Scheduling Policies and Process Management System calls in the Linux kernel

September 10, 2017

Group members:
Mallikarjuna(CS15B010),
Tejavardhan Reddy(CS15B014),
G. Pranav(CS15B015),
Giridhar S(CS15B016),
K. Mithilesh(CS15B020),
Lokesh K(CS15B049)

1 Introduction

In the following report, we analyse the CFS scheduler, which is the scheduler that is used in most Linux kernels today. We also talk about the various process management system calls that the Linux kernel allows us to make.

The task of a scheduler is to share CPU time between the programs to create the illusion of concurrent execution. Unlike the classical schedulers where the time slices for each process are computed and the processes are made to execute till these time slices are exhausted, the current scheduler considers only the wait time of the process- that is, how long it has been in the run-queue. Every time the scheduler is called, it picks the task with the highest waiting time and gives the CPU to it. The scheduler keeps track of the waiting time of the available processes by sorting them in a red-black tree.

2 Linux Calls to duplicate a Process

1. Fork
   (a) Heavy Weight call because creates full copy of parent process as child process.
   (b) To reduce effort in this call linux uses Copy-on-write technique.
2. vfork
   (a) Similar to fork,but doesn't create a copy of the data of the parent process.
   (b) Shares data between child and parent.

   (c) Designed for situation in which child process immediately calls execve system call to load new program.
   (d) Kernel ensures parent is blocked until child process exits or new progam starts.

3. clone
   (a) generates threads and enables decision to specify which elements are shared and which are copied.

3 Copy on write technique in fork call

1. Only page tables are copied to the child process.
2. These page tables are only readable from now, and a new page for it is created if process writes.

4 Executing System calls

1. The entry points are sysfork, sysvfork, and sysclone functions.
   (a) Their definitions are architecture-dependent because the way in which parameters are passed between userspace and kernel space differs on the various architectures.
   (b) The task of the above functions is to extract the information supplied by userspace from the registers of the processors and then to invoke the architecture independent dofork function responsible for process duplication.
2. kernel/fork.c
   (a)
   ```
   long do_fork(unsigned long clone_flags,
   unsigned long stack_start,
   struct pt_regs *regs,
   unsigned long stack_size,
   int __user *parent_tidptr,
   int __user *child_tidptr)
   ```
   (b) stack-start : The start address of the user mode stack to be used.
   (c) regs : A pointer to the register set holding the call parameters in raw form
   (d) stack-size : The size of the user mode stack ,usually unnecessary and set to 0.
   (e) Two pointers to addresses in userspace { parent-tidptr and child-tidptr} that hold the TIDs of the parent and child processes. They are needed for the thread implementation of the NPTL {Native Posix Threads Library} library.
   (f) A flag set { clone_flags } to specify duplication properties. The low byte specifies the signal number to be sent to the parent process when the child process terminates.

3. Call implementations

   (a) `arch/x86/kernel/process_32.c`
   ```
   asmlinkage int sys_fork(struct pt_regs regs)
   {
   return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
   }
   ```

   (b) `arch/x86/kernel/process_32.c`
   ```
   asmlinkage int sys_clone(struct pt_regs regs)
   {
   unsigned long clone_flags;
   unsigned long newsp;
   int __user *parent_tidptr, *child_tidptr;
   clone_flags = regs.ebx;
   newsp = regs.ecx;
   parent_tidptr = (int __user *)regs.edx;
   child_tidptr = (int __user *)regs.edi;
   if (!newsp)
   newsp = regs.esp;
   return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
   }
   ```

5 Functions of dofork

1. copy_process
2. Determine PID
3. Initialize vfork completion handler (only with CLONE_VFORK) and ptrace flags
4. wake_up_new_task
5. clone_vfork set - wait_for_completion

6 Copying processes

1. Check flags
2. dup_task_struct : create new task srtuct
3. Check resource limits
4. initialize task structure
5. sched_fork : enabling for the process to schedule
6. Copy/share process components
7. Set IDs, task relationships, etc.

