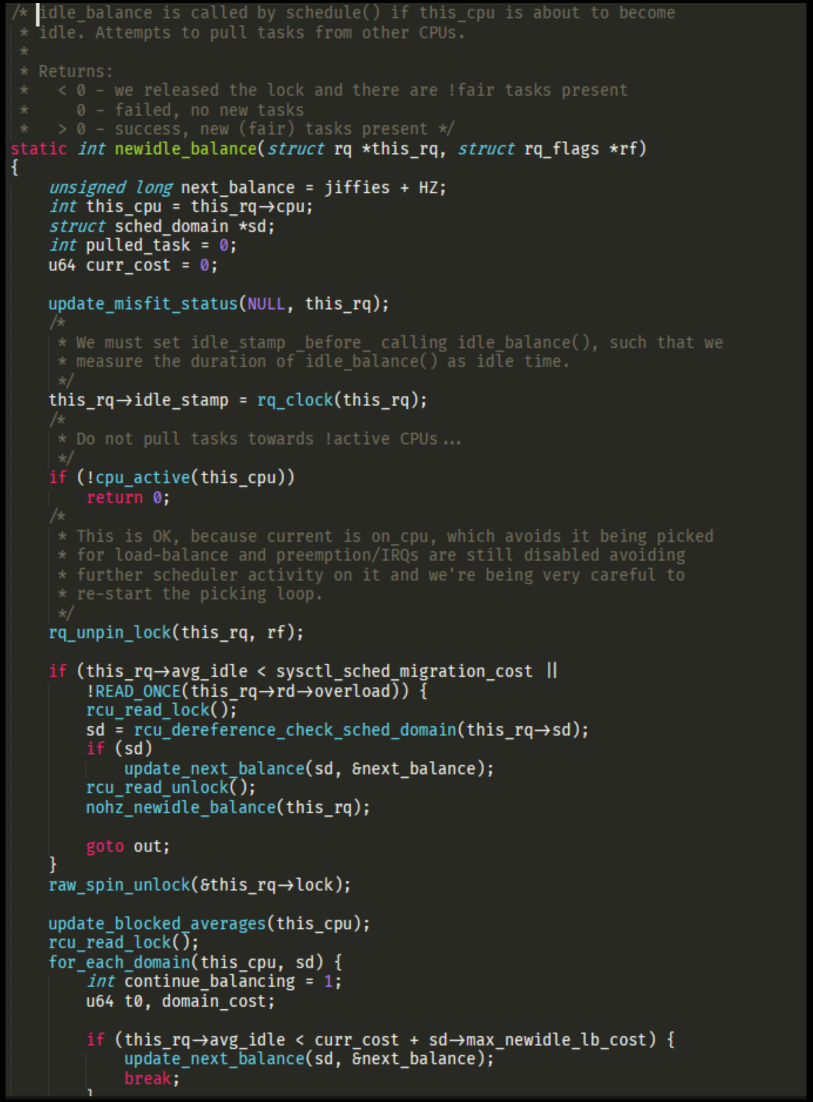
- Linux assigns processes a proportion of the processor, which related to the nice value.
- SCHED_FIFO, SCHED_RR(Round Robin), SCHED_NORMAL
- Use swapper (PID 0) to execute scheduling

Functions Used by the Scheduler

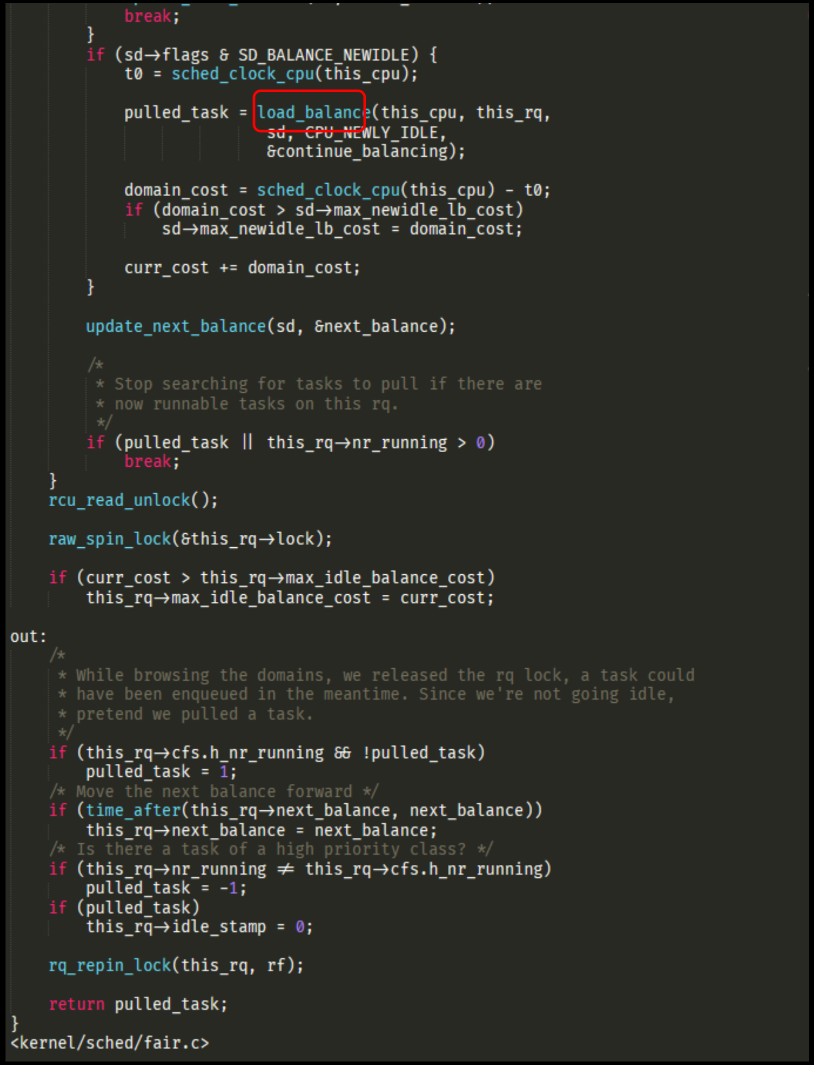
- `scheduler_tick()`
  - Keeps the time_slice counter of current up-to-date
- `try_to_wake_up(p, TASH_NORMAL, 0);`
  - Awakens a sleeping process
- `recalc_task_prio()`
  - Updates the dynamic priority of a process
- `schedule()`
  - Selects a new process to be executed
- `load_balance()`
  - Keeps the runqueues of a multiprocessor system balanced



```c
<kernel/time/timer.c>
/*
 * Called from the timer interrupt handler to charge one tick to the current
 * process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick)
{
    struct task_struct *p = current;

    PRANDOM_ADD_NOISE(jiffies, user_tick, p, 0);

    /* Note: this timer irq context must be accounted for as well. */
    account_process_tick(p, user_tick);
    run_local_timers();
    rcu_sched_clock_irq(user_tick);
#ifdef CONFIG_IRQ_WORK
    if (in_irq())
        irq_work_tick();
#endif
    scheduler_tick();
    if (IS_ENABLED(CONFIG_POSIX_TIMERS))
        run_posix_cpu_timers();
}
```



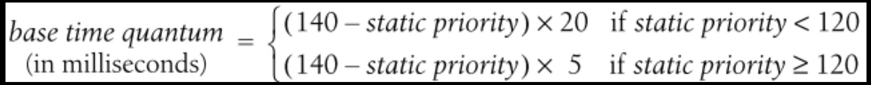
```c
/* idle_balance is called by schedule() if this_cpu is about to become
 * idle. Attempts to pull tasks from other CPUs.
 *
 * Returns:
 *   < 0 - we released the lock and there are !fair tasks present
 *     0 - failed, no new tasks
 *   > 0 - success, new (fair) tasks present */
static int newidle_balance(struct rq *this_rq, struct rq_flags *rf)
{
    unsigned long next_balance = jiffies + HZ;
    int this_cpu = this_rq->cpu;
    struct sched_domain *sd;
    int pulled_task = 0;
    u64 curr_cost = 0;

    update_misfit_status(NULL, this_rq);
    /*
     * We must set idle_stamp _before_ calling idle_balance(), such that we
     * measure the duration of idle_balance() as idle time.
     */
    this_rq->idle_stamp = rq_clock(this_rq);
    /*
     * Do not pull tasks towards !active CPUs...
     */
    if (!cpu_active(this_cpu))
        return 0;
    /*
     * This is OK, because current is on_cpu, which avoids it being picked
     * for load-balance and preemption/IRQs are still disabled avoiding
     * further scheduler activity on it and we're being very careful to
     * re-start the picking loop.
     */
    rq_unpin_lock(this_rq, rf);

    if (this_rq->avg_idle < sysctl_sched_migration_cost ||
        !READ_ONCE(this_rq->rd->overload)) {
        rcu_read_lock();
        sd = rcu_dereference_check_sched_domain(this_rq->sd);
        if (sd)
            update_next_balance(sd, &next_balance);
        rcu_read_unlock();
        nohz_newidle_balance(this_rq);

        goto out;
    }
    raw_spin_unlock(&this_rq->lock);

    update_blocked_averages(this_cpu);
    rcu_read_lock();
    for_each_domain(this_cpu, sd) {
        int continue_balancing = 1;
        u64 t0, domain_cost;

        if (this_rq->avg_idle < curr_cost + sd->max_newidle_lb_cost) {
            update_next_balance(sd, &next_balance);
            break;
        }
```



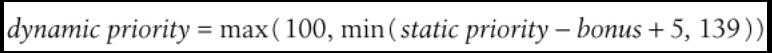
```c
            break;
        }
        if (sd->flags & SD_BALANCE_NEWIDLE) {
            t0 = sched_clock_cpu(this_cpu);

            pulled_task = load_balance(this_cpu, this_rq,
                                       sd, CPU_NEWLY_IDLE,
                                       &continue_balancing);

            domain_cost = sched_clock_cpu(this_cpu) - t0;
            if (domain_cost > sd->max_newidle_lb_cost)
                sd->max_newidle_lb_cost = domain_cost;

            curr_cost += domain_cost;
        }

        update_next_balance(sd, &next_balance);

        /*
         * Stop searching for tasks to pull if there are
         * now runnable tasks on this rq.
         */
        if (pulled_task || this_rq->nr_running > 0)
            break;
    }
    rcu_read_unlock();

    raw_spin_lock(&this_rq->lock);

    if (curr_cost > this_rq->max_idle_balance_cost)
        this_rq->max_idle_balance_cost = curr_cost;

out:
    /*
     * While browsing the domains, we released the rq lock, a task could
     * have been enqueued in the meantime. Since we're not going idle,
     * pretend we pulled a task.
     */
    if (this_rq->cfs.h_nr_running && !pulled_task)
        pulled_task = 1;
    /* Move the next balance forward */
    if (time_after(this_rq->next_balance, next_balance))
        this_rq->next_balance = next_balance;
    /* Is there a task of a high priority class? */
    if (this_rq->nr_running != this_rq->cfs.h_nr_running)
        pulled_task = -1;
    if (pulled_task)
        this_rq->idle_stamp = 0;

    rq_repin_lock(this_rq, rf);

    return pulled_task;
}
<kernel/sched/fair.c>
```

Normal Scheduling vs Real Time Scheduling

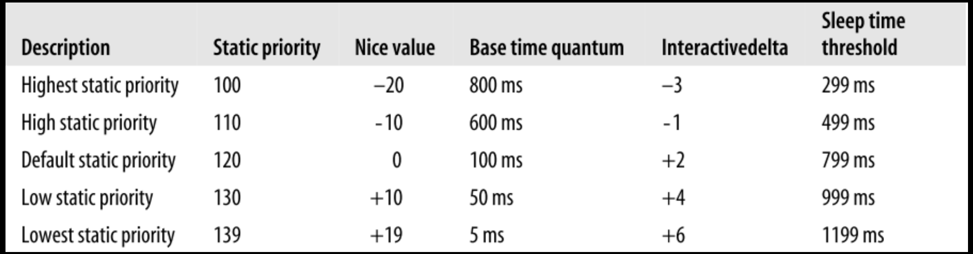
- Normal Scheduling
- Can use system call to change priority
  `nice() and setpriority()`



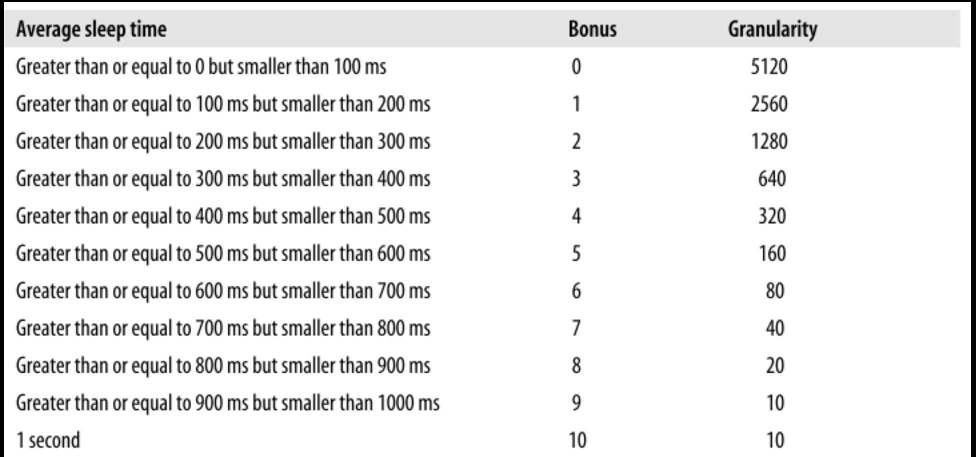
$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases}$$



$$\text{dynamic priority} = \max(100, \min(\text{static priority} - \text{bonus} + 5, 139))$$



Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	−20	800 ms	−3	299 ms
High static priority	110	-10	600 ms	-1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms



Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

- Real Time Scheduling
- SCHED_FIFO, SCHED_RR
  (SCHED_RR have time slice)
- Can use system call to change priority
  `sched_setparam() sched_setscheduler()`
- context switch in following situations
  - preempt by higher priority process
  - process block and sleep
  - process stop or be killed
  - yield
  - Finish time slice in SCHED_RR

Intro of Completely Fair Scheduler(CFS)

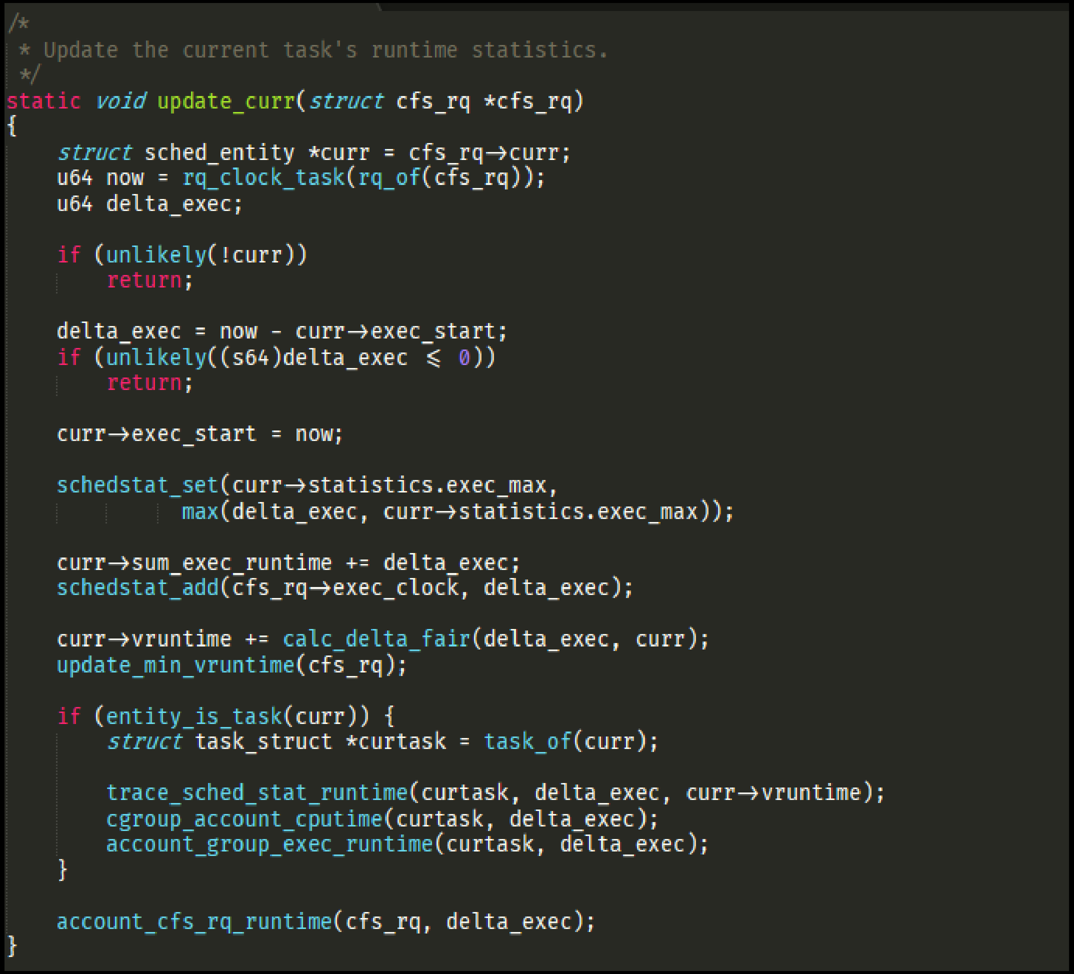
- defined in `<kernel/sched/fair.c>`
- Run each process
  - round-robin
  - selecting process that has run the least.
  - Calculates how long a process should run as a function of the total number of runnable processes.
  - nice value is only use for weighting the proportion of processor
- CFS sets a target for its approximation of the “infinitely small” scheduling duration in perfect multitasking.
- No time slice concept in CFS
- The minimum granularity : 1ms

Implement of CFS

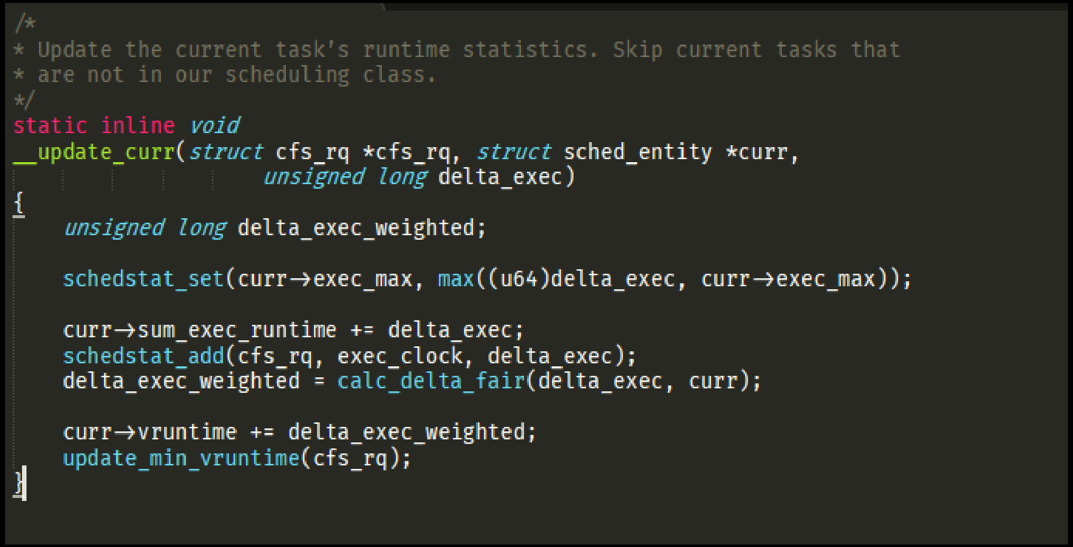
- Time Accounting
- Process Selection
- The Scheduler Entry Point
- Sleeping and Waking Up

Time Accounting

- Scheduler Entity Structure
  - defined in `<linux/sched.h>`, embedded in task_struct of process
- Virtual Runtime (ns)
  - = Actual runtime normalized (or weighted) by the # runnable process
  - To approximate the “ideal multitasking processor” in CFS



```c
/*
 * 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;
    if (unlikely((s64)delta_exec <= 0))
        return;

    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);
        cgroup_account_cputime(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```



```c
/*
* Update the current task's runtime statistics. Skip current tasks that
* are not in our scheduling class.
*/
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
          unsigned long delta_exec)
{
    unsigned long delta_exec_weighted;

    schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));

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

    curr->vruntime += delta_exec_weighted;
    update_min_vruntime(cfs_rq);
}
```

Process Selection

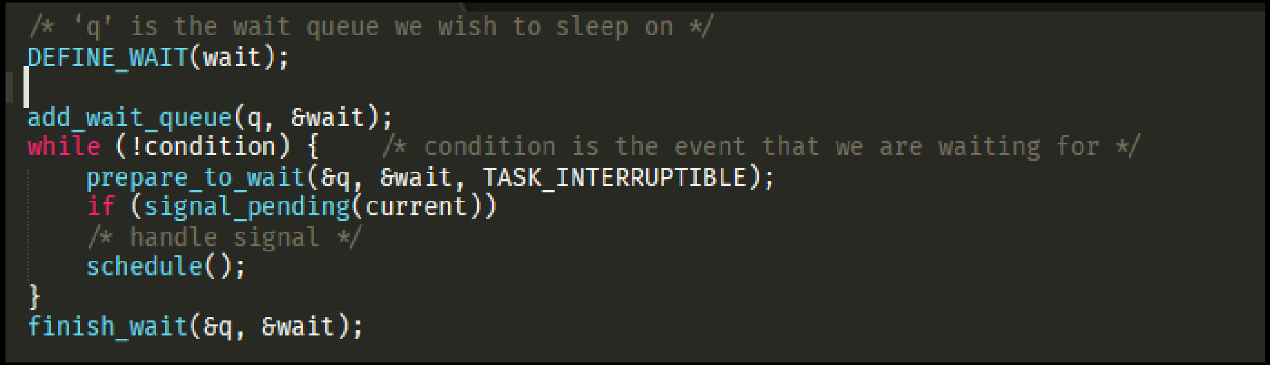
- Simply find the process with smallest vruntime
- `__pick_next_entity()`, defined in `<kernel/sched_fair.c>`
- We use RBtree, the key of each process are the vruntime
  - The far left node is the next process for run

The Scheduler Entry Point

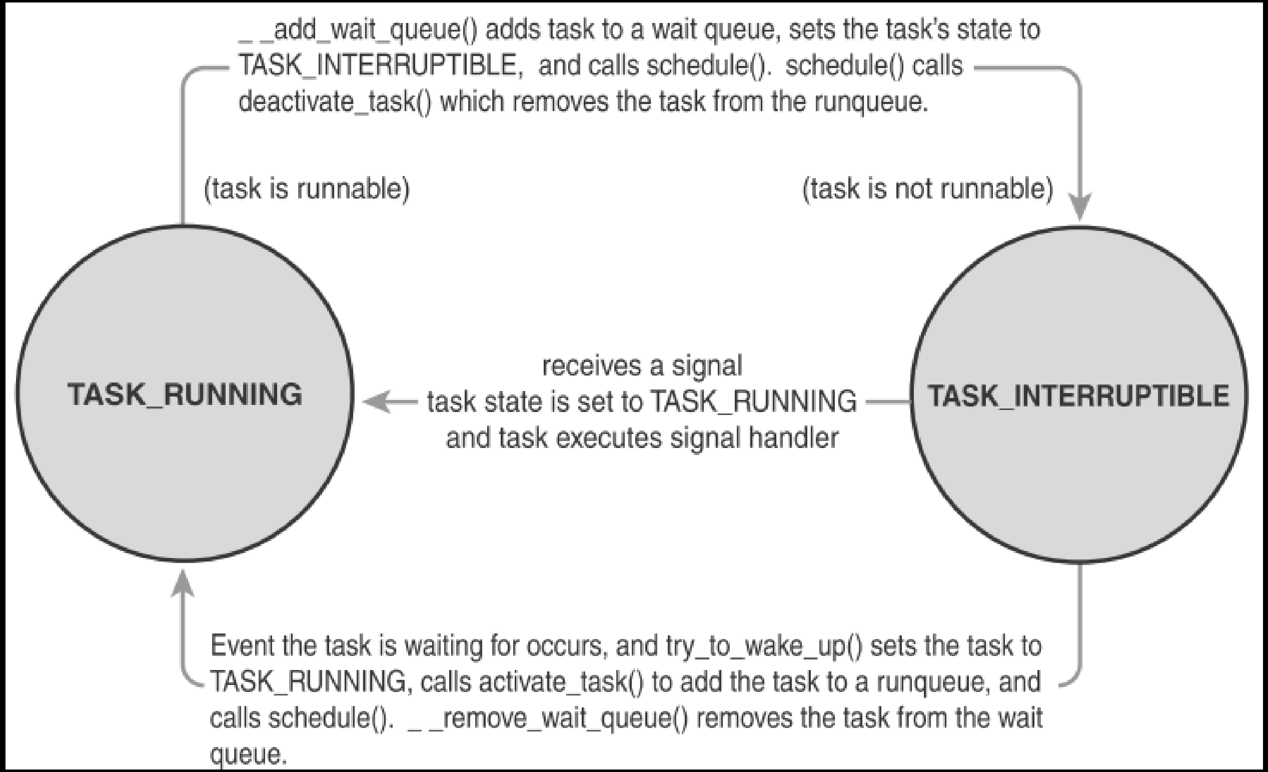
- `schedule()`, defined in `<kernel/sched.c>`
- The `schedule()` will call `pick_next_task()`

Sleeping and Waking Up

- State : TASK_INTERRUPTIBLE(wake up prematurely & response) or TASK_UNINTERRUPTIBLE()
- Waiting Queue
- Waking Up
  - use `wake_up()`, nested call `try_to_wake_up()`
  - `set TASK_RUNNING & use enqueuer_task to insert task to RBtree`



