A STUDY ON LINUX KERNEL SCHEDULER Version 2.6.32

Author: Thang Minh Le
(thangmle@gmail.com)

1. INTRODUCTION

Linux scheduler has been gone through some big improvements since kernel version 2.4. There were a lot of complaints about the interactivity of the scheduler in kernel 2.4. During this version, the scheduler was implemented with one running queue for all available processors. At every scheduling, this queue was locked and every task on this queue got its timeslice update. This implementation caused poor performance in all aspects. The scheduler algorithm and supporting code went through a large rewrite early in the 2.5 kernel development series. The new scheduler was arisen to achieve *O(1)* run-time regardless number of runnable tasks in the system. To achieve this, each processor has its own running queue. This helps a lot in reducing lock contention. The priority array was introduced which used active array and expired array to keep track running tasks in the system. The *O(1)* running time is primarily drawn from this new data structure. The scheduler puts all expired processes into expired array. When there is no active process available in active array, it swaps active array with expired array, which makes active array becomes expired array and expired array becomes active array. There were some twists made into this scheduler to optimize further by putting expired task back to active array instead of expired array in some cases. *O(1)* scheduler uses a heuristic calculation to update dynamic priority of tasks based on their interactivity (I/O bound versus CPU bound) The industry was happy with this new scheduler until Con Kolivas introduced his new scheduler named *Rotating Staircase Deadline* (RSDL) and then later *Staircase Deadline* (SD). His new schedulers proved the fact that fair scheduling among processes can be achieved without any complex computation. His scheduler was designed to run in *O(n)* but its performance exceeded the current *O(1)* scheduler.

The result achieved from SD scheduler surprised all kernel developers and designers. The fair scheduling approach in SD scheduler encouraged Igno Molnar to re-implement the new Linux scheduler named *Completely Fair Scheduler* (CFS). CFS scheduler was a big improvement over the existing scheduler not only in its performance and interactivity but also in simplifying the scheduling logic and putting more modularized code into the scheduler. CFS scheduler was merged into mainline version 2.6.23. Since then, there have been some minor improvements made to CFS scheduler in some areas such as optimization, load balancing and group scheduling feature.

This study will focus on the latest CFS scheduler in Linux kernel 2.6.32.

2. OVERVIEW

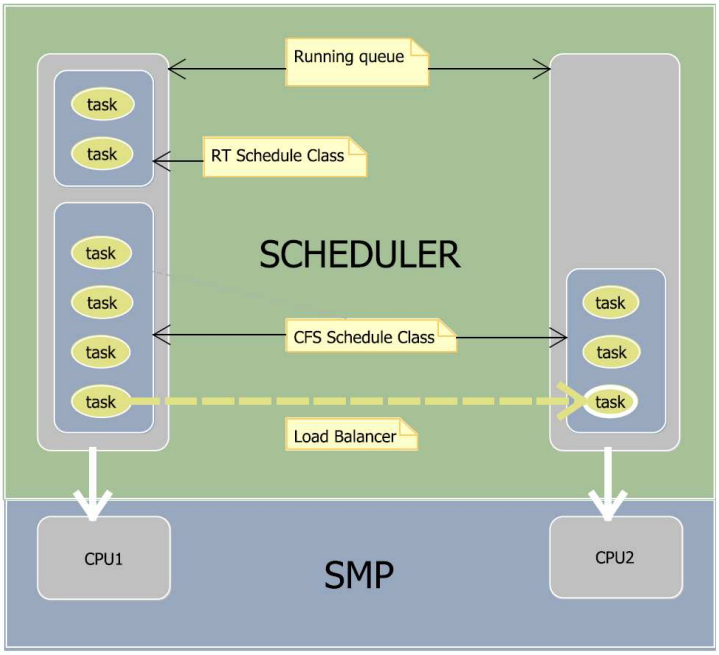


Figure.1 Linux Scheduler Overview

Linux scheduler contains:

- **Running queue:** a running queue (rq) is created for each processor (CPU). It is defined in *kernel/sched.c* as *struct_runqueue*. Each rq contains a list of runnable processes on a given processor. The *struct_runqueue* is defined in *sched.c* not *sched.h* to abstract the internal data structure of the scheduler.
- **Schedule class:** schedule class was introduced in 2.6.23. It is an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are called from the scheduler core without the core code assuming too much about them. Scheduling classes are implemented through the *sched_class* structure, which contains hooks to functions that must be called whenever an interesting event occurs. Tasks refer to their schedule policy through *struct task_struct.sched_class.* There are two schedule classes implemented in 2.6.32:
    1. **Completely Fair Schedule class:** schedules tasks following Completely Fair Scheduler (CFS) algorithm. Tasks which have policy set to *SCHED_NORMAL (SCHED_OTHER), SCHED_BATCH, SCHED_IDLE* are scheduled by this schedule class. The implementation of this class is *kernel/sched_fair.c*
    2. **RT schedule class:** schedules tasks following real-time mechanism defined in POSIX standard. Tasks which have policy set to *SCHED_FIFO, SCHED_RR* are scheduled using this schedule class. The implementation of this class is *kernel/sched_rt.c*
- **Load balancer:** In SMP environment, each CPU has its own rq. These queues might be unbalanced from time to time. A running queue with empty task pushes its associated CPU to idle, which does not take full advantage of symmetric multiprocessor systems. Load balancer is to address this issue. It is called every time the system requires scheduling tasks. If running queues are unbalanced, load balancer will try to pull idle tasks from busiest processors to idle processor.

3. SCHEDULER

3.1. DATA STRUCTURE

```
struct task_struct {
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 void *stack;
 atomic_t usage;
 unsigned int flags; /* per process flags, defined below */
 unsigned int ptrace;
 int lock_depth;  /* BKL lock depth */
 int prio, static_prio, normal_prio;
 unsigned int rt_priority;
 const struct sched_class *sched_class;
 struct sched_entity se;
 struct sched_rt_entity rt;
 unsigned int policy;
 cpumask_t cpus_allowed;
 struct list_head tasks;
 struct plist_node pushable_tasks;
 struct mm_struct *mm, *active_mm;
 int exit_state;
 int exit_code, exit_signal;
 int pdeath_signal;  /*  The signal sent when the parent dies  */
 unsigned in_iowait:1;
 pid_t pid;
 pid_t tgid;
 cputime_t utime, stime, utimescaled, stimescaled;
 cputime_t gtime;
 cputime_t prev_utime, prev_stime;
 unsigned long nvcsw, nivcsw; /* context switch counts */
 struct timespec start_time;   /* monotonic time */
 struct timespec real_start_time; /* boot based time */
 struct task_cputime cputime_expires;
 struct list_head cpu_timers[3];
 struct thread_struct thread;
 struct signal_struct *signal;
 struct sighand_struct *sighand;
 sigset_t blocked, real_blocked;
 sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
 struct sigpending pending;
 spinlock_t alloc_lock;
 spinlock_t pi_lock; /* Protection of the PI data structures: */
 struct io_context *io_context;
 struct rcu_head rcu;
 struct list_head *scm_work_list;
 unsigned long stack_start;
 ...
};
```