8. The task structures for parent and child differ only in one element: A new kernel mode stack is allocated for the new process. A pointer to it is stored in task_struct-stack . Usually the stack is stored in a union with thread_info , which holds all required processor-specific low-level information about the thread.

9.
```
<sched.h>
union thread_union {
struct thread_info thread_info;
unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

10. Thread_info holds process data that needs to be accessed by the architecture-specific assembly language code. Although the structure is defined differently from processor to processor, its contents are similar to the following on most systems.

11.
```
<asm-arch/thread_info.h>
struct thread_info {
struct task_struct *task;           /*main task structure */
struct exec_domain *exec_domain;    /*execution domain */
unsigned long flags;                 /*low level flags */
unsigned long status;               /*thread-synchronous flags */
__u32 cpu;                         /*current CPU */
int preempt_count;                  /*0 => preemptable, <0 => BUG */
mm_segment_t addr_limit;    /* thread address space */
struct restart_block restart_block;
```

12. The new process must then be linked with its parent process by means of the children list. This is handled by the auxiliary macro add_parent . Besides, the new process must be included in the ID data structure network

7 Kernel Threads

Kernel threads are processes started directly by the kernel itself. They delegate a kernel function to a separate process and execute it there in "parallel" to the other processes in the system

Basically, there are two types of kernel thread:

1. **Type 1**
   The thread is started and waits until requested by the kernel to perform a specific action

2. **Type 2**
   Once started, the thread runs at periodic intervals, checks the utilization of a specific resource, and takes action when utilization exceeds or falls below a set limit value.

The kernel-thread function is invoked to start a kernel thread.

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

Because kernel threads are generated by the kernel itself, two special points should be noted:

1. They execute in the supervisor mode of the CPU, not in the user mode
2. Type 2 They may access only the kernel part of virtual address space (all addresses above (`TASK_SIZE`) but not the virtual user area.

   Every time the kernel performs a context switch, the userland portion of the virtual address space must be replaced to match the then-running process.

   Two pointers to `mm_struct` s are contained in the task structure:

   ```
       struct task_struct {
   ...
   struct mm_struct *mm, *active_mm;
     ...}
   ```

   This can be optimized in the folowing way using
   LazyTLB handling:
   Since kernel threads are not associated with any particular userland process, the kernel does not need to rearrange the userland portion of the virtual address space and can just leave the old setting in place. Since any userland process can have been running before a kernel thread, the contents of the userspace part are essentially random, and the kernel thread must not modify it. To signalize that the userspace portion must not be accessed, mm is set to a NULL pointer.

   A kernel thread can be implemented in one of two ways.
   The older variant — which is still in use in some places in the kernel — is to pass a function directly to `kernel_thread` . The function is then responsible to assist the kernel in the transformation into a daemon by invoking daemonize
   The more modern possibility to create a kernel thread is the auxiliary function `kthread_create`.

   ```
       struct task_struct *kthread_create(int (*threadfn)(void *data),
       void *data,const char,
       namefmt[],...)
   ```

   Kernel threads appear in the system process list but are enclosed in square brackets in the output of `ps` to differentiate them from normal processes

8 Starting New Programs

New programs are started by replacing an existing program with new code. Linux provides the execve system call for this purpose

The entry point of the system call is the architecture-dependent `sys_execve` function. This function quickly delegates its work to the system-independent `do_execve` routine

```
int do_execve(char * filename,
    char __user *__user *argv,
    char __user *__user *envp,
struct pt_regs * regs)
```

Not only the register set with the arguments and the name of the executable file (filename) but also pointers to the arguments and the environment of the program are passed as in system programming.
`argv` and `envp` are arrays of pointers, and both the pointer to the array itself as well as all pointers in the array are located in the userspace portion of the virtual address space .
Precautions are required when userspace memory is accessed from the kernel, and that the `_user` annotations allow automated tools to check if everything is handled properly or not.
Code Flow

1. First, the file to be executed is opened.

2. `bprm_init` then handles several administrative tasks: `mm_alloc` generates a new instance of `mm_struct` to manage the process address space (see Chapter 4). `init_new_context` is an architecture-specific function that initializes the instance, and `_bprm_mm_init` sets up an initial stack.

3. Various parameters like euid, egid, argument list are combined into a structure of type `linux_binprm` ,the remaining data — the argument list — are then copied manually into the structure