```c
/* '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);
```



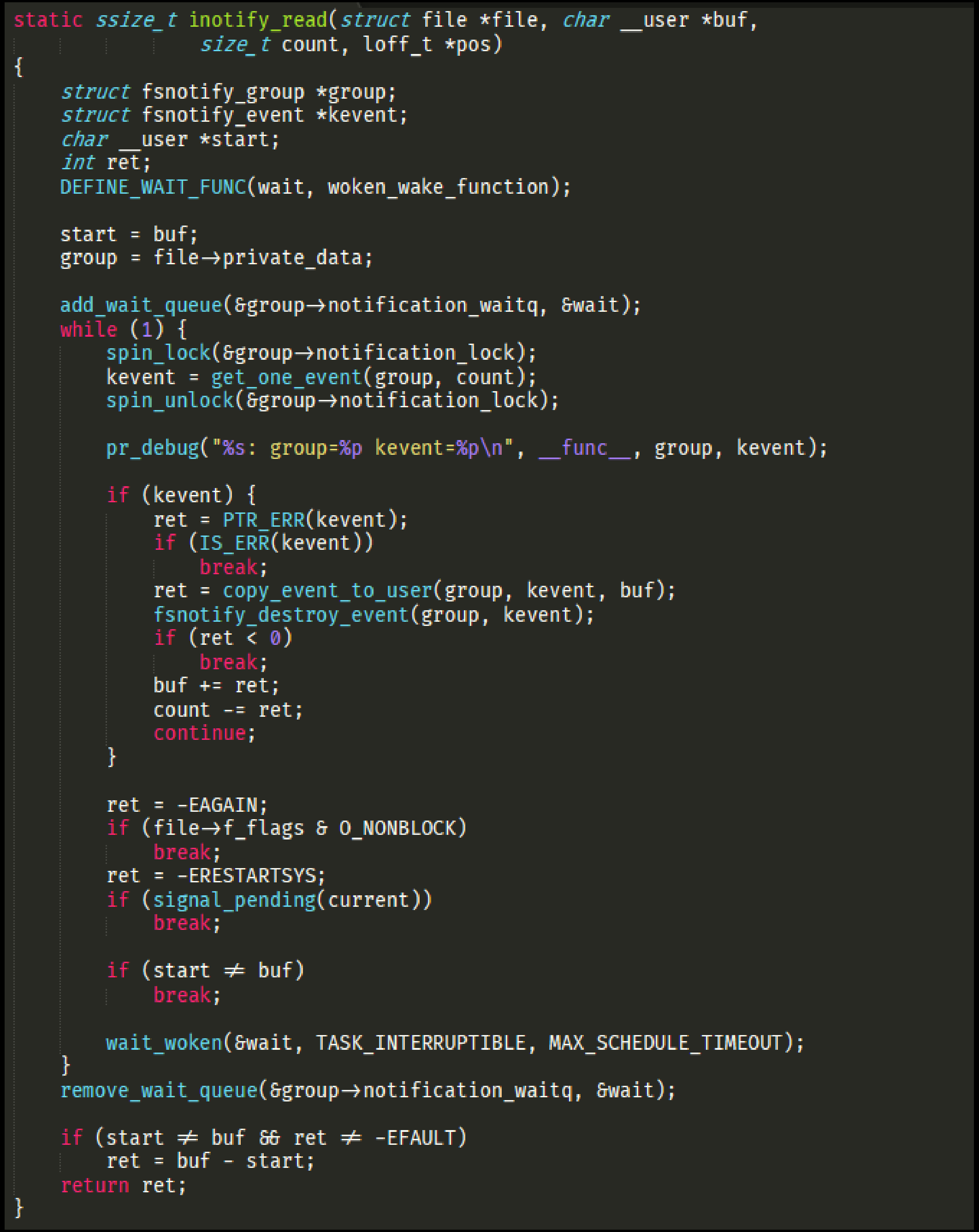
__add_wait_queue() adds task to a wait queue, sets the task's state to TASK_INTERRUPTIBLE, and calls schedule(). schedule() calls deactivate_task() which removes the task from the runqueue.

(task is runnable)

(task is not runnable)

receives a signal
task state is set to TASK_RUNNING
and task executes signal handler

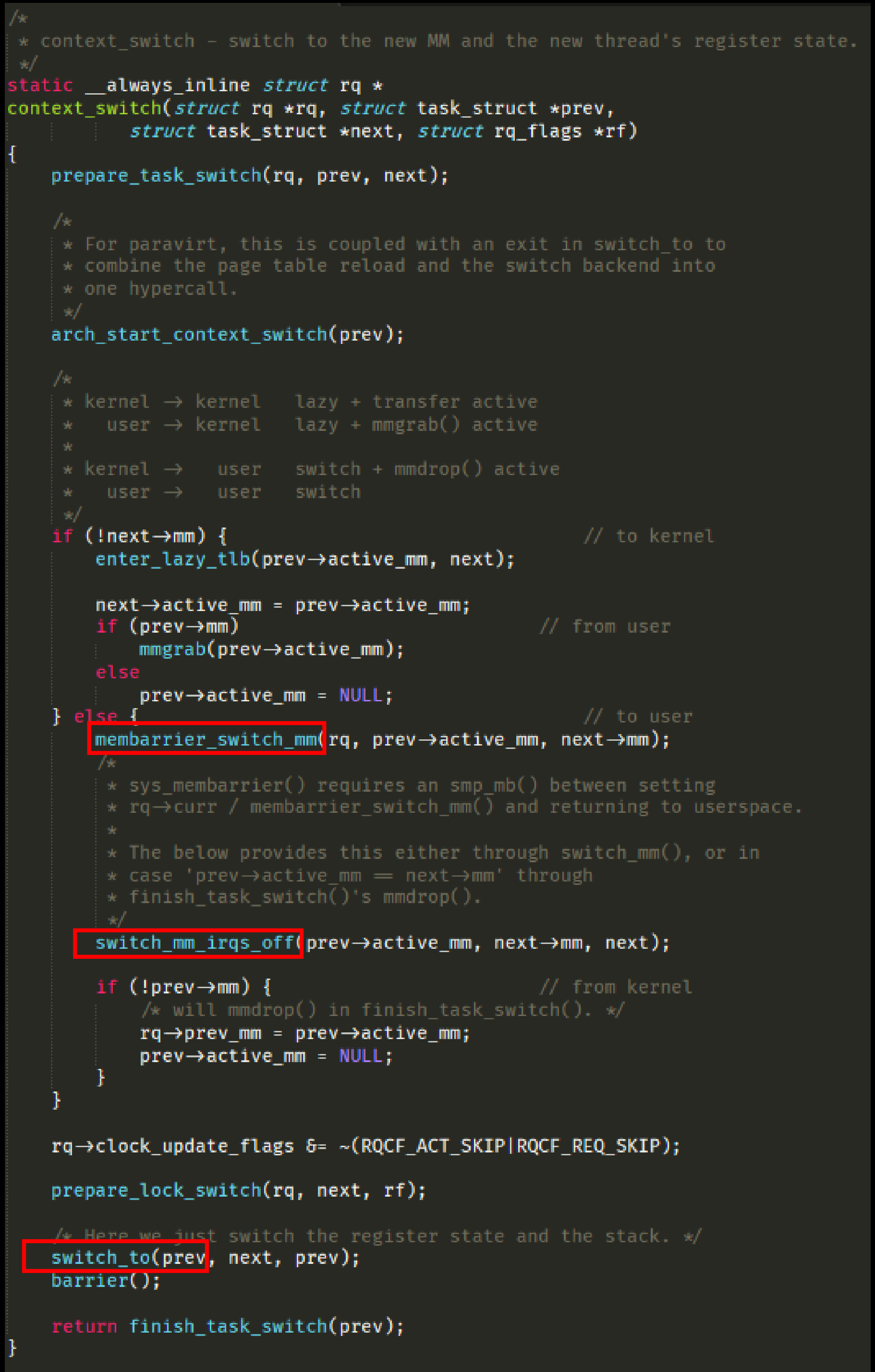
Event the task is waiting for occurs, and try_to_wake_up() sets the task to TASK_RUNNING, calls activate_task() to add the task to a runqueue, and calls schedule(). __remove_wait_queue() removes the task from the wait queue.



```c
static ssize_t inotify_read(struct file *file, char __user *buf,
                            size_t count, loff_t *pos)
{
    struct fsnotify_group *group;
    struct fsnotify_event *kevent;
    char __user *start;
    int ret;
    DEFINE_WAIT_FUNC(wait, woken_wake_function);

    start = buf;
    group = file->private_data;

    add_wait_queue(&group->notification_waitq, &wait);
    while (1) {
        spin_lock(&group->notification_lock);
        kevent = get_one_event(group, count);
        spin_unlock(&group->notification_lock);

        pr_debug("%s: group=%p kevent=%p\n", __func__, group, kevent);

        if (kevent) {
            ret = PTR_ERR(kevent);
            if (IS_ERR(kevent))
                break;
            ret = copy_event_to_user(group, kevent, buf);
            fsnotify_destroy_event(group, kevent);
            if (ret < 0)
                break;
            buf += ret;
            count -= ret;
            continue;
        }

        ret = -EAGAIN;
        if (file->f_flags & O_NONBLOCK)
            break;
        ret = -ERESTARTSYS;
        if (signal_pending(current))
            break;

        if (start != buf)
            break;

        wait_woken(&wait, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
    }
    remove_wait_queue(&group->notification_waitq, &wait);

    if (start != buf && ret != -EFAULT)
        ret = buf - start;
    return ret;
}
```

Preemption and Context Switching

- Schedule() → context_switch(), defined in `<kernel/sched.c>`
    - `switch_mm(), switch process in virtual memory`
    - `switch_to(), save & restore stack information and the processor registers`
- Use need_resched to check the needed of rescheduling
- User preemption
    - (1).Return from a system call (2).Return from an interrupt handler
    - check need_resched, if set ? find another process, entry.s
- Kernel preemption
    - (1) interrupt handler exits before returning to kermel (2)Kernel code becomes preemptible (3)kernel explicitly calls schedule() (4)kernel blocks
    - preempt_count, lock++, release—, != 0 means has holding lock



```c
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
	       struct task_struct *next, struct rq_flags *rf)
{
	prepare_task_switch(rq, prev, next);

	/*
	 * For paravirt, this is coupled with an exit in switch_to to
	 * combine the page table reload and the switch backend into
	 * one hypercall.
	 */
	arch_start_context_switch(prev);

	/*
	 * kernel -> kernel   lazy + transfer active
	 *   user -> kernel   lazy + mmgrab() active
	 *
	 * kernel ->   user   switch + mmdrop() active
	 *   user ->   user   switch
	 */
	if (!next->mm) {                                // to kernel
		enter_lazy_tlb(prev->active_mm, next);

		next->active_mm = prev->active_mm;
		if (prev->mm)                           // from user
			mmgrab(prev->active_mm);
		else
			prev->active_mm = NULL;
	} else {                                        // to user
		membarrier_switch_mm(rq, prev->active_mm, next->mm);
		/*
		 * sys_membarrier() requires an smp_mb() between setting
		 * rq->curr / membarrier_switch_mm() and returning to userspace.
		 *
		 * The below provides this either through switch_mm(), or in
		 * case 'prev->active_mm == next->mm' through
		 * finish_task_switch()'s mmdrop().
		 */
		switch_mm_irqs_off(prev->active_mm, next->mm, next);

		if (!prev->mm) {                        // from kernel
			/* will mmdrop() in finish_task_switch(). */
			rq->prev_mm = prev->active_mm;
			prev->active_mm = NULL;
		}
	}

	rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);

	prepare_lock_switch(rq, next, rf);