Figure.2.a struct task_struct

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
 spinlock_t lock; /* runqueue lock: */
 unsigned long nr_running; /* number of runnable tasks */
 unsigned long cpu_load[CPU_LOAD_IDX_MAX];
 struct load_weight load;
 unsigned long nr_load_updates;
 u64 nr_switches; /* number of context switches */
 u64 nr_migrations_in;
 struct cfs_rq cfs; /* CFS running queue */
 struct rt_rq rt; /* Realtime running queue */
 unsigned long nr_uninterruptible; /* number of tasks in uninterruptible sleep
*/
 struct task_struct *curr; /* this processor's currently running task */
 struct task_struct *idle; /* this processor's idle task */
 unsigned long next_balance;
 struct mm_struct *prev_mm;
 u64 clock;
 atomic_t nr_iowait;
 unsigned long calc_load_update;
 long calc_load_active;
};
```

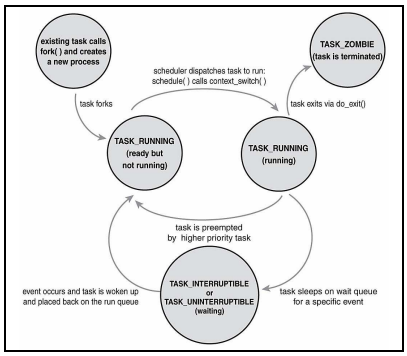
Figure.2.b struct rq

The most important data structures used in Linux scheduler are *struct task_struct* and *struct rq*. Many of methods in the kernel scheduler work with these two data structures.

STRUCT TASK_STRUCT

Each process in the system is represented by a *task_struct*. Since *task_struct* data type must be able to capture all information of a process, it is relatively large around 1.7KB in size. *Figure.2a* only shows some important fields which are frequently used by the scheduler. When a process/thread is created, the kernel allocates a new *task_struct* for it. The kernel then stores this *task_struct* in a circular linked list call *task_list*. There is a convenient macro *current* to obtain the current running process. Macro *next_task* and *prev_task* allow a process to obtain its next task and its previous task respectively. Linux scheduler uses many fields of task_struct for its scheduling task. Most important fields are:

- **State:** this field describes current state of process.



TASK_RUNNING: The process is runnable; it is either currently running or on a running queue waiting to run.
TASK_INTERRUPTIBLE: The process is sleeping (that is, it is blocked), waiting for some condition to exist.
TASK_UNINTERRUPTIBLE: The process is sleeping. It does not wake up and become runnable if it receives a signal.
TASK_ZOMBIE: The task has terminated, but its parent has not yet issued a wait() system call.
TASK STOPPED: Process execution has stopped; the task is not running nor is it eligible to run.

- **static_prio:** is static priority of a process. The value of this field does not get changed. Static priority is also called nice value which ranges from -20 to 19. Before 2.6.23, Linux scheduler depends heavily on the value of this field to perform heuristic calculation during scheduling task. In CFS scheduling policy, the scheduler no longer uses the value of this field in scheduling tasks.
- **prio:** this field holds dynamic priority of a process. Prior to 2.6.23, this field was calculated as a function of static priority (static_prio) and the task's interactivity. This calculation is done by method *effective_prio()* of *sched.c.* Since CFS scheduling policy was introduced, linux scheduler no longer uses the value stored in this field in scheduling tasks.
- **normal_prio:** holds expected priority of a process. In most cases, for non real-time processes, values of *normal_prio* and *static_prio* are the same. For real-time processes, the value of *normal_prio* might be boosted to avoid deadlock when accessing critical sections.
- **rt_priority:** this field is used for real-time process. It holds real-time priority value. Competition among real-time tasks is strictly based upon *rt_priority.*
- **sched_class:** a pointer points to schedule class.
- **sched_entity:** a pointer points to CFS schedule entity
- **sched_rt_entity:** a pointer points to RT schedule entity
- **policy:** holds a value of scheduling policies

STRUCT RQ

At boot time, Linux system creates running queue of type *struct rq* for each available processor. Each running queue captures these data:

- **nr_running:** number of runnable task.
- **nr_switches:** number of switches.
- **cfs:** CFS running queue structure.
- **rt:** RT running queue structure.
- **next_balance:** timestamp to next load balance check.
- **curr:** pointer points to currently running task of this running queue.
- **idle:** pointer points to currently idle task of this running queue.
- **lock:** spin lock of running queue. *task_rq_lock()* and *task_rq_unlock()* can be used to lock running queue which a specific task runs on. It is important to obtain running queue locks in the same order in the case of locking multiple queues.

STRUCT SCHED_CLASS: fair_sched_class & rt_sched_class

```
struct sched_class {
 const struct sched_class *next;

 void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
 void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
 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 (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
 int  (*select_task_rq)(struct task_struct *p, int sd_flag, int flags);

 unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
   struct rq *busiest, unsigned long max_load_move,
   struct sched_domain *sd, enum cpu_idle_type idle,
   int *all_pinned, int *this_best_prio);

 int (*move_one_task) (struct rq *this_rq, int this_cpu,
         struct rq *busiest, struct sched_domain *sd,
         enum cpu_idle_type idle);
 void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
 void (*post_schedule) (struct rq *this_rq);
 void (*task_wake_up) (struct rq *this_rq, struct task_struct *task);

 void (*set_cpus_allowed)(struct task_struct *p,
     const struct cpumask *newmask);

 void (*rq_online)(struct rq *rq);
 void (*rq_offline)(struct rq *rq);
#endif

 void (*set_curr_task) (struct rq *rq);
 void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
 void (*task_new) (struct rq *rq, struct task_struct *p);

 void (*switched_from) (struct rq *this_rq, struct task_struct *task,
          int running);
 void (*switched_to) (struct rq *this_rq, struct task_struct *task,
        int running);
 void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
        int oldprio, int running);

 unsigned int (*get_rr_interval) (struct task_struct *task);

#ifdef CONFIG_FAIR_GROUP_SCHED
 void (*moved_group) (struct task_struct *p);
#endif
};
```

Figure.3.a struct sched_class

```
/*
 * All the scheduling class methods:
 */
static const struct sched_class fair_sched_class = {
 .next   = &idle_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,
 .put_prev_task  = put_prev_task_fair,

#ifdef CONFIG_SMP
 .select_task_rq  = select_task_rq_fair,

 .load_balance  = load_balance_fair,
 .move_one_task  = move_one_task_fair,
#endif