```
    int prepare_binprm(struct linux_binprm *bprm)
{
...
bprm->e_uid = current->euid;
bprm->e_gid = current->egid;
if(!(bprm->file->f_vfsmnt->mnt_flags & MNT_NOSUID)) {
/* Set-uid? */
if (mode & S_ISUID) {
bprm->e_uid = inode->i_uid;
}
/* Set-gid? */
/*
* If setgid is set but no group execute bit then this
* is a candidate for mandatory locking, not a setgid
* executable.
*/
if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
bprm->e_gid = inode->i_gid;
}
}
...
}
```

4. After making sure that MNT_NOSUID is not set for the mount from which the file originates, the kernel checks if the SUID or SGID bit is set. The first case is simple to handle: If S_ISUID is set, then the effective UID gets the same value as the inode (otherwise, the process's effective UID is used). The SGID case is similar, but the kernel must additionally make sure that the execute bit is also set for the group.

   Linux supports various organization formats for executable files. The standard format is ELF (Executable and Linkable Format)

   Even though many binary formats can be used on different architectures (ELF was designed explicitly to be as system-independent as possible), this does not mean that programs in a specific binary format are able to run on multiple architectures.

   search_binary_handler is used at the end of do_execve to find a suitable binary format for the particular file.

   Each binary format is represented in the Linux kernel by an instance of the following (simplified) data structure:

```
    struct linux_binfmt {
struct linux_binfmt * next;
struct module *module;
int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
int (*load_shlib)(struct file *);
int (*core_dump)(long signr, struct pt_regs * regs, struct file * file);
unsigned long min_coredump;
/* minimal dump size */
};
```

9 Exiting A Process

Processes must terminate with the exit system call. This gives the kernel the opportunity to free the resources used by the processes to the system. 19 The entry point for this call is the sys_exit function that requires an error code as its parameter in order to exit the process. Its definition is architecture-independent and is held in kernel/exit.c

10 Data Structures

10.1 Task Structure

The kernel supports different scheduling policies and scheduling classes allow us for implementing these policies in a modular way. Scheduling classes are used to decide which task runs next. Every task belongs to exactly one of the scheduling classes, and each scheduling class is responsible to manage their tasks.

```
struct task_struct {
....
int prio, static_prio, normal_prio;
unsigned int rt_priority;

struct list_head run_list;
const struct sched_class *sched_class;
struct sched_entity se;

unsigned int policy;
cpumask_t cpus_allowed;
unsigned int time_slice;
....
}
```

prio and normalprio indicate the dynamic priorities, staticprio the static priority of a process when it was started. normalpriority denotes a priority that is computed based on the static priority and the scheduling policy of the process. However, the priority considered by the scheduler is kept in prio. A staticpriority is required because situations can arise in which the kernel needs to temporarily boost the priority of a process. rtpriority denotes the priority of a real-time process. The scheduler is not limited to schedule processes, but can also work with larger entities and allows implementing group scheduling. This generality requires that the scheduler does not directly operate on processes but works with schedulable entities. An entity is represented by an instance of schedentity. schedclass denotes the scheduler class the process is in. policy holds the scheduling policy applied to the process.

10.2 Scheduler Classes

Scheduler classes provide the connection between the generic scheduler and individual scheduling methods. They are represented by several function pointers collected in a special data structure. This allows for creation of the generic scheduler without any knowledge about the internal working of different scheduler classes.