	/* Here we just switch the register state and the stack. */
	switch_to(prev, next, prev);
	barrier();

	return finish_task_switch(prev);
}
```

Scheduler-Related System Calls

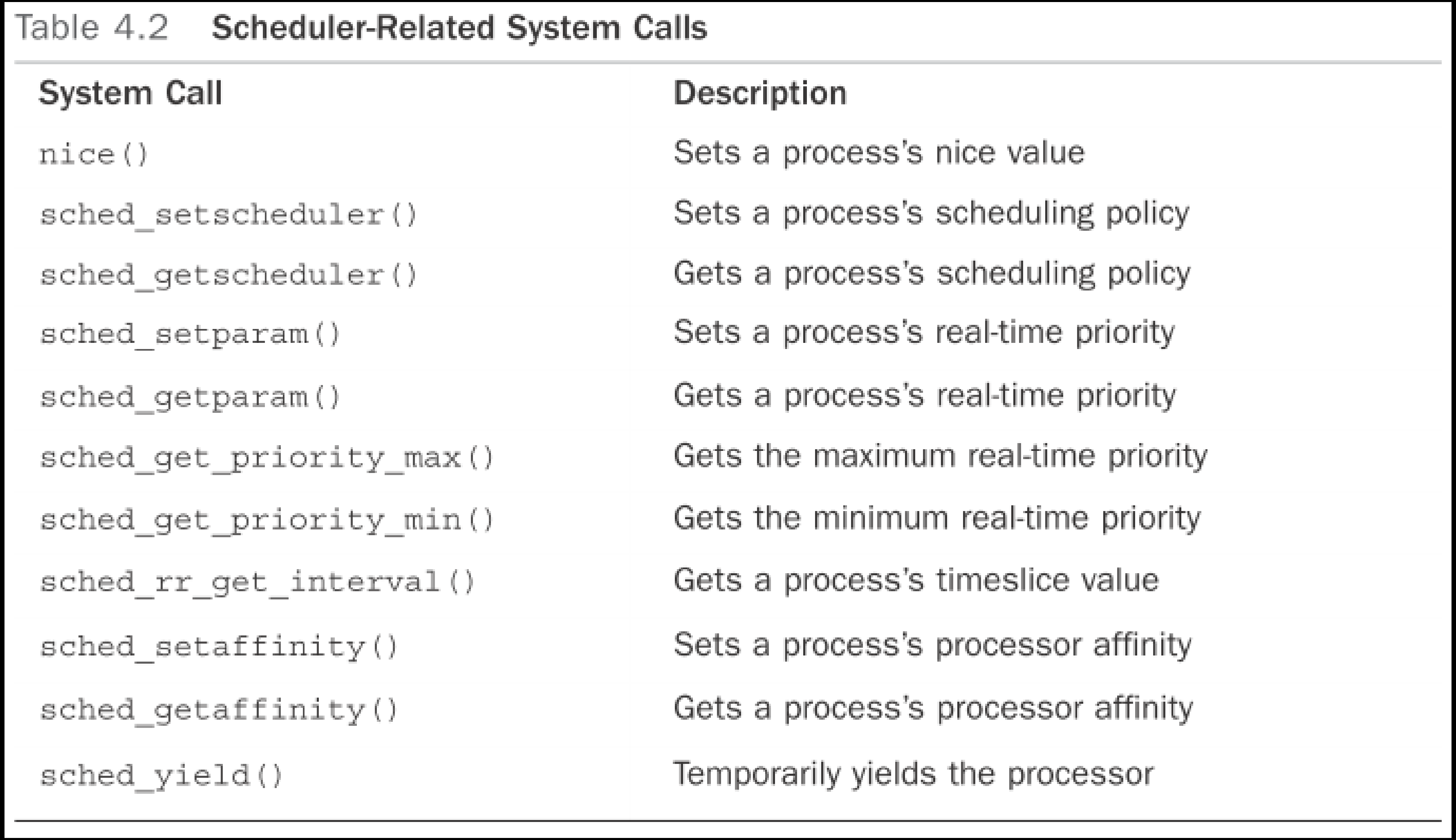


Table 4.2 **Scheduler-Related System Calls**

System Call	Description
`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

Kernel Synchronization

Intro of Synchronization

- Critical Section(CS) & Race Condition
- Lock, atomic operation
- Concurrency
  - Cause by:
    - interrupt
    - softirq & tasklet
    - Kernel preemption
    - Sleeping and synchronization with user-space
    - Symmetrical multiprocessing(different cpu run same process)
  - Solution:
    - interrupt-saft
    - SMP-safe
    - Preempt-safe

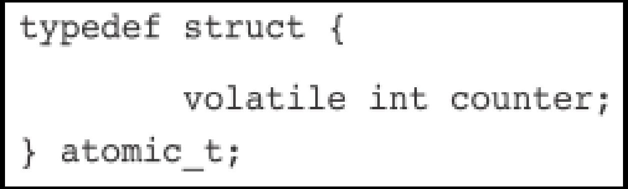
Intro of Synchronization

- Deadlock
  - some simple rules:
    - Implement lock ordering.
    - Prevent starvation.
    - Design for simplicity
- lock contention
  - High contented lock may lead to overhead
- scalability
  - The granularity of locking

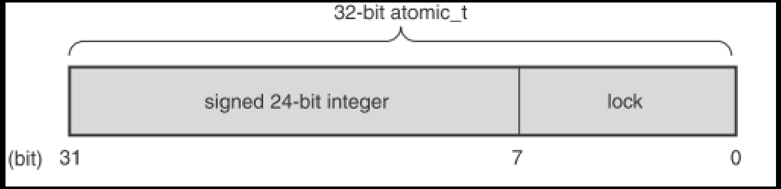
Solutions for Synchronization

- Atomic Operation
- Spinlock
- Reader-Writer Spinlock
- Semaphore
- Reader-Writer Semaphores
- Mutexes
- Completion Variables
- BKL: The Big Kernel Lock
- Sequential Locks
- Preemption Disable
- Ordering & Barriers

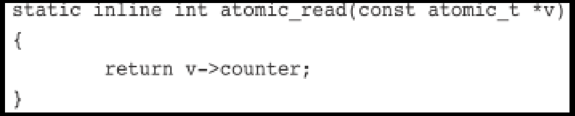
Atomic Operation



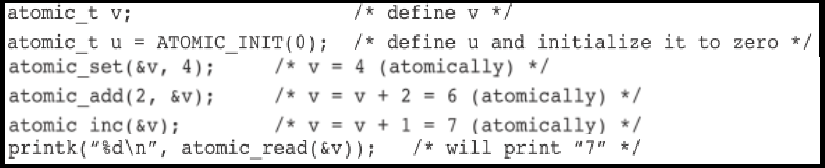
```
typedef struct {
        volatile int counter;
} atomic_t;
```



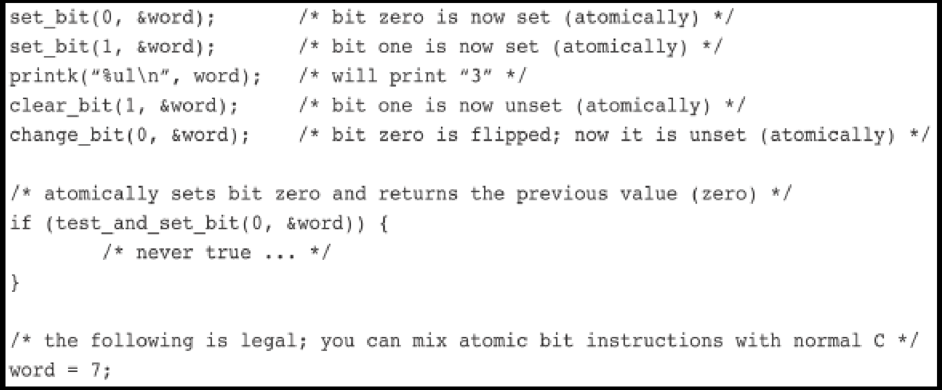
- 1. Operates on integers 2. Operates on individual bits
- atomic_t data type (instead of int), defined in `<linux/types.h>` :
  - Ensures the compiler does not optimize access to the value
  - in old Linux, atomic_t has only 24 bits for integer operation
  - atomic64_T
  - low cost



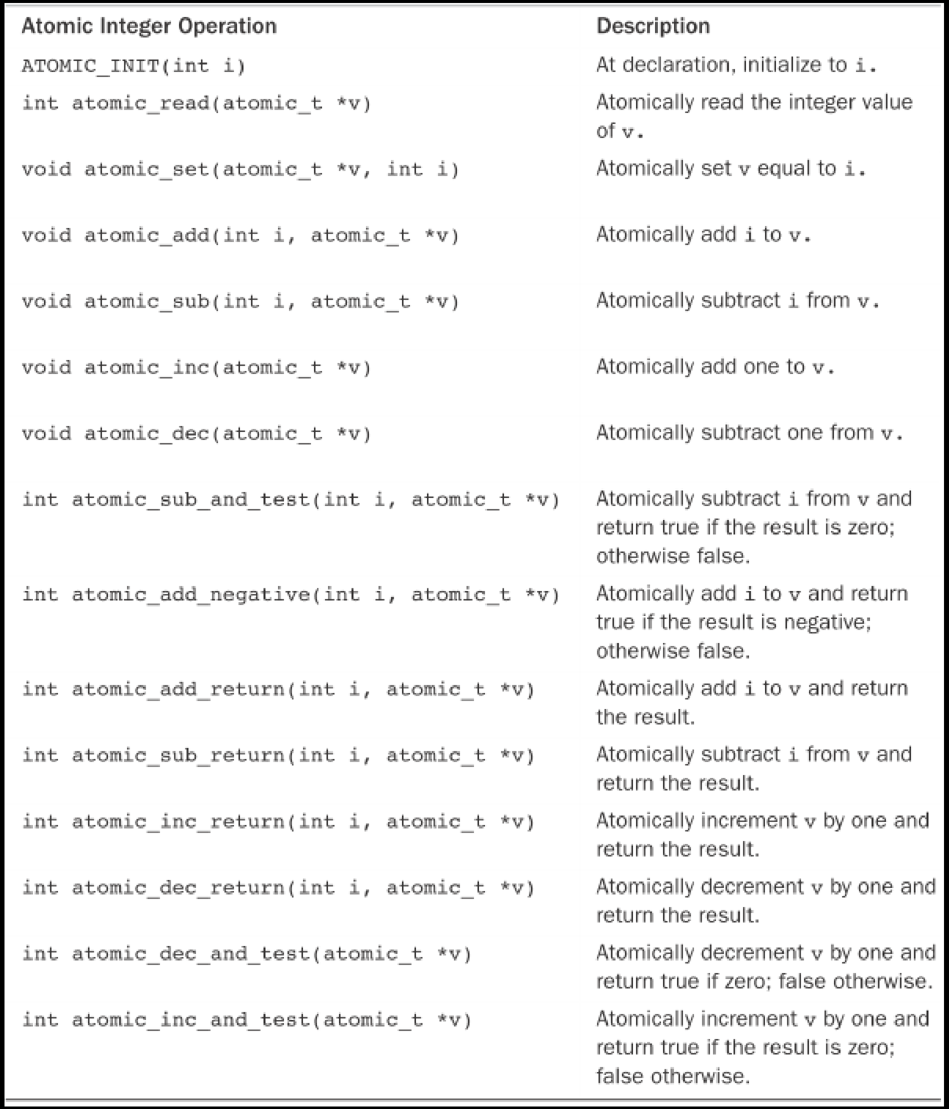
```
static inline int atomic_read(const atomic_t *v)
{
        return v->counter;
}
```



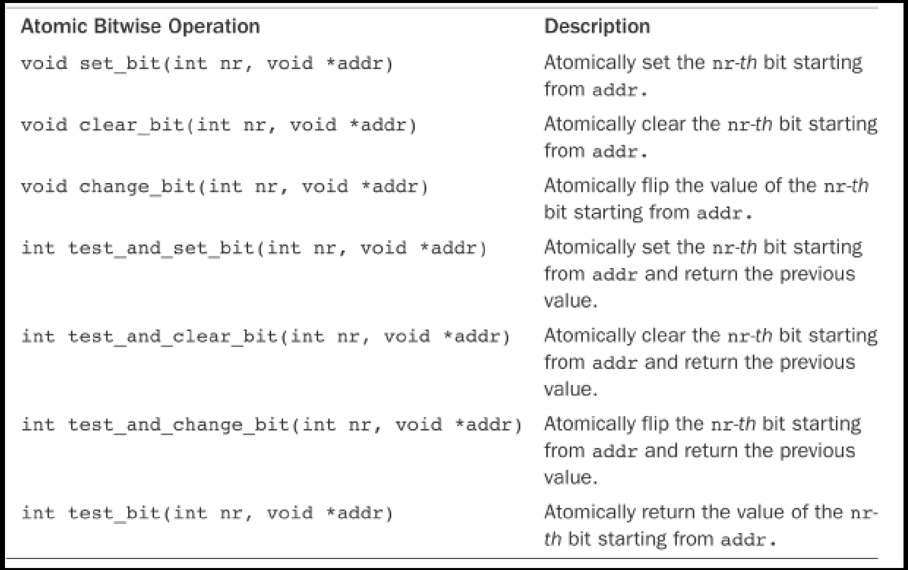
```
atomic_t v;                    /* define v */
atomic_t u = ATOMIC_INIT(0);  /* define u and initialize it to zero */
atomic_set(&v, 4);     /* v = 4 (atomically) */
atomic_add(2, &v);     /* v = v + 2 = 6 (atomically) */
atomic inc(&v);        /* v = v + 1 = 7 (atomically) */
printk("%d\n", atomic_read(&v));   /* will print "7" */
```



```
set_bit(0, &word);       /* bit zero is now set (atomically) */
set_bit(1, &word);       /* bit one is now set (atomically) */
printk("%ul\n", word);   /* will print "3" */
clear_bit(1, &word);     /* bit one is now unset (atomically) */
change_bit(0, &word);    /* bit zero is flipped; now it is unset (atomically) */

/* atomically sets bit zero and returns the previous value (zero) */
if (test_and_set_bit(0, &word)) {
        /* never true ... */
}