 .set_curr_task          = set_curr_task_fair,
 .task_tick  = task_tick_fair,
 .task_new  = task_new_fair,

 .prio_changed  = prio_changed_fair,
 .switched_to  = switched_to_fair,

 .get_rr_interval = get_rr_interval_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
 .moved_group  = moved_group_fair,
#endif
};
```

Figure.3.b

```
static const struct sched_class rt_sched_class = {
 .next   = &fair_sched_class,
 .enqueue_task  = enqueue_task_rt,
 .dequeue_task  = dequeue_task_rt,
 .yield_task  = yield_task_rt,

 .check_preempt_curr = check_preempt_curr_rt,

 .pick_next_task  = pick_next_task_rt,
 .put_prev_task  = put_prev_task_rt,

#ifdef CONFIG_SMP
 .select_task_rq  = select_task_rq_rt,

 .load_balance  = load_balance_rt,
 .move_one_task  = move_one_task_rt,
 .set_cpus_allowed       = set_cpus_allowed_rt,
 .rq_online              = rq_online_rt,
 .rq_offline             = rq_offline_rt,
 .pre_schedule  = pre_schedule_rt,
 .post_schedule  = post_schedule_rt,
 .task_wake_up  = task_wake_up_rt,
 .switched_from  = switched_from_rt,
#endif