```
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);

struct task_struct * (*pick_next_task) (struct rq *rq);
void (*put_prev_task) (struct rq *rq, struct task_struct *p);

void (*set_curr_task) (struct rq *rq);
void (*task_tick) (struct rq *rq, struct task_struct *p);
void (*task_new) (struct rq *rq, struct task_struct *p);
}
```

The next element connects the schedclass instances of the different scheduling classes. Enqueuetask and dequeuetask are used to add and remove tasks to and from the run queue. When a process wants to relinquish control of the processor voluntarily, it can use the schedyield system call. This triggers yieldtask to be called in the kernel. checkpreemptcurr is used to preempt the current task with a newly woken task if this is necessary. picknexttask selects the next task that is supposed to run, while putprevtask is called before the currently executing task is replaced with another one. setcurrtask is called when the scheduling policy of a task is changed. tasktick is called by the periodic scheduler each time it is activated. newtask allows for setting up a connection between the fork system call and the scheduler. Each time a new task is created, the scheduler is knows about this with newtask.

10.3 Run queues

The central data structure of the core scheduler that is used to manage active processes is known as the run queue. Each CPU has its own run queue, and each active process appears on just one run queue. It is not possible to run a process on several CPUs at the same time. Run queues are implemented using the following data structure.

```
struct rq {
unsigned long nr_running;
#define CPU_LOAD_IDX_MAX 5
unsigned long cpu_load[CPU_LOAD_IDX_MAX];
...
struct load_weight load;
struct cfs_rq cfs;
struct rt_rq rt;
struct task_struct *curr, *idle;
u64 clock;
...
};
```

nrrunning specifies the number of runnable processes on the queue. Load provides a measure for the current load on the run queue. CPUload allows for tracking the load behavior back into the past. cfs and rt are the embedded sub-run queues for the completely fair and real-time scheduler, respectively. curr points to the task structure of the process currently running. idle points to the task structure of the idle process called when no other runnable process is available.

11 Representation of Priorities

The static priority of a process can be set in userspace by means of the nice command, which internally invokes the nice system call.The nice value of a process is between -20 and +19

The kernel uses a simpler scale ranging from 0 to 139 inclusive to represent priorities internally. Again, lower values mean higher priorities. The range from 0 to 99 is reserved for real-time processes. The nice values $[-20, +19]$ are mapped to the range from 100 to 139. Real−time processes thus always have a higher priority than normal processes can ever have.

12 Computing Priorities

We assume that static priority is already set and describe how to compute the dynamic and normal priorities.

This is done by the following code

```
#define  MAX_RT_PRIO  100

p->prio = effective_prio(p);

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

static inline int normal_prio(struct task_struct *p)
{
   int prio;

   if (task_has_rt_policy(p))
      prio = MAX_RT_PRIO-1 - p->rt_priority;
   else
      prio = __normal_prio(p);
      return prio;
}

static inline int __normal_prio(struct task_struct *p)
{
   return p->static_prio;
}
```

The above code is mostly self explanatory.

The above code assigns the priorities . Now the rt_prio function checks if the prio is in 0 to 99 and returns 1 if yes else 0

The function task_has_rt_policy is used in normal_prio instead of rt_prio because it is required for non-real-time tasks that have been temporarily boosted to a real-time priority, which can happen when RT-Mutexes are in use.

When a process forks off a child, the current static priority will be inherited from the parent. The dynamic priority of the child, that is, task_struct->prio, is set to the normal priority of the parent. This ensures that priority boosts caused by RT-Mutexes are not transferred to the child process.

13 Computing Load Weights

The importance of a task is not only specified by its priority, but also by the load weight stored in task_struct->se.load.set_load_weight is responsible to compute the load weight depending on the process type and its static priority.

The general idea is that every process that changes the priority by one nice level down gets 10 percent more CPU power, while changing one nice level up gives 10 percent CPU power less. To enforce this policy, the kernel converts priorities to weight values.

Base case : nice level zero is given a weight of 1024 . And the rest are adjusted as per above criterion with a multiplier of 1.25. Notice the exponential behaviour of the weights wiht nice values.

While performing the conversion we also needs to account for real-time tasks. These will get double of the weight of a normal task. SCHED_IDLE tasks, on the other hand, will always receive a very small weight.

13.1 Code