/* the following is legal; you can mix atomic bit instructions with normal C */
word = 7;
```



Atomic Integer Operation	Description
`ATOMIC_INIT(int i)`	At declaration, initialize to `i`.
`int atomic_read(atomic_t *v)`	Atomically read the integer value of `v`.
`void atomic_set(atomic_t *v, int i)`	Atomically set `v` equal to `i`.
`void atomic_add(int i, atomic_t *v)`	Atomically add `i` to `v`.
`void atomic_sub(int i, atomic_t *v)`	Atomically subtract `i` from `v`.
`void atomic_inc(atomic_t *v)`	Atomically add one to `v`.
`void atomic_dec(atomic_t *v)`	Atomically subtract one from `v`.
`int atomic_sub_and_test(int i, atomic_t *v)`	Atomically subtract `i` from `v` and return true if the result is zero; otherwise false.
`int atomic_add_negative(int i, atomic_t *v)`	Atomically add `i` to `v` and return true if the result is negative; otherwise false.
`int atomic_add_return(int i, atomic_t *v)`	Atomically add `i` to `v` and return the result.
`int atomic_sub_return(int i, atomic_t *v)`	Atomically subtract `i` from `v` and return the result.
`int atomic_inc_return(int i, atomic_t *v)`	Atomically increment `v` by one and return the result.
`int atomic_dec_return(int i, atomic_t *v)`	Atomically decrement `v` by one and return the result.
`int atomic_dec_and_test(atomic_t *v)`	Atomically decrement `v` by one and return true if zero; false otherwise.
`int atomic_inc_and_test(atomic_t *v)`	Atomically increment `v` by one and return true if the result is zero; false otherwise.

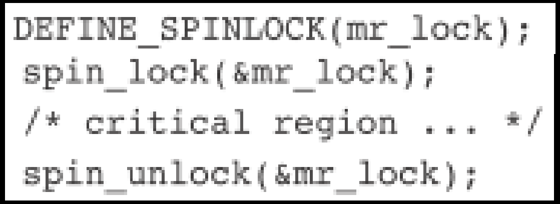


Atomic Bitwise Operation	Description
`void set_bit(int nr, void *addr)`	Atomically set the `nr`-*th* bit starting from `addr`.
`void clear_bit(int nr, void *addr)`	Atomically clear the `nr`-*th* bit starting from `addr`.
`void change_bit(int nr, void *addr)`	Atomically flip the value of the `nr`-*th* bit starting from `addr`.
`int test_and_set_bit(int nr, void *addr)`	Atomically set the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_and_clear_bit(int nr, void *addr)`	Atomically clear the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_and_change_bit(int nr, void *addr)`	Atomically flip the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_bit(int nr, void *addr)`	Atomically return the value of the `nr`-*th* bit starting from `addr`.

`<asm/atomic.h>`

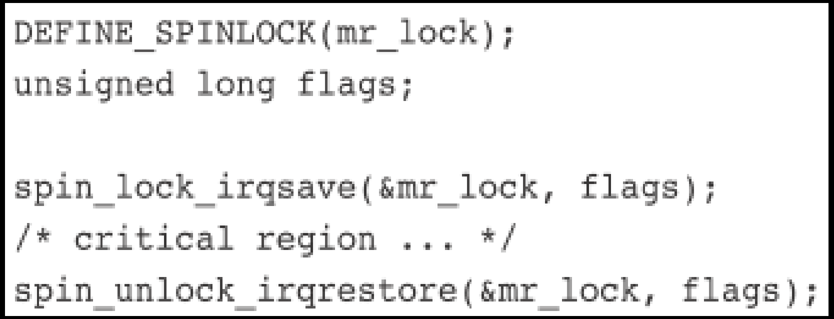
Spinlock

- Atomic can only work for variables. Complicate function can us lock
- simple, busy-waiting, should not hold for a long time (t < 2*context sw)
- Not Recursive

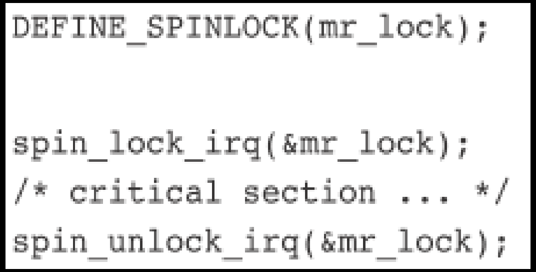


```
DEFINE_SPINLOCK(mr_lock);
 spin_lock(&mr_lock);
 /* critical region ... */
 spin_unlock(&mr_lock);
```

<linux/spinlock.h>



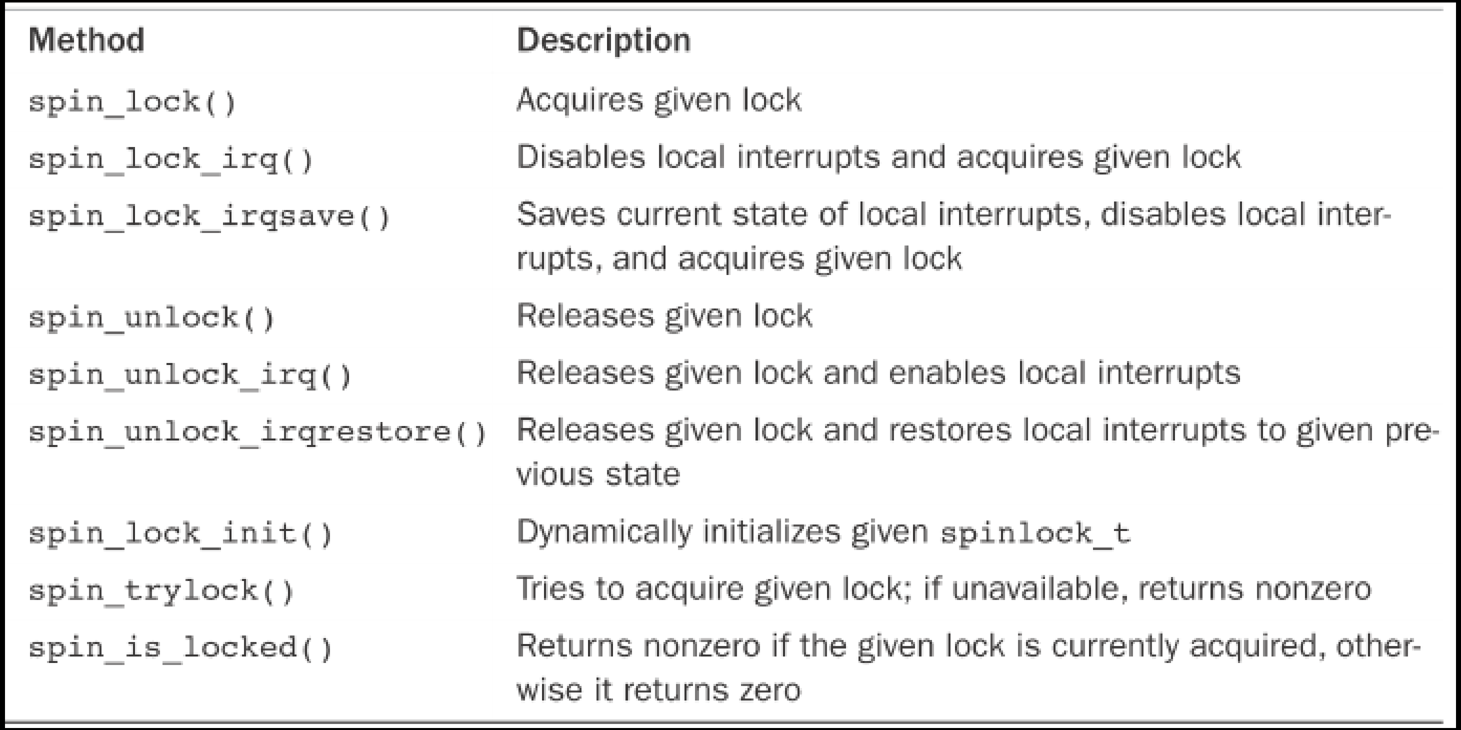
```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */
spin_unlock_irqrestore(&mr_lock, flags);
```



```
DEFINE_SPINLOCK(mr_lock);


spin_lock_irq(&mr_lock);
/* critical section ... */
spin_unlock_irq(&mr_lock);
```

Initialize:

```
    #DEFINE_SPINLOCK(my_lock);
        or
    spin_lock_init(spinlock_t *lock);
```



Method	Description
`spin_lock()`	Acquires given lock
`spin_lock_irq()`	Disables local interrupts and acquires given lock
`spin_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires given lock
`spin_unlock()`	Releases given lock
`spin_unlock_irq()`	Releases given lock and enables local interrupts
`spin_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`spin_lock_init()`	Dynamically initializes given `spinlock_t`
`spin_trylock()`	Tries to acquire given lock; if unavailable, returns nonzero
`spin_is_locked()`	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Bottom Halve:

```
    spin_lock_bh(spinlock_t *lock);
    spin_unlock_bh(spinlock_t *lock);
```

Bottom half preempt process context code,
if data share between BH process context, we use:
1. A lock 2. Disable bottom halves. to protect

Interrupt handler may preempt a bottom half,
if data share between interrupt handler and a BH, we use:
1. A lock 2. Disable interrupt. to protect

Read-Write Lock

- The RW lock of linux
  - One or more readers can concurrently hold the reader lock
  - Only one write lock (shared and exclusive form)

```
DEFINE_RWLOCK(mr_rwlock);