 .set_curr_task          = set_curr_task_rt,
 .task_tick  = task_tick_rt,

 .get_rr_interval = get_rr_interval_rt,

 .prio_changed  = prio_changed_rt,
 .switched_to  = switched_to_rt,
};
```

Figure.3.c

struct sched_class was newly introduced in kernel 2.6.23. The intention is to remove the logic of scheduling policies from the main *sched.c* file. This makes the codes more modularized. Some of important methods defined in *sched_class* are:

- **enqueue_task()**: calls when a task enters a runnable state. It puts the scheduling entity (task) into the red-black tree and increments the *nr_running* variable.
- **dequeue_tree()**: when a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the *nr_running* variable.
- **yield_task():t**his function is basically just a dequeue followed by an enqueue, unless the *compat_yield sysctl* is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.
- **check_preempt_curr():** this function checks if a task that entered the runnable state should preempt the currently running task.
- **pick_next_task():** this function chooses the most appropriate task eligible to run next.
- **set_curr_task():** this function is called when a task changes its scheduling class or changes its task group.
- **task_tick():** this function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.

- **task_new():** The core scheduler gives the scheduling module an opportunity to manage new task startup. The CFS scheduling module uses it for group scheduling, while the scheduling module for a real-time task does not use it.

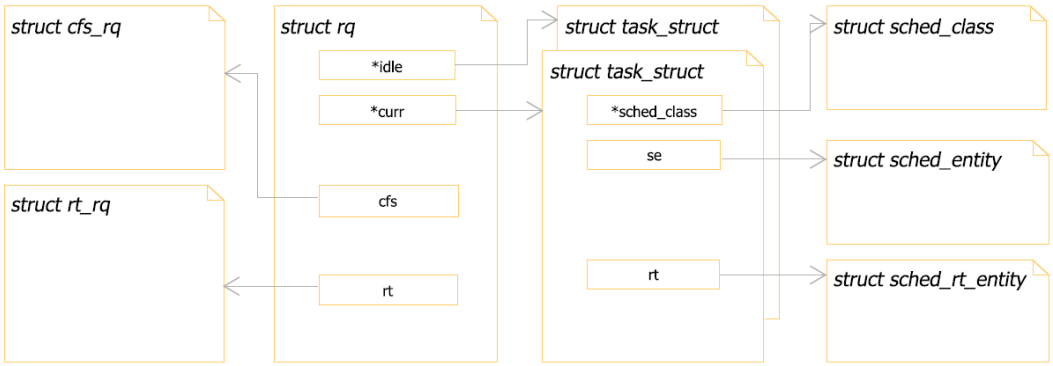


Figure.4 Data Structure Relation

3.2. SCHEDULING TASKS

schedule()

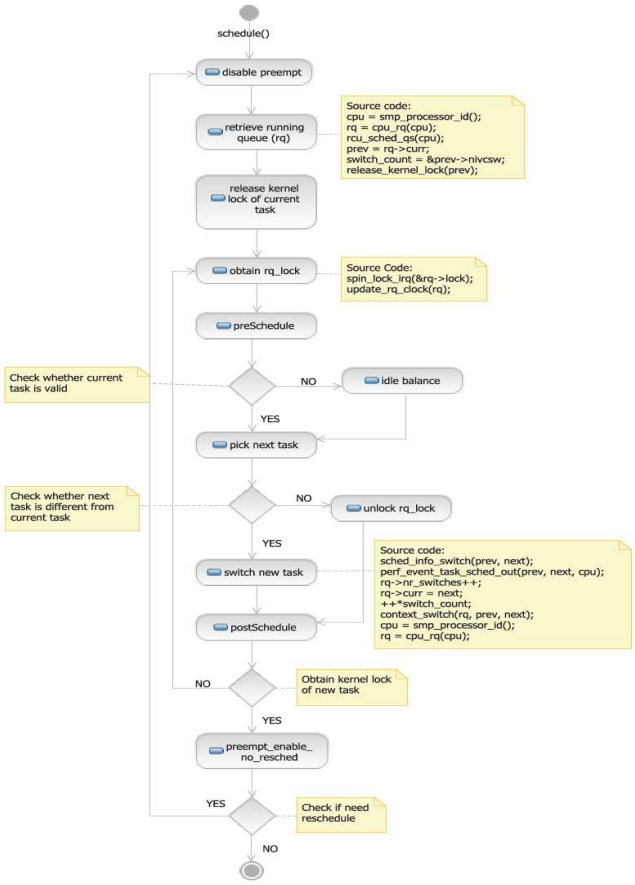


Figure.5 *schedule()* workflow

schedule() is the most important method of the scheduler. This method is responsible to pick the next task and switch current task with the next task. This method is executed whenever:

- A process voluntarily yields the CPU.
- A process waits for signal to occur or wants to sleep.
- Timer task occurs through *scheduler_tick()*.
- And other cases.

Step1: disables preemption. During execution scheduling logic, process preemption must be disabled. It is because we will obtain spin lock later.
Step 2: retrieves running queue based on current processor. We call *smp_processor_id()* to get current CPU. Calling *cpu_rq()* to retrieve rq of this CPU. Then, calling *release_kernel_lock()* to release kernel lock on the current task and obtain the lock of current rq.
Step 3: executes *pre_schedule()* method. Recalls that *pre_schedule()* method is defined in struct *sched_class*. This is the hook for scheduling policy to execute any logic before scheduler calls *pick_next_task()*. Currently, only RT schedule class implements this method. This is because Linux favors real-time tasks over normal tasks. Implementing *pre_schedule()* in RT schedule

class allows RT schedule to verify if it has any runnable real-time tasks at this point. If there is a runnable real-time task in the system, RT schedule will make this task to be current task.
Step 4: executes *pick_next_task()*. This method is called on the associated schedule class stored in *task_struct.sched_class* of the current task. If the current task is normal task, *pick_next_task()* of CFS schedule class will be execute, which will follow *Complete Fair Scheduling* algorithm to pick the next task. If the current task is real-time task, *pick_next_task()* of RT schedule class will be executed, which will follow POSIX real-time standard requirement.
Step 5: checks whether current task is the same as next task. If they are the same, there is no need context switch. Simply releases the lock of running queue and executes *post_schedule()*. Otherwise, context switch is required to switch current task with the next task. When switching tasks is done, the code calls *post_schedule()*.