```
#define WEIGHT_IDLEPRIO 2
#define WMULT_IDLEPRIO (1 << 31)

static void set_load_weight(struct task_struct *p)
{
   if (task_has_rt_policy(p)) {
      p->se.load.weight = prio_to_weight[0] * 2;
      p->se.load.inv_weight = prio_to_wmult[0] >> 1;
      return;
   }
   /*
   * SCHED_IDLE tasks get minimal weight:
   */
   if (p->policy == SCHED_IDLE) {
      p->se.load.weight = WEIGHT_IDLEPRIO;
      p->se.load.inv_weight = WMULT_IDLEPRIO;
      return;
   }

   p->se.load.weight = prio_to_weight[p->static_prio - MAX_RT_PRIO];
   p->se.load.inv_weight = prio_to_wmult[p->static_prio - MAX_RT_PRIO];
}
```

Every time a process is added or deleted or updated in run queue we need to appropriately change the data associated with number of processes in run queue and also weight of the queue

14 The Completely Fair Scheduler

14.1 fair_sched_class

All information that the core scheduler needs to know about the completely fair scheduler is contained in **fair_sched_class**, which can be found inside **kernel/sched/fair.c**, in the mool kernel source code :

```
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,
    .yield_to_task = yield_to_task_fair,

    .check_preempt_curr = check_preempt_wakeup,

    .pick_next_task = pick_next_task_fair,
    .put_prev_task = put_prev_task_fair,

...
    .set_curr_task          = set_curr_task_fair,
...
    .update_curr = update_curr_fair,
...
};
```

The meaning of each of these fields has already been described earlier.

14.2 The CFS Run Queue

```
struct cfs_rq {
    struct load_weight load;
    unsigned int nr_running;
    u64 exec_clock;
    u64 min_vruntime;
...
    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;

    struct sched_entity *curr, *next, *last, *skip;
...
};
```

Explanation of components:

1. *load:* Cumulative load values of all processes on queue
2. *nr_running:* Number of running processes on the queue.
3. *min_vruntime:* Minimum virtual run time of all processes on the queue. Forms the basis to implement the virtual clock associated with a run queue.
4. *tasks_timeline:* Base element to manage all processes in a time-ordered red-black tree.
5. *rb_leftmost:* The leftmost element of the tree, that is, the element that deserves to be scheduled most. Storing it explicitly causes speed up.
6. *curr:* Points to the schedulable entity of the currently executing process

14.3 CFS Operations: Virtual Clock

The CFS does not store the virtual time on any seperate data structure, but insteads calculates it on the basis of ther parameters, using the **update_curr** method.

```
-----------
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;
```

If no process is currently executing on the run queue, there is nothing to do, so the function returns.

```
-----------
    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;
```

Otherwise, the kernel computes the time difference between the last update of the load statistics and now, and stores it in *delta_exec* variable. It further checks for the edge case where *delta_exec* takes a negative or zero value, in which case there is no need to proceed. If not, the *exec_start* of *cur* is updated to *now*, as that is the latest time of upload.

```
-----------
    schedstat_set(curr->statistics.exec_max,
      max(delta_exec, curr->statistics.exec_max));
    curr->sum_exec_runtime += delta_exec;
```

Now the function has to update the physical and virtual time that the current process has spent executing on the CPU. For the physical time, the time difference just needs to be added to the previously accounted time.

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

The non-existing virtual clock is emulated using the given information. The virtual time must be weighted according to the load weight of the process. What calc_delta_fair does is to compute the value given by the following formula:

$delta\_exec\_weighted = delta\_exec$ x $\frac{NICE\_0\_LOAD}{curr\_load\_weight}$

The inverse weight values mentioned above mean that more important tasks with higher priorities (i.e., lower nice values) will get larger weights, so the virtual run time accounted to them will be smaller.

Finally, by calling *update_min_vruntime* the function updates the *min_vruntime* in a monotonically increasing fashion.

14.4 Queue Manipulation:

Two functions are available to move elements to and from the run queue: enqueue_task_fair and dequeue_task_fair .

First, we consider enqueuing.

```
static void enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags){
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
    for_each_sched_entity(se) {
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, flags);

        ...
    }
    ...
hrtick_update(rq);
}
```

Besides pointers to the generic run queue and the task structure in question, the function accepts one more parameter: flags . This allows for specifying if the task that is enqueued has only recently been woken up and changed into the running state (flags is 1 in this case), or if it was runnable before (flags is 0 then). If the task is already on the run queue as signaled by the *on_rq* element of struct *sched_entity* , nothing needs to be done. Otherwise, the work is delegated to *enqueue_entity* , where the kernel updates the statistics with *update_curr* .

If the task has recently been running, its virtual run time is still valid, and (unless it is currently executing) it can be directly included into the red-black tree with __enqueue_entity. This function handles insertion to the red-black tree, and it relies on standard methods of the kernel. The process is placed at the proper position, this has already been ensured by setting the *vruntime* field of the process, and by the constant *min_vruntime* updates performed by the kernel for the queue.

If the process has been sleeping before, the virtual run time of the process is first adjusted in *place_entity* function. After *place_entity* has determined the proper virtual run time for the process, it is placed on the red-black tree with *__enqueue_entity.*

14.5 Selecting Next Task:

Selecting the next task to run is performed in pick_next_task_fair.

Note: Code not pasted here due to huge size.

If no tasks are currently runnable on the queue as indicated by an empty *nr_running* counter, the function handles it by going to the *idle* label. Otherwise, the work is delegated to *pick_next_entity* . If a leftmost task is available in the tree, it can immediately be determined and extracted from the red-black tree. Now the task has been selected, but some more work is required to mark it as the running task. This is handled by *set_next_entity* .The currently executing process is not kept on the run queue, so *__dequeue_entity* removes it from the red-black tree, setting the leftmost pointer to the next leftmost task if the current task has been the leftmost one. Although the process is not contained in the red-black tree anymore, the connection between process and run queue is not lost, because *curr* marks it as the running one now.

14.6 The Periodic Tick

vruntime is used to track the virtual runtime of runnable tasks in CFS' redblack-tree. The **scheduler_tick()** function of the scheduler skeleton regularly calls the **task_tick()** hook into CFS. This hook internally calls **task_tick_fair()** which is the entry point into the CFS task update :

```
/*
 * scheduler tick hitting a task of our scheduling class:
 */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
struct cfs_rq *cfs_rq;
struct sched_entity *se = &curr->se;

for_each_sched_entity(se) {
cfs_rq = cfs_rq_of(se);
entity_tick(cfs_rq, se, queued);
}

if (numabalancing_enabled)
task_tick_numa(rq, curr);

update_rq_runnable_avg(rq, 1);
}
```

task_tick_fair() calls **entity_tick()** for the tasks scheduling entity and corresponding runqueue.
The statistics updated using **update_curr**. If the **nr_running** counter ofthe queue indicates that fewer than two processes are runnable on the queue, nothing needs to be done.If a process is supposed to be preempted, there needs to be at least another one that could preempt it. Otherwise, the decision is left to **check_preempt_tick**.

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
unsigned long ideal_runtime, delta_exec;
struct sched_entity *se;
s64 delta;

ideal_runtime = sched_slice(cfs_rq, curr);
delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
if (delta_exec > ideal_runtime) {
resched_curr(rq_of(cfs_rq));
/*
 * The current task ran long enough, ensure it doesn't get
 * re-elected due to buddy favours.
 */
clear_buddies(cfs_rq, curr);
return;
}
```

The purpose of the function is to ensure that no process runs longer than specified by its share of the latency period. This length of this share in real-time is computed in **sched_slice**, and the realtime interval during which the process has been running on the CPU is given by **sum_exec_runtime - prev_sum_exec_runtime** .

14.7 Wake-up Preemption

When tasks are woken up in **try_to_wake_up** and **wake_up_new_task** , the kernel uses **check_preempt_curr** to see if the new task can preempt the currently running one. For completely fair handled tasks, the function **check_preempt_wakeup** performs the desired check.