read_lock(&mr_rwlock);
/* critical section (read only)*/
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* critical section (read and write)*/
write_unlock(&mr_lock);
```

Warning:

A read-write lock pair may cause deadlock

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

If cannot serperate read & write:

use spinlock

wrIte(X) : read(O)

use `read_lock()` is enough

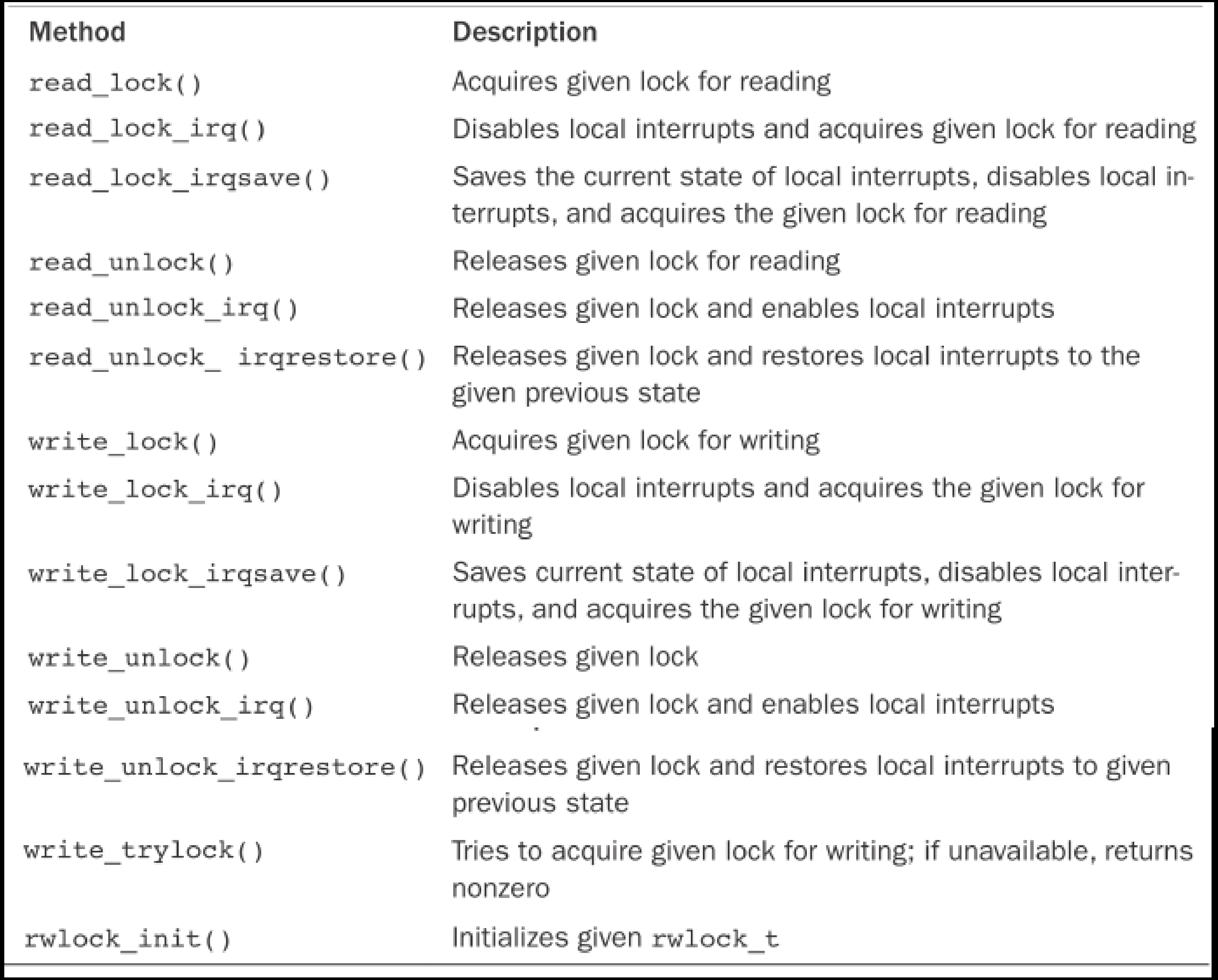
write(O) : read(O)

disable interrupts for write access

use `write_lock_irqsave()`

Reader favorite

May lead to write starvation



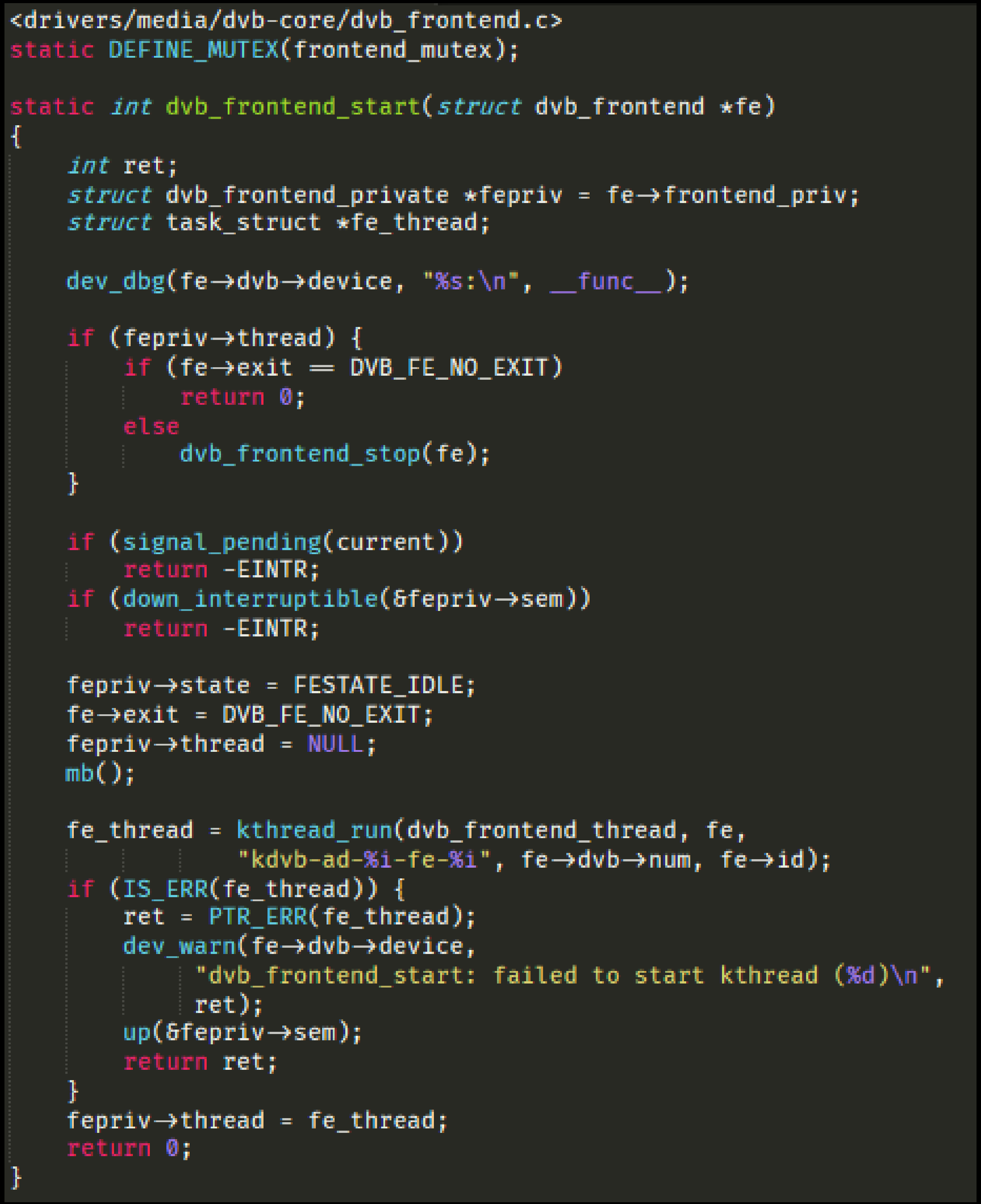
Method	Description
`read_lock()`	Acquires given lock for reading
`read_lock_irq()`	Disables local interrupts and acquires given lock for reading
`read_lock_irqsave()`	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
`read_unlock()`	Releases given lock for reading
`read_unlock_irq()`	Releases given lock and enables local interrupts
`read_unlock_ irqrestore()`	Releases given lock and restores local interrupts to the given previous state
`write_lock()`	Acquires given lock for writing
`write_lock_irq()`	Disables local interrupts and acquires the given lock for writing
`write_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing
`write_unlock()`	Releases given lock
`write_unlock_irq()`	Releases given lock and enables local interrupts
`write_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`write_trylock()`	Tries to acquire given lock for writing; if unavailable, returns nonzero
`rwlock_init()`	Initializes given `rwlock_t`

Semaphore

- Sleeping locks, push to wait queue to sleep if the lock is occupied
- Wake up if when the semaphore is released.
- Lead to:
  1. Better CPU utility
  2. Higher cost than spinlock (the waiting time is important)
- Obtain semaphore in process not interrupt context(not schedulable).

Initialize:

```
struct semaphore name;
sema_init(&name, count);

sema_init(sem, count);
static DECLARE_MUTEX(name); // binary semaphore
init_MUTEX(sem);
```



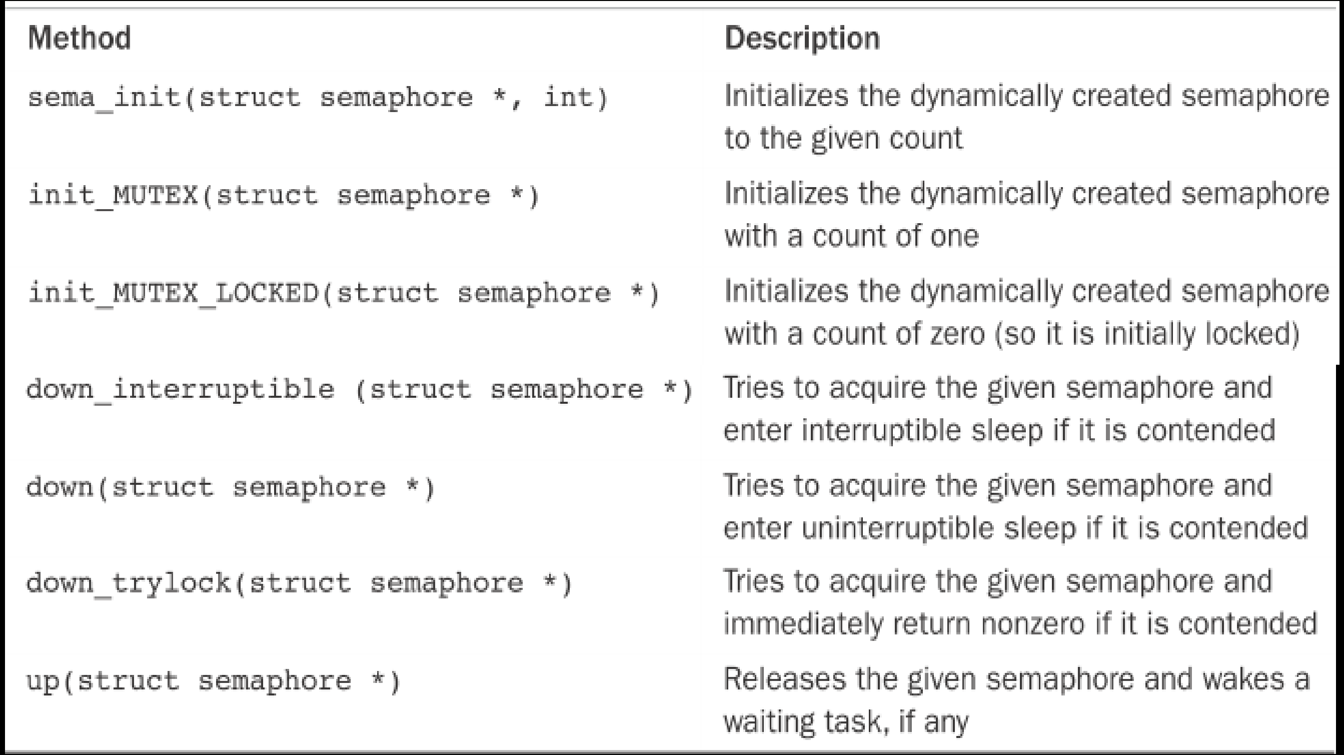
```
<drivers/media/dvb-core/dvb_frontend.c>
static DEFINE_MUTEX(frontend_mutex);

static int dvb_frontend_start(struct dvb_frontend *fe)
{
	int ret;
	struct dvb_frontend_private *fepriv = fe->frontend_priv;
	struct task_struct *fe_thread;

	dev_dbg(fe->dvb->device, "%s:\n", __func__);

	if (fepriv->thread) {
		if (fe->exit == DVB_FE_NO_EXIT)
			return 0;
		else
			dvb_frontend_stop(fe);
	}

	if (signal_pending(current))
		return -EINTR;
	if (down_interruptible(&fepriv->sem))
		return -EINTR;

	fepriv->state = FESTATE_IDLE;
	fe->exit = DVB_FE_NO_EXIT;
	fepriv->thread = NULL;
	mb();

	fe_thread = kthread_run(dvb_frontend_thread, fe,
			"kdvb-ad-%i-fe-%i", fe->dvb->num, fe->id);
	if (IS_ERR(fe_thread)) {
		ret = PTR_ERR(fe_thread);
		dev_warn(fe->dvb->device,
				"dvb_frontend_start: failed to start kthread (%d)\n",
				ret);
		up(&fepriv->sem);
		return ret;
	}
	fepriv->thread = fe_thread;
	return 0;
}
```

- Initialize counting semaphore

```
struct semaphore name;
sema_init(&name, count);

sema_init(sem, count);

static DECLARE_MUTEX(name); // binary semaphore

init_MUTEX(sem);
```

- Application

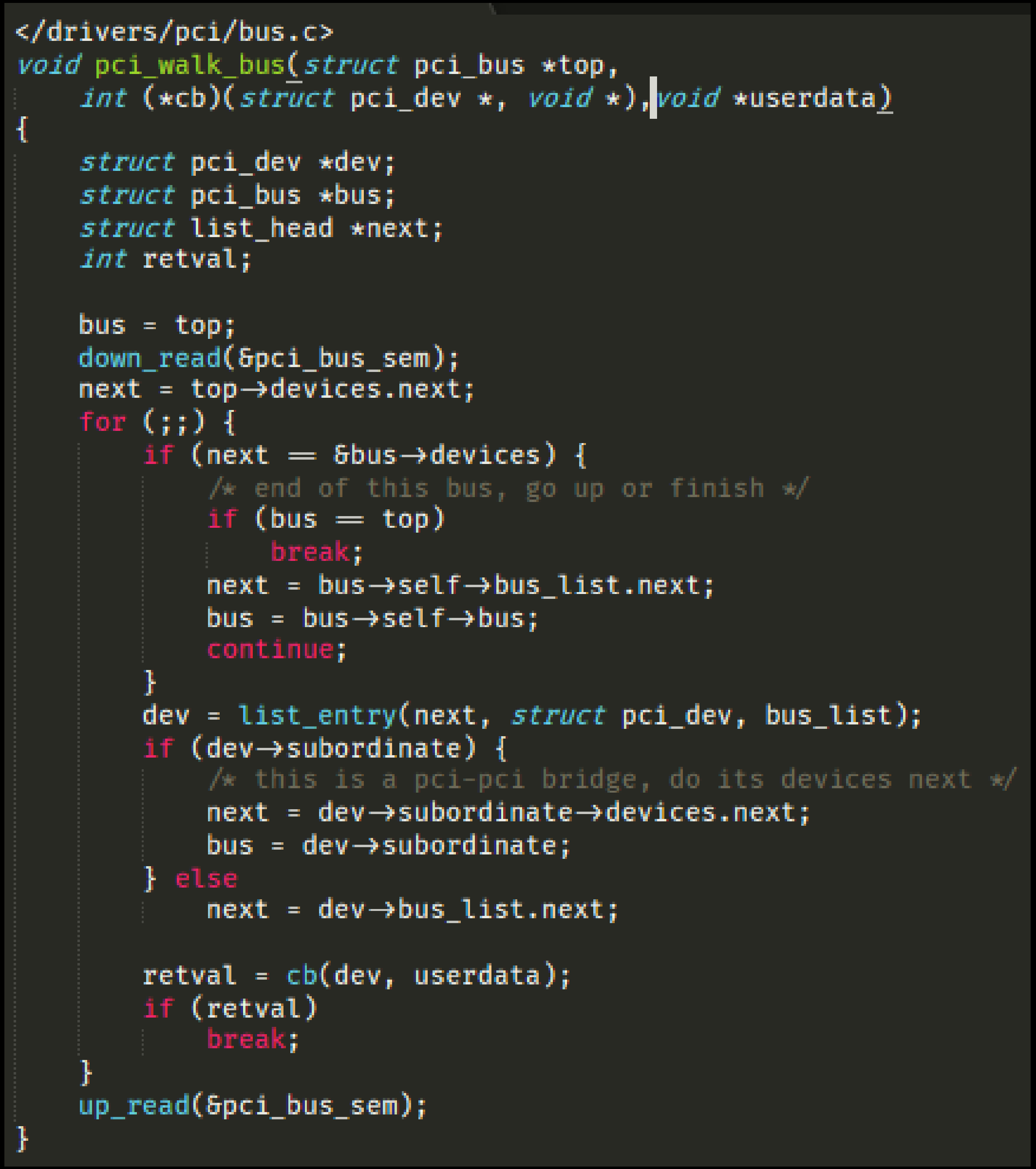
```
/* define and declare a semaphore, named mr_sem, with a count of one */

static DECLARE_MUTEX(mr_sem);