Step 6: *post_schedule()* is a hook which allows RT schedule class to push real-time task on the current processor.
Step 7: acquires lock on the current task which might be the original task or the next task and enables preemption. The code then checks if we need to reschedule again; if so, we go back to corresponding label and redo scheduling.

context_switch()

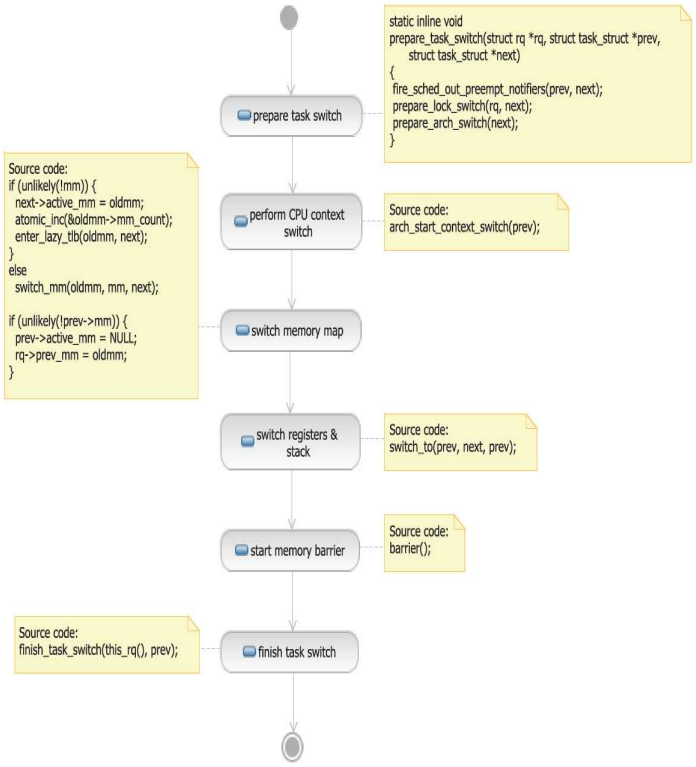


Figure.6 *context_switch()* workflow

context_switch() is called from *schedule()* to perform switching the current task and the next task. This method does the machine-specific work of switching process memory, registers and stack.

Step 1: prepares task switch by call *prepare_lock_switch()* and *prepare_arch_switch()*.
Step 2: performs CPU context switch. This will put CPU in the prepared state for context switching (CPU & MMU)
Step 3: starts to switch process virtual memory by calling *switch_mm()*. This step directly uses utility provided by different processors to load page table of the next task. On *x86*, it uses *cr3* register in *load_cr3()* method to complete memory switch. Most processes share the same LDT. If LDT is required, we just load it here from the new context
Step 4: switch process registers and process stack. This step involves mostly assembly codes which save value of current *esp* register to *prev* task structure. Then moving stack pointer of the new task to current processor *esp* register. We are now in new kernel stack. The code loads value of new kernel stack to cpu register. The *switch_to(prev, next, prev)* passes back prev in the *eax* register, which is the third parameter. In other words, the input parameter prev is passed out of the *switch_to()* macro as the output parameter last.
Step 5: we then request to start memory barrier optimization and then call *finish_task_switch()* to complete the context switch.

Schedule_fork()

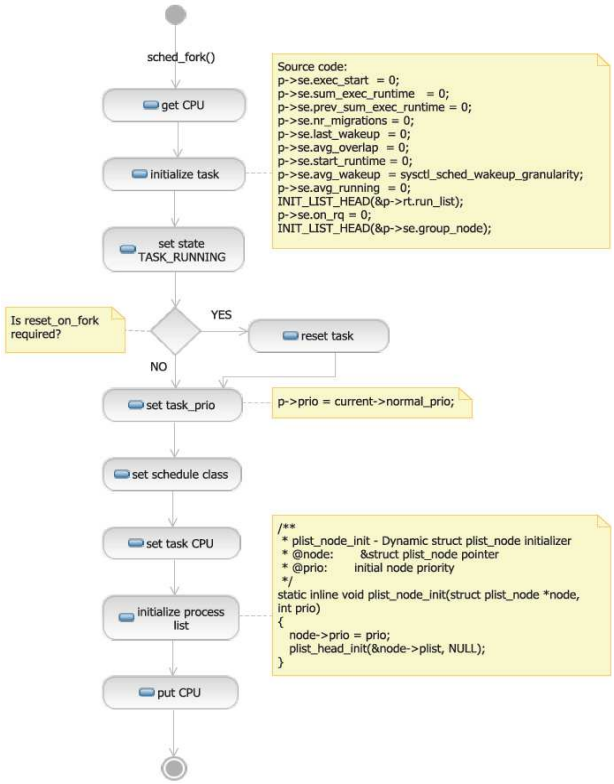


Figure.7 *schedule_fork()* workflow

schedule_fork() is called during f*ork()/clone()* system call. The method initializes all scheduling related fields defined in *struct task_struct*.

Step 1: it retrieves the current CPU in method *get_cpu()* by disabling preemption and get CPU id on SMP system.
Step 2: initializes value for fields of schedule entity of the new task_struct.
Step 3: sets state to *TASK_RUNNING*
Step 4: if reset on fork is required, it will reset new *task_struct*. Otherwise, it sets priority to the new task.
Step 5: sets schedule class for the new task based on its priority. If its priority is real-time priority, schedule class is RT schedule class. Otherwise, its schedule class is set to CFS schedule class.
Step 6: sets cpu id to *task_struct*'s cpu field.
Step 7: puta the new task to process list by calling *plist_head_init()*
Step 8: it then calls *put_cpu()* to enable preemption.

3.3. LOAD BALANCER

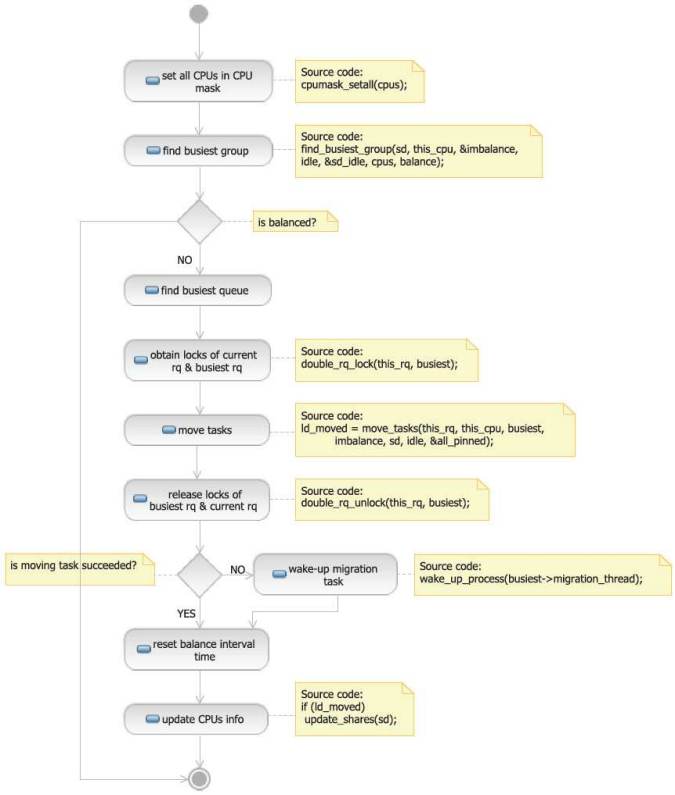


Figure.8 *load_balance()* workflow

load_balance() is called as part of *schedule()* process. It looks into CPU domain and tries to balance tasks between busiest CPUs and idle CPU. Load balancer performs these steps:

Step 1: sets all available CPUs in CPU mask. If there is a CPU, its associated bit map is set to 1.
Step 2: using above CPU mask, the code attempts to find the busiest group. Method *find_busiest_group()* returns the busiest group within the *sched_domain* if there is an imbalance. If there isn't an imbalance, and the user has opted for power-savings, it returns a group whose CPUs can be put to idle by rebalancing those tasks elsewhere, if such a group exists. Also calculates the amount of weighted load which should be moved to restore balance.
Step 3: checks if we can find the busiest group. If yes, the method continues. Otherwise, the execution returns.
Step 4: we just found the busiest group, let find the busiest running queues among this group.
Step 5: obtains double locks in order: current running queue & the busiest queue.