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
struct task_struct *curr = rq->curr;
struct sched_entity *se = &curr->se, *pse = &p->se;
struct cfs_rq *cfs_rq = task_cfs_rq(curr);
int scale = cfs_rq->nr_running >= sched_nr_latency;
int next_buddy_marked = 0;
    .........
    .........
}
```

The newly woken task need not necessarily be handled by the completely fair scheduler. If the new task is a real-time task, rescheduling is immediately requested because real-time tasks always preempt CFS tasks.

15 Real Time Scheduling Class

Each process or Thread in system has associated scheduling policy and a static scheduling priority, sched_priority. The Process or Thread that has higher static scheduling priority (from 0-99) are handled by real time scheduling class of the scheduler.
The static_prio has the static priority of the process and the rt_task macro defines wheather the process is real time or not and task_has_rt_policy checks if the process is associated with a real-time scheduling policy.

15.1 Introduction

Real Time process differ from normal process in a essential way that if a Real time process exist in the system it will always be selected first unless there is another real time process with higher priority.

There are two scheduling policies for real time processes :

15.1.1 First In First Out (SCHED_FIFO)

In this policy the task which come first will be executed first and will be kept running till its end until it yields by calling sched_yield() or blocked for an I/O operation or some higher priority task preempts it.

It follows the following rules:

1. A SCHED_FIFO thread that has been preempted by another thread of higher priority will stay at the head of the list for its priority and will resume execution as soon as all threads of higher priority are blocked again.

2. When a SCHED_FIFO thread becomes runnable, it will be inserted at the end of the list for its priority

3.A thread calling sched_yield() will be put at the end of the list.

15.1.2 Round Robin Policy (SCHED_RR)

A process or task in round robin scheduling have a time slice whose value is reduced when they run if they are normal processes.Once all time quantums have expired, the value is reset to the initial value,but the process is placed at the end of the queue. This ensures that if there are several SCHED_RR processes with the same priority, they are always executed in turn.

15.2 Underlying Data structure

The scheduling class for real-time tasks is defined in **rt.c** inside the sched folder in the kernel directory of boss-mool linux.

Mostly there are analogs for the basic scheduling data structure but the implementations are much simpler then Complete Fair scheduling.
it is as follows :

```
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,

.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,

.update_curr = update_curr_rt,
};
```

The analog of **update_cur** for the real-time scheduler class is **update_curr_rt**. The function keeps track of the time the current process spent executing on the CPU in **sum_exec_runtime**. All calculations are performed with real times; virtual times are not required
In **rt_sched** the core run queue **rt_rq** defined in sched.h file also have sub run queue instance.

```
struct rt_rq {
struct rt_prio_array active;
unsigned int rt_nr_running;
......
struct rq *rq;
struct task_group *tg;
}
```

The run queue is very simple here just a Linked list.

All real-time tasks with the same priority are kept in a linked list headed by **active.queue**[**prio**], and the bitmap active.bitmap signals in which list tasks are present by a set bit. If no tasks are on the list,the bit is not set.
structured as below : (decalred in sched.h file)

```
/*
 * This is the priority-queue data structure of the RT scheduling class:
 */
struct rt_prio_array {
DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
struct list_head queue[MAX_RT_PRIO];
};
```

15.3 Operation And Features

- Enqueue and Dequeue :
  The task is placed or removed from appropriate list selected by $array->queue+p->prio$, and the corresponding bit in the bitmap is set if at least one task is present, or removed if no tasks are left on the queue.

- Picking Next Task :
  pick_next_task_rt handles selection of the next task first with **sched_find_first_bit** is a standard function that finds the first set bit in **active.bitmap** and this means that higher real-time priorities are handled before lower realtime priorities.The first task on the selected list is taken out, and **se.exec_start** is set to the current real-time clock value of the run queue.

- Converting to Real time process :
  sched_setscheduler system call is used to convert a normal process to a Real Time process it removes the process from its current queue using **deactivate_task** and sets real time priority for it in the scheduling class data structure and then re-activates the task.