/* attempt to acquire the semaphore ... */

if (down_interruptible(&mr_sem)) {
/* signal received, semaphore not acquired ... */
}

/* critical region ... */
/* release the given semaphore */

up(&mr_sem);
```



Method	Description
`sema_init(struct semaphore *, int)`	Initializes the dynamically created semaphore to the given count
`init_MUTEX(struct semaphore *)`	Initializes the dynamically created semaphore with a count of one
`init_MUTEX_LOCKED(struct semaphore *)`	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
`down_interruptible (struct semaphore *)`	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
`down(struct semaphore *)`	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
`down_trylock(struct semaphore *)`	Tries to acquire the given semaphore and immediately return nonzero if it is contended
`up(struct semaphore *)`	Releases the given semaphore and wakes a waiting task, if any

Read-Write Semaphore

- Initialize:
  - `static DECLARE_RWSEM(mr_rwsem);`
  - `init_rwsem(struct rw_semaphore *sem)`
- Functions
  - `down_read(&mr_rwsem);`
  - `up_read(&mr_rwsem);`
  - `down_write(&mr_rwsem);`
  - `up_write(&mr_sem);`
  - `down_read_trylock(*sem);`
  - `down_write_trylock(*sem);`
  - `downgrade_write(*sem);`
    `// dynamically turn write to read`



```
</drivers/pci/bus.c>
void pci_walk_bus(struct pci_bus *top,
    int (*cb)(struct pci_dev *, void *), void *userdata)
{
    struct pci_dev *dev;
    struct pci_bus *bus;
    struct list_head *next;
    int retval;

    bus = top;
    down_read(&pci_bus_sem);
    next = top->devices.next;
    for (;;) {
        if (next == &bus->devices) {
            /* end of this bus, go up or finish */
            if (bus == top)
                break;
            next = bus->self->bus_list.next;
            bus = bus->self->bus;
            continue;
        }
        dev = list_entry(next, struct pci_dev, bus_list);
        if (dev->subordinate) {
            /* this is a pci-pci bridge, do its devices next */
            next = dev->subordinate->devices.next;
            bus = dev->subordinate;
        } else
            next = dev->bus_list.next;

        retval = cb(dev, userdata);
        if (retval)
            break;
    }
    up_read(&pci_bus_sem);
}
```

Mutex

- Initialize:
  - `DEFINE_MUTEX(name);`
  - `mutex_init(&mutex);`
- Functions
  - `mutex_lock(struct mutex *)`
  - `mutex_unlock(struct mutex *)`
  - `mutex_trylock(struct mutex *)`
  - `mutex_is_locked (struct mutex *)`

- Not just binary semaphore, STRICTER
  1. Lock, unlock from the same context.
  2. Cannot exit while holding a mutex.
  3. Cannot used by an interrupt handler or bottom half

  …

- Prefer using mutex than semaphore

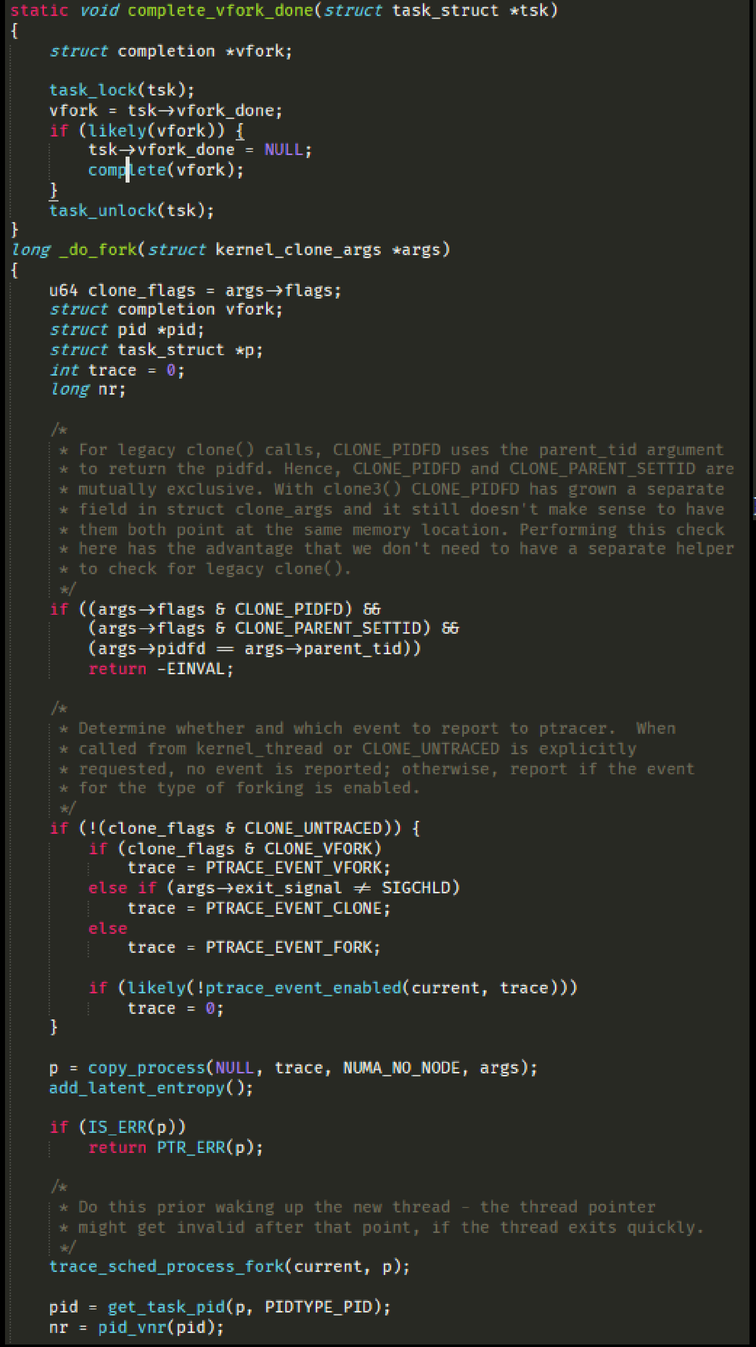
Requirement	Recommended Lock
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Mutex is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Mutex is required.

Completion Variables

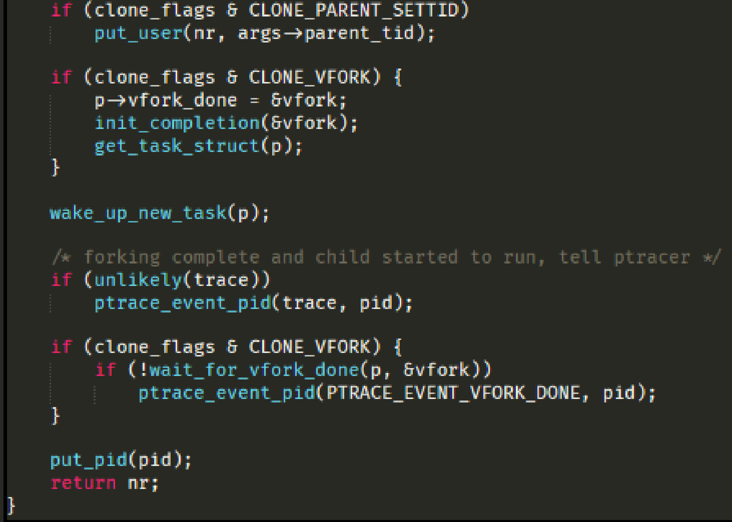
- one task needs to signal to the other that an event has occurred

- Initialize:
- DECLARE_COMPLETION(mr_comp);
- init_completion();

- Functions
- init_completion(struct completion *)
- wait_for_completion(struct completion *)
- complete(struct completion *)



```
static void complete_vfork_done(struct task_struct *tsk)
{
    struct completion *vfork;

    task_lock(tsk);
    vfork = tsk->vfork_done;
    if (likely(vfork)) {
        tsk->vfork_done = NULL;
        complete(vfork);
    }
    task_unlock(tsk);
}
long _do_fork(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p;
    int trace = 0;
    long nr;

    /*
     * For legacy clone() calls, CLONE_PIDFD uses the parent_tid argument
     * to return the pidfd. Hence, CLONE_PIDFD and CLONE_PARENT_SETTID are
     * mutually exclusive. With clone3() CLONE_PIDFD has grown a separate
     * field in struct clone_args and it still doesn't make sense to have
     * them both point at the same memory location. Performing this check
     * here has the advantage that we don't need to have a separate helper
     * to check for legacy clone().
     */
    if ((args->flags & CLONE_PIDFD) &&
        (args->flags & CLONE_PARENT_SETTID) &&
        (args->pidfd == args->parent_tid))
        return -EINVAL;

    /*
     * Determine whether and which event to report to ptracer.  When
     * called from kernel_thread or CLONE_UNTRACED is explicitly
     * requested, no event is reported; otherwise, report if the event
     * for the type of forking is enabled.
     */
    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if (args->exit_signal != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }

    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    add_latent_entropy();

    if (IS_ERR(p))
        return PTR_ERR(p);

    /*
     * Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
     */
    trace_sched_process_fork(current, p);

    pid = get_task_pid(p, PIDTYPE_PID);
    nr = pid_vnr(pid);
```

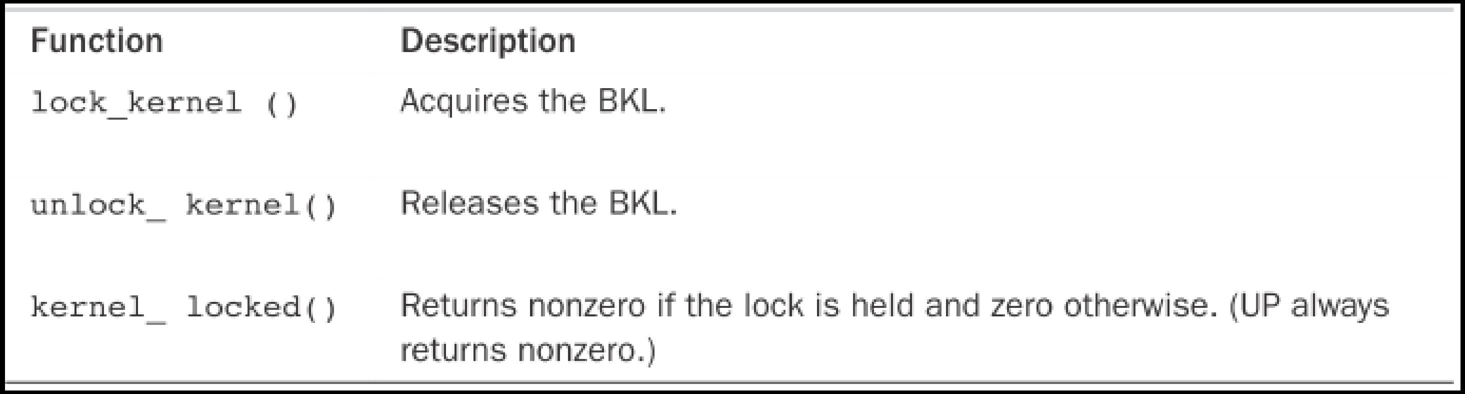


```
    if (clone_flags & CLONE_PARENT_SETTID)
        put_user(nr, args->parent_tid);

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }

    wake_up_new_task(p);