Step 6: begins moving task. Method *move_task()* tries to move up to *max_load_move* weighted load from busiest to *this_rq*, as part of a balancing operation within domain *sd*. Returns 1 if successful and 0 otherwise.
Step 7: release locks obtained in step 5 (releasing in reverse order when obtaining locks)
Step 8: checks whether moving task in step 6 is successful. If it is successful, the code resets balance interval time. Otherwise, we wake up migration task so that load balance will be kicked off later by this task.
Step 9: finally, the code updates CPU information after balancing.

3.4. CFS SCHEDULE CLASS

In theory, CFS scheduler should be slower than its predecessor scheduler. However, in practice, CFS scheduler achieves better performance and interactivity over its predecessor. The reason behind the improvement is due to the fact it does not need to perform any heuristic calculation which is the problem of the old scheduler. Let us recall the implementation of the previous scheduler version. The old scheduler works on two priority arrays which have their size equal to *MAX_PRIO*. One priority array is being active array which contains all active tasks. The other one is expired array which holds all expired tasks. A task becomes expired if it runs out of its allocated timeslice. Once active array has no task, the scheduler performs swapping two arrays which makes the active array becomes expired array and vice versa. The key of *O(1)* running time of this scheduler is the fact of using priority array with size=*MAX_PRIO* and swapping expired array with active array when in need.

The bottom line of this scheduler is the overhead when calculating in priority and timeslice of a process. The achievement of fairness and process interactivity lies in the values of priority and timeslice. The idea is the scheduler must accurately define a process whether it is an I/O bound or a CPU bound. Since I/O bound processes do not hog CPU, hence, they seem to execute fast. Delaying I/O bound process might cause unresponsive experience to users. Based on this fact, the scheduler favors I/O bound processes over CPU bound processes. The question is how scheduler can determine if a process is I/O bound or not. The solution is to perform some heuristic calculation (implemented in *effective_prio()*) based on task's static priority and the amount of sleeping time versus running time. This calculation is then scaled down to some number which is set to task's dynamic priority. The overhead in this scheduler is the work to maintain task's dynamic priority so that its value can reflect the task itself accurately.

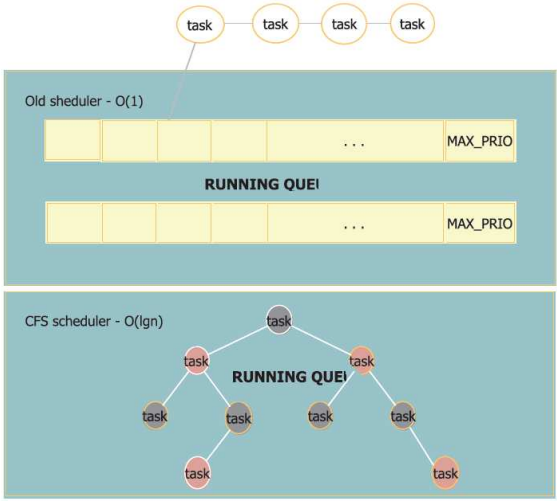


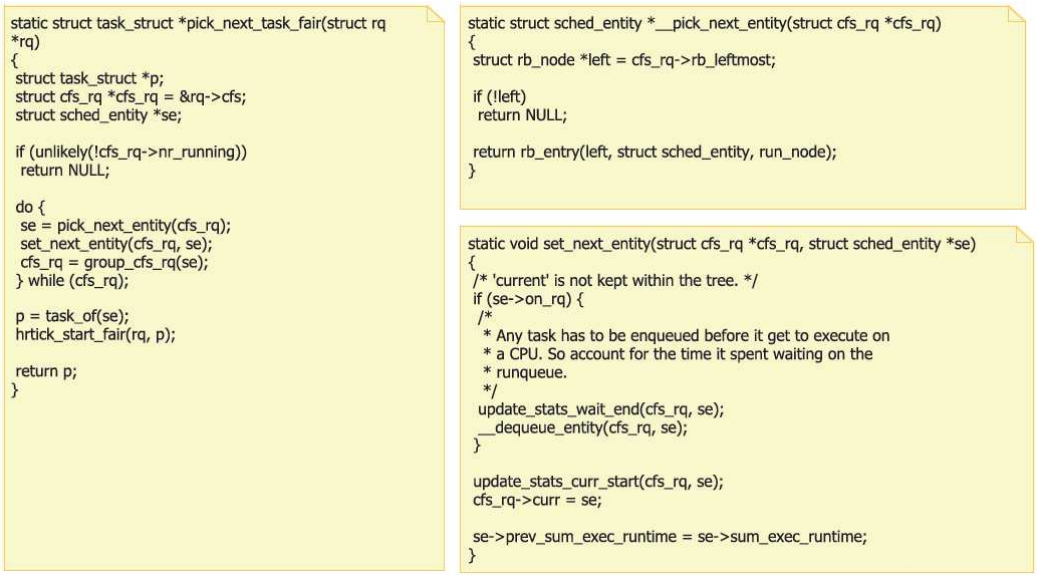
Figure.9 CFS vs The old scheduler

CFS scheduler was designed from the fresh approach which had nothing to do with any of priority matrix. CFS has its picking logic is based on this *schedule_entity.vruntime* value and it is thus very simple: it always tries to run the task with the smallest *p->se.vruntime* value (i.e., the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to "ideal multitasking hardware" as possible. For the internal data structure, CFS

scheduler uses Red-Black (RB) tree which is self-balanced tree. The use of RB tree in CFS has two advantages:

- It has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected).
- It takes for granted the feature of self-balanced of RB tree to reduce complexity in scheduler

CFS maintains a time-ordered RB tree, where all runnable tasks are sorted by the vruntime key (there is a subtraction using *rq->cfs.min_vruntime* to account for possible wraparounds). CFS picks the "leftmost" task from this tree and sticks to it. As the system progresses forwards, the executed tasks are put into the tree more and more to the right slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time.



```
static struct task_struct *pick_next_task_fair(struct rq
*rq)
{
 struct task_struct *p;
 struct cfs_rq *cfs_rq = &rq->cfs;
 struct sched_entity *se;

 if (unlikely(!cfs_rq->nr_running))
  return NULL;

 do {
  se = pick_next_entity(cfs_rq);
  set_next_entity(cfs_rq, se);
  cfs_rq = group_cfs_rq(se);
 } while (cfs_rq);

 p = task_of(se);
 hrtick_start_fair(rq, p);

 return p;
}
```

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
 struct rb_node *left = cfs_rq->rb_leftmost;

 if (!left)
  return NULL;

 return rb_entry(left, struct sched_entity, run_node);
}
```

```
static void set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
 /* 'current' is not kept within the tree. */
 if (se->on_rq) {
  /*
   * Any task has to be enqueued before it get to execute on
   * a CPU. So account for the time it spent waiting on the
   * runqueue.
   */
  update_stats_wait_end(cfs_rq, se);
  __dequeue_entity(cfs_rq, se);
 }

 update_stats_curr_start(cfs_rq, se);
 cfs_rq->curr = se;

 se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

Figure.10 CFS pick_next_task()

The implementation of *pick_next_task_fair()* in CFS schedule class is quite simple thanks to RT tree data structure.

Step 1: checks whether there is any running task on *cfs_rq*.
Step 2: calls *pick_next_entity()* which implicitly call *rb_entry()* to retrieve the next task having smallest value *vruntime*.
Step 3: *sets_next_entity()* performs update values in *sched_entity* of retrieved node from step 2.
Step 4: calls *group_cfs_rq()* to return owned group of the next task. The grouping enhancement was introduced in kernel 2.6.24 to achieve fairness to users or groups rather than just for tasks. The grouping scheduling can be enabled if *CONFIG_FAIR_GROUP_SCHED* is selected during the kernel build. If there is owned group for this task, it performs looping from step 2.

4. CONCLUSION

The study covered most important aspects of Linux scheduler 2.6.32. Kernel scheduler is one of the most frequently executed components in Linux system. Hence, it has gained a lot of attentions from kernel developers who have thrived to put the most optimized algorithms and codes into the scheduler. Different algorithms used in kernel scheduler were discussed in the study. CFS scheduler achieves a good performance and responsiveness while being relatively simple compared with the previous algorithm. CFS exceeds performance expectation in some workloads. But it still shows some weakness in other workloads. There are some complaints about irresponsiveness of CFS scheduler in 3D game area. It is difficult to make one such scheduler for all performance purposes. There are some discussions around the idea of allowing more than one scheduler available to user space in Linux system. At the point of writing this study, there is a new rival *Brain Fuck Scheduler (BFS)* which claimed to achieve a better performance than the current CFS scheduler while being much simpler.

5. APPENDIX

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
        /* runqueue lock: */
        spinlock_t lock;

        /*
         * nr_running and cpu_load should be in the same cacheline because
         * remote CPUs use both these fields when doing load calculation.
         */
        unsigned long nr_running;
        #define CPU_LOAD_IDX_MAX 5
        unsigned long cpu_load[CPU_LOAD_IDX_MAX];
#ifdef CONFIG_NO_HZ
        unsigned long last_tick_seen;
        unsigned char in_nohz_recently;
#endif
        /* capture load from *all* tasks on this cpu: */
        struct load_weight load;
        unsigned long nr_load_updates;
        u64 nr_switches;
        u64 nr_migrations_in;

        struct cfs_rq cfs;
        struct rt_rq rt;

#ifdef CONFIG_FAIR_GROUP_SCHED
        /* list of leaf cfs_rq on this cpu: */
        struct list_head leaf_cfs_rq_list;
#endif
#ifdef CONFIG_RT_GROUP_SCHED
        struct list_head leaf_rt_rq_list;
#endif

        /*
         * This is part of a global counter where only the total sum
         * over all CPUs matters. A task can increase this counter on
         * one CPU and if it got migrated afterwards it may decrease
         * it on another CPU. Always updated under the runqueue lock:
         */
        unsigned long nr_uninterruptible;

        struct task_struct *curr, *idle;
        unsigned long next_balance;
        struct mm_struct *prev_mm;

        u64 clock;

        atomic_t nr_iowait;

#ifdef CONFIG_SMP
        struct root_domain *rd;
        struct sched_domain *sd;

        unsigned char idle_at_tick;
        /* For active balancing */
        int post_schedule;
        int active_balance;
        int push_cpu;
        /* cpu of this runqueue: */
        int cpu;
        int online;
```

```
        unsigned long avg_load_per_task;

        struct task_struct *migration_thread;
        struct list_head migration_queue;

        u64 rt_avg;
        u64 age_stamp;
#endif

        /* calc_load related fields */
        unsigned long calc_load_update;
        long calc_load_active;

#ifdef CONFIG_SCHED_HRTICK
#ifdef CONFIG_SMP
        int hrtick_csd_pending;
        struct call_single_data hrtick_csd;
#endif
        struct hrtimer hrtick_timer;
#endif

#ifdef CONFIG_SCHEDSTATS
        /* latency stats */
        struct sched_info rq_sched_info;
        unsigned long long rq_cpu_time;
        /* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */

        /* sys_sched_yield() stats */
        unsigned int yld_count;

        /* schedule() stats */
        unsigned int sched_switch;
        unsigned int sched_count;
        unsigned int sched_goidle;

        /* try_to_wake_up() stats */
        unsigned int ttwu_count;
        unsigned int ttwu_local;

        /* BKL stats */
        unsigned int bkl_count;
#endif
};

struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
        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 (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
        int  (*select_task_rq)(struct task_struct *p, int sd_flag, int flags);

        unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
                        struct rq *busiest, unsigned long max_load_move,
                        struct sched_domain *sd, enum cpu_idle_type idle,
                        int *all_pinned, int *this_best_prio);

        int (*move_one_task) (struct rq *this_rq, int this_cpu,
                              struct rq *busiest, struct sched_domain *sd,
                              enum cpu_idle_type idle);
        void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
        void (*post_schedule) (struct rq *this_rq);
        void (*task_wake_up) (struct rq *this_rq, struct task_struct *task);

        void (*set_cpus_allowed)(struct task_struct *p,
                                 const struct cpumask *newmask);
```

```
        void (*rq_online)(struct rq *rq);
        void (*rq_offline)(struct rq *rq);
#endif

        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
        void (*task_new) (struct rq *rq, struct task_struct *p);

        void (*switched_from) (struct rq *this_rq, struct task_struct *task,
                               int running);
        void (*switched_to) (struct rq *this_rq, struct task_struct *task,
                             int running);
        void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                              int oldprio, int running);

        unsigned int (*get_rr_interval) (struct task_struct *task);

#ifdef CONFIG_FAIR_GROUP_SCHED
        void (*moved_group) (struct task_struct *p);
#endif
};


/*
 * All the scheduling class methods:
 */
static const struct sched_class fair_sched_class = {
        .next                   = &idle_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,
        .put_prev_task          = put_prev_task_fair,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_fair,

        .load_balance           = load_balance_fair,
        .move_one_task          = move_one_task_fair,
#endif

        .set_curr_task          = set_curr_task_fair,
        .task_tick              = task_tick_fair,
        .task_new               = task_new_fair,

        .prio_changed           = prio_changed_fair,
        .switched_to            = switched_to_fair,

        .get_rr_interval        = get_rr_interval_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
        .moved_group            = moved_group_fair,
#endif
};

static const struct sched_class rt_sched_class = {
        .next                   = &fair_sched_class,
        .enqueue_task           = enqueue_task_rt,
        .dequeue_task           = dequeue_task_rt,
        .yield_task             = yield_task_rt,

        .check_preempt_curr     = check_preempt_curr_rt,

        .pick_next_task         = pick_next_task_rt,
        .put_prev_task          = put_prev_task_rt,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_rt,
```

```
	.load_balance		= load_balance_rt,
	.move_one_task		= move_one_task_rt,
	.set_cpus_allowed	= set_cpus_allowed_rt,
	.rq_online		= rq_online_rt,
	.rq_offline		= rq_offline_rt,
	.pre_schedule		= pre_schedule_rt,
	.post_schedule		= post_schedule_rt,
	.task_wake_up		= task_wake_up_rt,
	.switched_from		= switched_from_rt,
#endif

	.set_curr_task		= set_curr_task_rt,
	.task_tick		= task_tick_rt,

	.get_rr_interval	= get_rr_interval_rt,

	.prio_changed		= prio_changed_rt,
	.switched_to		= switched_to_rt,
};


/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
	struct task_struct *prev, *next;
	unsigned long *switch_count;
	struct rq *rq;
	int cpu;

need_resched:
	preempt_disable();
	cpu = smp_processor_id();
	rq = cpu_rq(cpu);//get run queue of CPU
	rcu_sched_qs(cpu);//some lock mechaism
	prev = rq->curr;//current task_struct
	switch_count = &prev->nivcsw;

	release_kernel_lock(prev);
need_resched_nonpreemptible:

	schedule_debug(prev);

	//If we are in HRTICK, clear it
	if (sched_feat(HRTICK))
		hrtick_clear(rq);

	spin_lock_irq(&rq->lock);
	update_rq_clock(rq);
	clear_tsk_need_resched(prev);

	if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
		//what is it?
		if (unlikely(signal_pending_state(prev->state, prev)))
			prev->state = TASK_RUNNING;
		else
			deactivate_task(rq, prev, 1);
		switch_count = &prev->nvcsw;
	}

	pre_schedule(rq, prev);

	if (unlikely(!rq->nr_running))
		idle_balance(cpu, rq);

	put_prev_task(rq, prev);
	next = pick_next_task(rq);

	if (likely(prev != next)) {
		sched_info_switch(prev, next);
		perf_event_task_sched_out(prev, next, cpu);
```

```
                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;

                context_switch(rq, prev, next); /* unlocks the rq */
                /*
                 * the context switch might have flipped the stack from under
                 * us, hence refresh the local variables.
                 */
                cpu = smp_processor_id();
                rq = cpu_rq(cpu);
        } else
                spin_unlock_irq(&rq->lock);

        post_schedule(rq);

        if (unlikely(reacquire_kernel_lock(current) < 0))
                goto need_resched_nonpreemptible;

        preempt_enable_no_resched();
        if (need_resched())
                goto need_resched;
}
EXPORT_SYMBOL(schedule);
/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline void
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
        struct mm_struct *mm, *oldmm;

        prepare_task_switch(rq, prev, next);
        trace_sched_switch(rq, prev, next);
        mm = next->mm;
        oldmm = prev->active_mm;
        /*
         * 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);

        if (unlikely(!mm)) {
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next);
        } else
                switch_mm(oldmm, mm, next);

        if (unlikely(!prev->mm)) {
                prev->active_mm = NULL;
                rq->prev_mm = oldmm;
        }
        /*
         * Since the runqueue lock will be released by the next
         * task (which is an invalid locking op but in the case
         * of the scheduler it's an obvious special-case), so we
         * do an early lockdep release here:
         */
#ifndef __ARCH_WANT_UNLOCKED_CTXSW
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
#endif

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

        barrier();
        /*
         * this_rq must be evaluated again because prev may have moved
         * CPUs since it called schedule(), thus the 'rq' on its stack
```

```
		 * frame will be invalid.
		 */
		finish_task_switch(this_rq(), prev);
}

/*
 * fork()/clone()-time setup:
 */
void sched_fork(struct task_struct *p, int clone_flags)
{
		int cpu = get_cpu();

		__sched_fork(p);

		/*
		 * Revert to default priority/policy on fork if requested.
		 */
		if (unlikely(p->sched_reset_on_fork)) {
				if (p->policy == SCHED_FIFO || p->policy == SCHED_RR) {
						p->policy = SCHED_NORMAL;
						p->normal_prio = p->static_prio;
				}

				if (PRIO_TO_NICE(p->static_prio) < 0) {
						p->static_prio = NICE_TO_PRIO(0);
						p->normal_prio = p->static_prio;
						set_load_weight(p);
				}

				/*
				 * We don't need the reset flag anymore after the fork. It has
				 * fulfilled its duty:
				 */
				p->sched_reset_on_fork = 0;
		}

		/*
		 * Make sure we do not leak PI boosting priority to the child.
		 */
		p->prio = current->normal_prio;

		if (!rt_prio(p->prio))
				p->sched_class = &fair_sched_class;

#ifdef CONFIG_SMP
		cpu = p->sched_class->select_task_rq(p, SD_BALANCE_FORK, 0);
#endif
		set_task_cpu(p, cpu);

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
		if (likely(sched_info_on()))
				memset(&p->sched_info, 0, sizeof(p->sched_info));
#endif
#if defined(CONFIG_SMP) && defined(__ARCH_WANT_UNLOCKED_CTXSW)
		p->oncpu = 0;
#endif
#ifdef CONFIG_PREEMPT
		/* Want to start with kernel preemption disabled. */
		task_thread_info(p)->preempt_count = 1;
#endif
		plist_node_init(&p->pushable_tasks, MAX_PRIO);

		put_cpu();
}


/*
 * Check this_cpu to ensure it is balanced within domain. Attempt to move
 * tasks if there is an imbalance.
 */
static int load_balance(int this_cpu, struct rq *this_rq,
					struct sched_domain *sd, enum cpu_idle_type idle,
					int *balance)
```

```
{
        int ld_moved, all_pinned = 0, active_balance = 0, sd_idle = 0;
        struct sched_group *group;
        unsigned long imbalance;
        struct rq *busiest;
        unsigned long flags;
        struct cpumask *cpus = __get_cpu_var(load_balance_tmpmask);

        cpumask_setall(cpus);

        /*
         * When power savings policy is enabled for the parent domain, idle
         * sibling can pick up load irrespective of busy siblings. In this case,
         * let the state of idle sibling percolate up as CPU_IDLE, instead of
         * portraying it as CPU_NOT_IDLE.
         */
        if (idle != CPU_NOT_IDLE && sd->flags & SD_SHARE_CPUPOWER &&
            !test_sd_parent(sd, SD_POWERSAVINGS_BALANCE))
                sd_idle = 1;

        schedstat_inc(sd, lb_count[idle]);

redo:
        update_shares(sd);
        group = find_busiest_group(sd, this_cpu, &imbalance, idle, &sd_idle,
                                   cpus, balance);

        if (*balance == 0)
                goto out_balanced;

        if (!group) {
                schedstat_inc(sd, lb_nobusyg[idle]);
                goto out_balanced;
        }

        busiest = find_busiest_queue(group, idle, imbalance, cpus);
        if (!busiest) {
                schedstat_inc(sd, lb_nobusyq[idle]);
                goto out_balanced;
        }

        BUG_ON(busiest == this_rq);

        schedstat_add(sd, lb_imbalance[idle], imbalance);

        ld_moved = 0;
        if (busiest->nr_running > 1) {
                /*
                 * Attempt to move tasks. If find_busiest_group has found
                 * an imbalance but busiest->nr_running <= 1, the group is
                 * still unbalanced. ld_moved simply stays zero, so it is
                 * correctly treated as an imbalance.
                 */
                local_irq_save(flags);
                double_rq_lock(this_rq, busiest);
                ld_moved = move_tasks(this_rq, this_cpu, busiest,
                                      imbalance, sd, idle, &all_pinned);
                double_rq_unlock(this_rq, busiest);
                local_irq_restore(flags);

                /*
                 * some other cpu did the load balance for us.
                 */
                if (ld_moved && this_cpu != smp_processor_id())
                        resched_cpu(this_cpu);

                /* All tasks on this runqueue were pinned by CPU affinity */
                if (unlikely(all_pinned)) {
                        cpumask_clear_cpu(cpu_of(busiest), cpus);
                        if (!cpumask_empty(cpus))
                                goto redo;
                        goto out_balanced;
                }
```

```
		}

		if (!ld_moved) {
			schedstat_inc(sd, lb_failed[idle]);
			sd->nr_balance_failed++;

			if (unlikely(sd->nr_balance_failed > sd->cache_nice_tries+2)) {

				spin_lock_irqsave(&busiest->lock, flags);

				/* don't kick the migration_thread, if the curr
				 * task on busiest cpu can't be moved to this_cpu
				 */
				if (!cpumask_test_cpu(this_cpu,
						&busiest->curr->cpus_allowed)) {
					spin_unlock_irqrestore(&busiest->lock, flags);
					all_pinned = 1;
					goto out_one_pinned;
				}

				if (!busiest->active_balance) {
					busiest->active_balance = 1;
					busiest->push_cpu = this_cpu;
					active_balance = 1;
				}
				spin_unlock_irqrestore(&busiest->lock, flags);
				if (active_balance)
					wake_up_process(busiest->migration_thread);

				/*
				 * We've kicked active balancing, reset the failure
				 * counter.
				 */
				sd->nr_balance_failed = sd->cache_nice_tries+1;
			}
		} else
			sd->nr_balance_failed = 0;

		if (likely(!active_balance)) {
			/* We were unbalanced, so reset the balancing interval */
			sd->balance_interval = sd->min_interval;
		} else {
			/*
			 * If we've begun active balancing, start to back off. This
			 * case may not be covered by the all_pinned logic if there
			 * is only 1 task on the busy runqueue (because we don't call
			 * move_tasks).
			 */
			if (sd->balance_interval < sd->max_interval)
				sd->balance_interval *= 2;
		}

		if (!ld_moved && !sd_idle && sd->flags & SD_SHARE_CPUPOWER &&
		    !test_sd_parent(sd, SD_POWERSAVINGS_BALANCE))
			ld_moved = -1;

		goto out;

	out_balanced:
		schedstat_inc(sd, lb_balanced[idle]);

		sd->nr_balance_failed = 0;

	out_one_pinned:
		/* tune up the balancing interval */
		if ((all_pinned && sd->balance_interval < MAX_PINNED_INTERVAL) ||
				(sd->balance_interval < sd->max_interval))
			sd->balance_interval *= 2;

		if (!sd_idle && sd->flags & SD_SHARE_CPUPOWER &&
		    !test_sd_parent(sd, SD_POWERSAVINGS_BALANCE))
			ld_moved = -1;
		else
```

```
                ld_moved = 0;
out:
        if (ld_moved)
                update_shares(sd);
        return ld_moved;
}
```