    /* forking complete and child started to run, tell ptracer */
    if (unlikely(trace))
        ptrace_event_pid(trace, pid);

    if (clone_flags & CLONE_VFORK) {
        if (!wait_for_vfork_done(p, &vfork))
            ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
    }

    put_pid(pid);
    return nr;
}
```

Big Kernel Lock

- A Transition mechanism from linux 2.0 to 2.2
- Fine-grained locking for SMP
  - You can sleep while holding the BKL. The lock is automatically dropped when the task is unscheduled and reacquired when the task is rescheduled

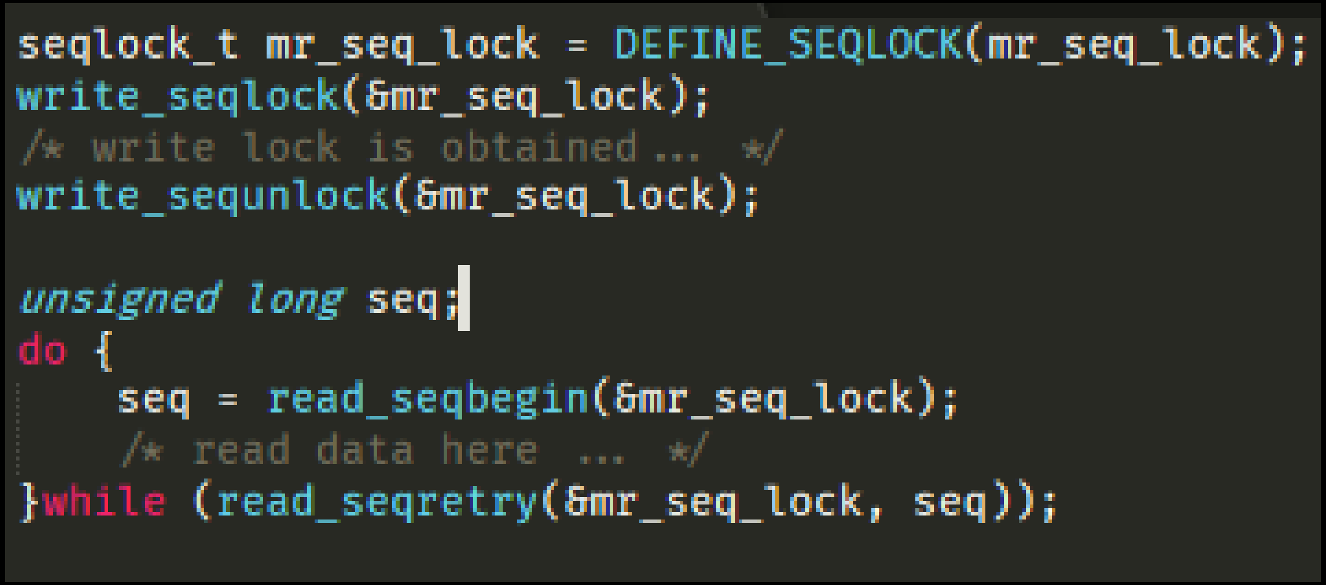
    => avoid deadlock cause by sleeping.
  - The BKL is a recursive lock.
  - You can use the BKL only in process context, not interrupt context



Function	Description
`lock_kernel ()`	Acquires the BKL.
`unlock_ kernel()`	Releases the BKL.
`kernel_ locked()`	Returns nonzero if the lock is held and zero otherwise. (UP always returns nonzero.)

Sequential Locks

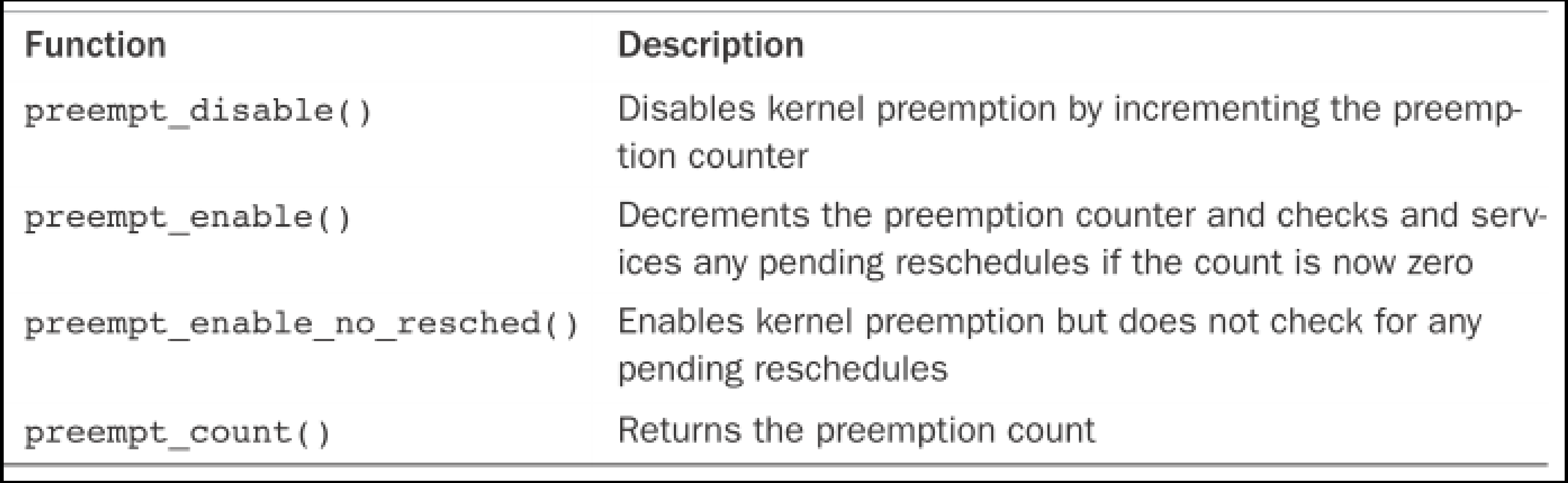
- A better method for writer, which readers can not starve writer
- Reader may read multiple times to get the latest version
- The sequence number increase when writing
- Same number: not interrupt yet; Odd: is writing; Even: finish writing



```
seqlock_t mr_seq_lock = DEFINE_SEQLOCK(mr_seq_lock);
write_seqlock(&mr_seq_lock);
/* write lock is obtained ... */
write_sequnlock(&mr_seq_lock);

unsigned long seq;
do {
    seq = read_seqbegin(&mr_seq_lock);
    /* read data here ... */
}while (read_seqretry(&mr_seq_lock, seq));
```

Preemption Disabling

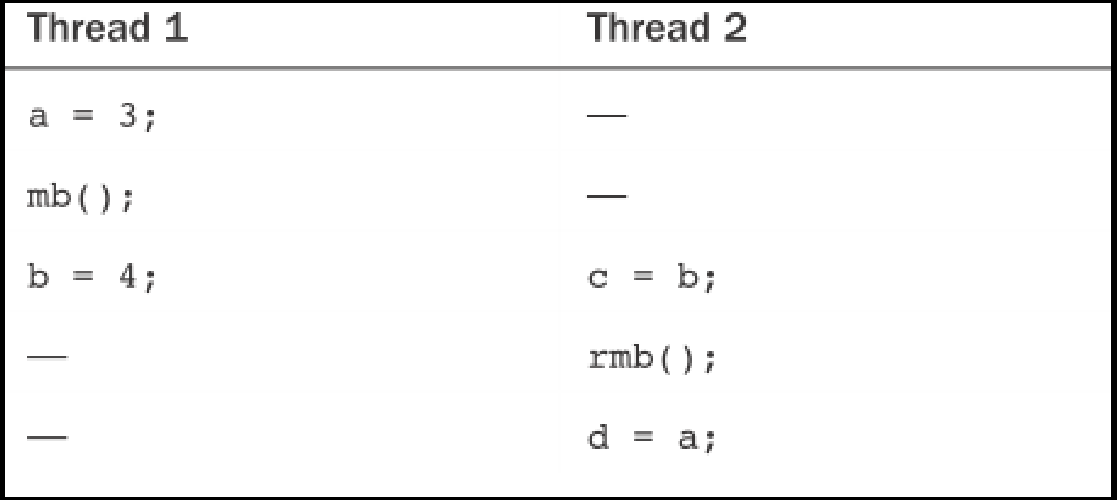
- Kernel is preemptive, may preempt when another process in CS
- 1. spinlock(preempt-free) 2. preempt_disable()



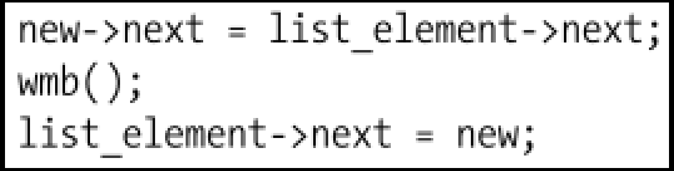
Function	Description
`preempt_disable()`	Disables kernel preemption by incrementing the preemption counter
`preempt_enable()`	Decrements the preemption counter and checks and services any pending reschedules if the count is now zero
`preempt_enable_no_resched()`	Enables kernel preemption but does not check for any pending reschedules
`preempt_count()`	Returns the preemption count

Ordering and Barriers

- compiler and processor can reorder reads and writes for performance reasons (compiler time, runtime)
- barrier is use as a block



Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`b = 4;`	`c = b;`
—	`rmb();`
—	`d = a;`

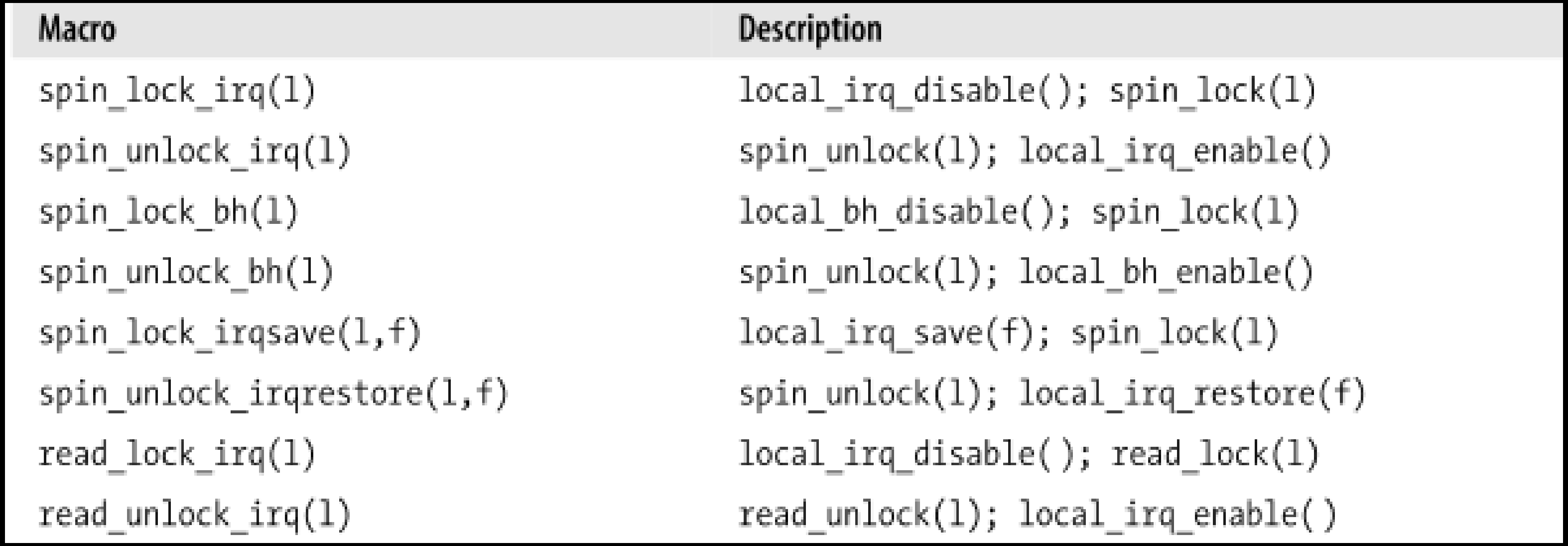


```
new->next = list_element->next;
wmb();
list_element->next = new;
```

Barrier	Description
`rmb()`	Prevents loads from being reordered across the barrier
`read_barrier_depends()`	Prevents data-dependent loads from being re-ordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents load or stores from being reordered across the barrier
`smp_rmb()`	Provides an `rmb()` on SMP, and on UP provides a `barrier()`
`smp_read_barrier_depends()`	Provides a `read_barrier_depends()` on SMP, and provides a `barrier()` on UP
`smp_wmb()`	Provides a `wmb()` on SMP, and provides a `barrier()` on UP
`smp_mb()`	Provides an `mb()` on SMP, and provides a `barrier()` on UP
`barrier()`	Prevents the compiler from optimizing stores or loads across the barrier

Protect a data accessed by interrupts

- If multiple irq access data at same time…, even multiple cpu
- In uniprocessor, disable interrupt in CS
- In multiprocessor,
  disabling local interrupts + use spin lock to protect data.



Macro	Description
spin_lock_irq(l)	local_irq_disable(); spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l); local_irq_enable()
spin_lock_bh(l)	local_bh_disable(); spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l); local_bh_enable()
spin_lock_irqsave(l,f)	local_irq_save(f); spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l); local_irq_restore(f)
read_lock_irq(l)	local_irq_disable(); read_lock(l)
read_unlock_irq(l)	read_unlock(l); local_irq_enable()

Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock (see Table 5-9)
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock