Processes , threads and cpu scheduler

process and thread management gives rise to several issues that the kernel must resolve, the most important of which are listed below.

- applications must not interfere with each other unless this is expressly desired. For example, an error in application A must not be propagated to application B. Because Linux is a multiuser system, it must also be ensured that programs are not able to read or modify the memory contents of other programs — otherwise, it would be extremely easy to access the private data of other users.

- CPU time must be shared as fairly as possible between the various applications, whereby some programs are regarded as more important than others.

- all processes and threads must be consistently managed

we deal with the first requirement — memory protection — in the next set of slides. In the present slides, we focus my attention on the methods employed by the kernel to share CPU time and to switch between processes. Specifically, process and thread management data-structures are is seen detail to enable us use the process manager effectively.

Process data-structure

- all algorithms of the Linux kernel concerned with processes and programs are built around a data structure named task_struct and defined in include/linux/sched.h.

- this is one of the central structures in the system. before we move on to deal with the implementation of the scheduler, it is essential to examine how Linux manages processes.

- the task structure includes a large number of elements that link the process with the kernel subsystems which we discuss below.

- a part of it is shown below:

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned long flags;    /* per process flags, defined below */
    unsigned long ptrace;
    int lock_depth;         /* BKL lock depth */
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
```

```
        struct sched_entity se;
        unsigned short ioprio;
        unsigned long policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice;

        ..................

}
```

- more can be seen in the appropriate header

- let is discuss some of the elements that may be of interest to us

Process states

- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- EXIT_ZOMBIE
- EXIT_DEAD

- let us discuss their meanings

Different types of processes

- a classical Unix/Linux process is an application that consists of binary code, a chronological thread (the computer follows a single path through the code, no other paths run at the same time), and a set of resources allocated to the application — for example, memory, files, and so on.

New processes are generated using the fork and exec system calls:

- fork generates an identical copy of the current process; this copy is known as a child process. All resources of the original process are copied in a suitable way so that after the system call there are two independent instances of the original process. These instances are not linked in any way but have, for example, the same set of open files, the same working directory, the same data in memory (each with its own copy of the data), and so on.3

- exec replaces a running process with another application loaded from an executable binary file. In other words, a new program is loaded. Because exec does not create a new process, an old program must first be duplicated using fork, and then exec must be called to generate an additional application on the system.

- Linux also provides the clone system call in addition to the two calls above that are available in all Unix flavors and date back to very early days. In principle, clone works in the same way as fork, but the new process is not independent of its parent process and can share some resources with it. It is possible to specify which resources are to be shared and which are to be copied — for example, data in memory open files, or the installed signal handlers of the parent process.

- clone is used to implement threads. However, it is not called directly – threading Libraries are also needed in userspace to complete implementation. NPTL is the current library that is used in Linux – we will be seeing more on

----------------intentionally left blank------------------------

Process identification

- Linux processes are always assigned a number to uniquely identify them in their namespace. This number is called the process identification number or PID for short. Each process generated with fork or clone is automatically assigned a new unique PID value by the kernel.

- each process is, however, not only characterized by its PID but also by other identifiers. Several types are possible:

  - all processes in a thread group (i.e., different execution contexts of a process created by calling clone with CLONE_THREAD as we will see below) have a uniform thread group id (TGID). if a process does not use threads, its PID and TGID are identical.
    the main process in a thread group is called the group leader. The group_leader element of the task structures of all cloned threads points to the task_struct instance of the group leader.

  - otherwise, independent processes can be combined into a process group (using the setpgrp system call). The pgrp elements of their task structures all have the same value, namely, the PID of the process group leader. Process groups facilitate the sending of signals to all members of the group, which is helpful for various system programming applications (see the literature on system programming). Notice that processes connected with pipes are contained in a
    process group.

  - Several process groups can be combined in a session. All processes in a session have the same session ID which is held in the session element of the task structure. The SID can be set using the setsid system call. It is used in terminal programming but is of no particular relevance to us here.

Note : we will see the usefulness of the above in the upcoming discussions

---------------intentionally left blank---------------

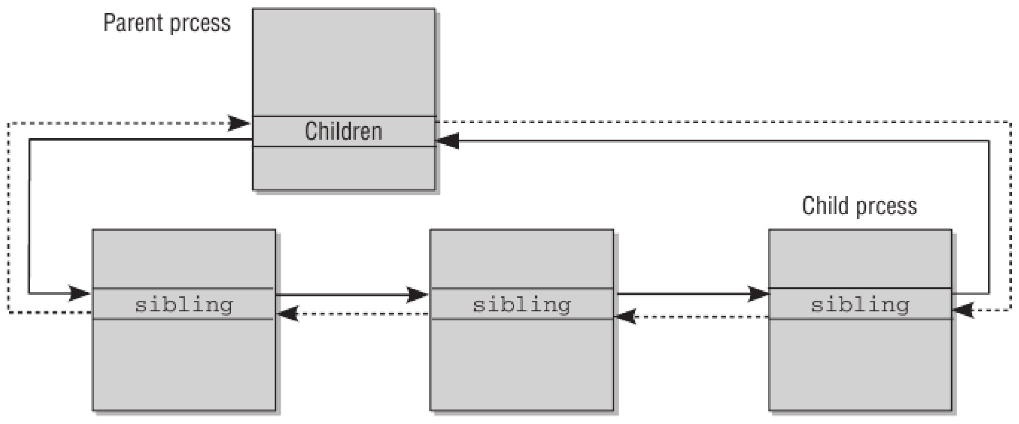
Process relationships

- the task_struct task data structure provides two list heads to help implement these relationships:

```
   <linux/sched.h>
   struct task_struct {
   ...
           struct list_head children; /* list of my children */
                 struct list_head sibling; /* linkage in my parent's children list */
...
}
```

- children is the list head for the list of all child elements of the process.
- siblings is used to link siblings with each other.

Illustration of process relationship



Process creation:

- may be done with fork(), vfork() and clone()
- all three end up calling do_fork()

Illustration of do_fork()

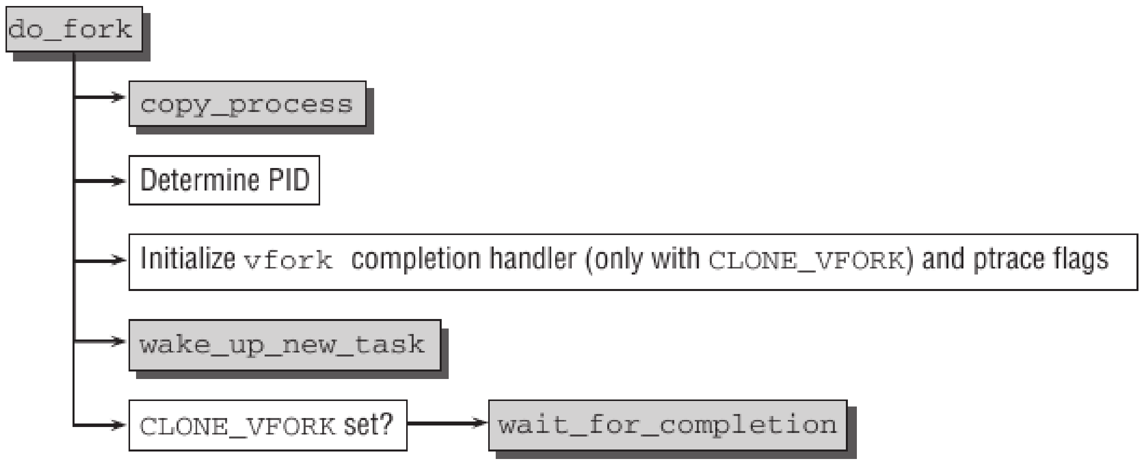
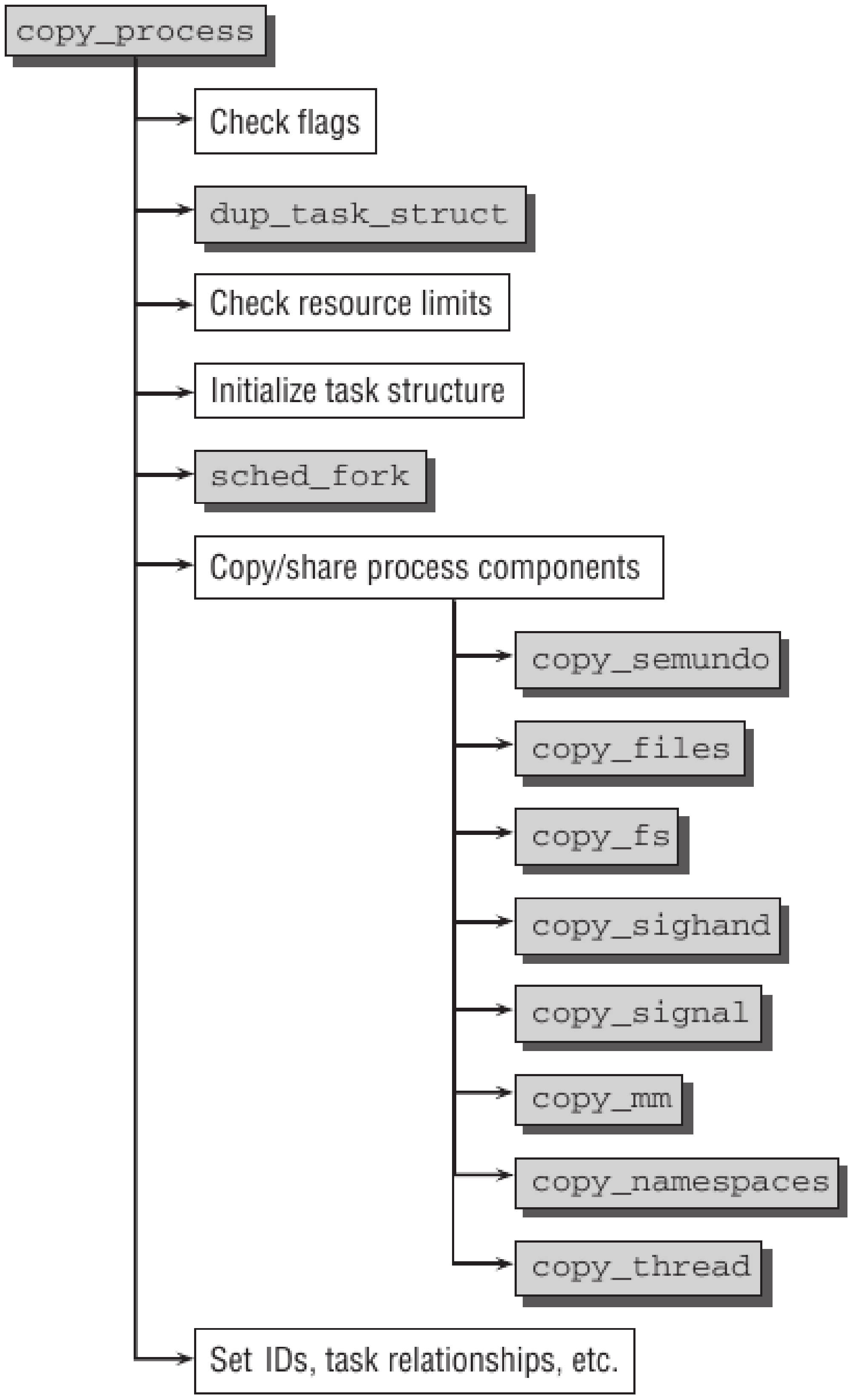
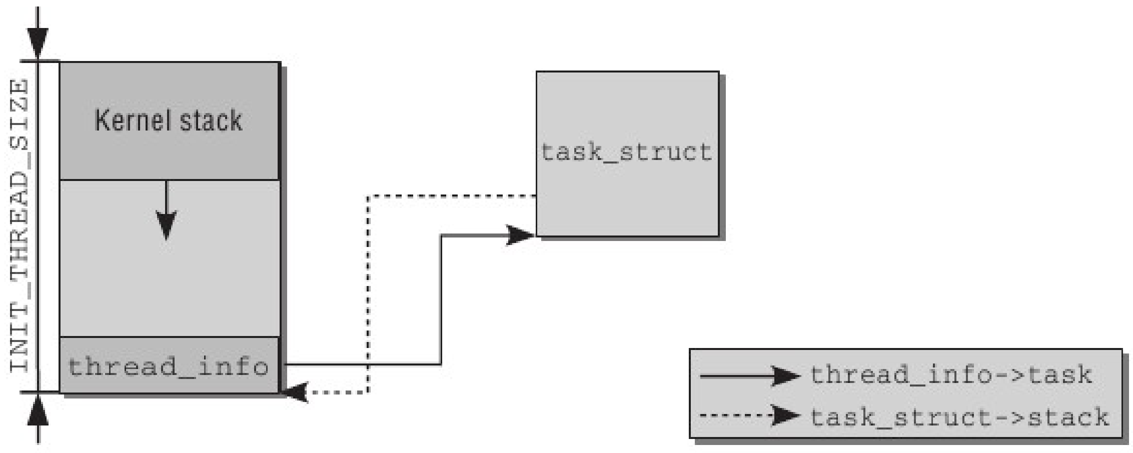


Illustration of copy_process internal call



kernel-stack, thread_info and struct_task



- importance of thread_info

There are also two symbols named current and current_thread_info that are defined as macros or functions by all architectures. Their meanings are as follows:

- current_thread_info delivers a pointer to the thread_info instance of the process currently executing. The address can be determined from the kernel stack pointer because the instance is always located at the top of the stack. Because a separate kernel stack is used for each process, the process to stack assignment is unique.

- current specifies the address of the task_struct instance of the current process. This function appears very frequently in the sources. The address can be determined using get_thread_info: current = current_thread_info()->task.

- flags field in thread_info can hold various process-specific flags, two of which are of particular interest to us:
  - TIF_SIGPENDING is set if the process has pending signals.
  - TIF_NEED_RESCHED indicates that the process should be or would like to be replaced with another process by the scheduler.

- thread_info is very useful while discussing about the cpu scheduler

Illustration of resource sharing between processes

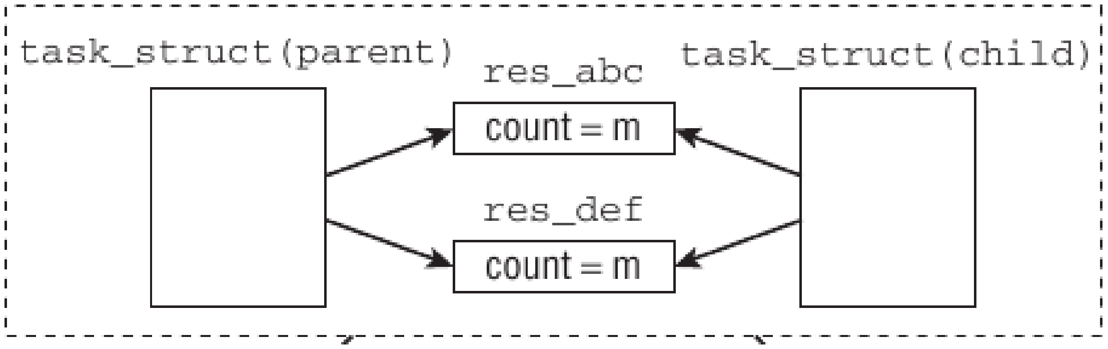
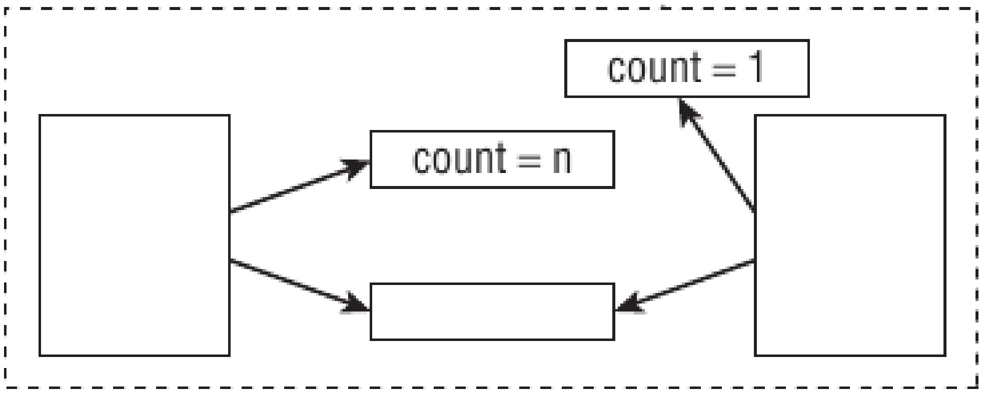


Illustration of independent resources for processes



overlaying the current process image with a new one

- execve() system call is used to replace the current process image with a new one
- the process remains the same , but the program, address-space and page-frames are replaced – old are freed and new are allocated
- this is the Linux way of process management

- this idea will get more clearer when we see virtual process address space

Process termination

- processes can be terminated using exit() system call (there is more to it – we will discuss during system programming sessions)
- this system call basically frees the resources of the process and sends the process to zombie state

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 (and, in fact, in parallel to execution of the kernel itself).

- kernel threads are often referred to as (kernel) daemons.

- they are used to perform, for example, the following tasks:
  - to periodically synchronize modified memory pages with the block device from which the pages originate (e.g., files mapped using mmap).
  - to write memory pages into the swap area if they are seldom used.
  - to manage deferred actions.
  - to implement transaction journals for filesystems.

- because kernel threads are generated by the kernel itself, two special points should be noted:
  - they execute in the supervisor mode of the CPU, not in the user mode
  - they may access only the kernel part of virtual address space (all addresses above TASK_SIZE) but not the virtual user area – kernel threads do not own virtual address-space in userland – contrast this with normal user-space processes

two pointers to mm_structs are contained in the task structure:

```
<sched.h>
struct task_struct {
...
        struct mm_struct *mm, *active_mm;
...
}
```

scheduler – a primer before the details

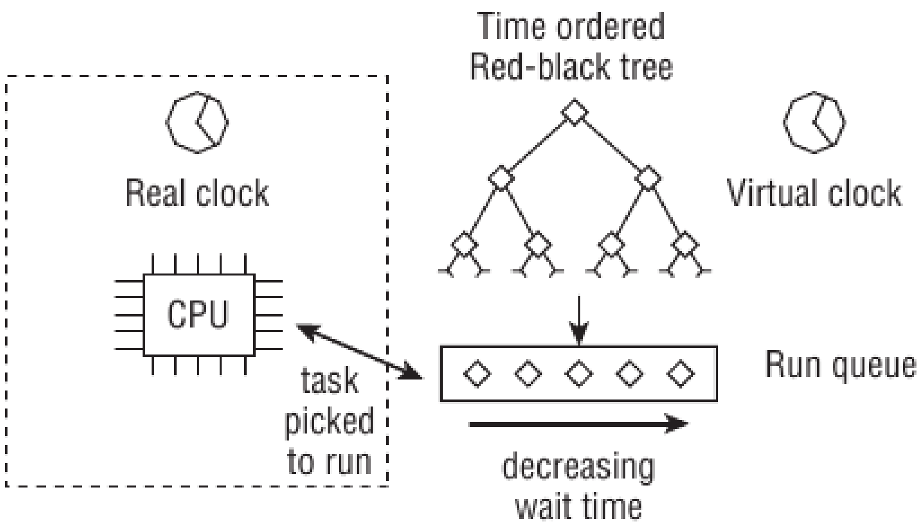
- if a system has only a single CPU, at most one process can be run simultaneously.
- multitasking is only achieved by switching back and forth between the tasks with high frequency. For users/applications, who think considerably more slowly than the switching f frequency, this creates the illusion of parallel executing, but in reality, it is not.

- while more CPUs in the system improve the situation and allow perfect parallel execution of a small number of tasks, there will always be situations in which fewer CPUs than processes that are to be run are available, and the problem starts anew.

- if multitasking is simulated by running one process after another, then the process that is currently running is favored over those waiting to be picked by the scheduler — the poor waiting processes are being treated unfairly. The unfairness is directly proportional to the waiting time.

- every time the scheduler is called, it picks the task with the highest waiting time and gives the CPU to it. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system.

Run-queue maintained by the system – listing all runnable processes



- all runnable tasks are time-ordered in a red-black tree, essentially with respect to their waiting time.

- the task that has been waiting for the CPU for the largest amount of time is the leftmost entry and will be considered next by the scheduler. Tasks that have been waiting less long are sorted on the tree from left to right.

- if you are not familiar with red-black trees, suffice it to know here that this data structure allows for efficient management of the entries it contains, and that the time required for lookup, insertion, and deletion operations will only moderately rise with the number of processes present in the tree.

- a run queue is also equipped with a virtual clock.
  time passes slower on this clock than in real time, and the exact speed depends on the number of processes that are currently waiting to be picked by the scheduler. Suppose that four processes are on the queue: Then the virtual clock will run at one-quarter of the speed of a real clock. This is the basis to determine how much CPU time a waiting process would have gotten if computational power could be shared in a completely fair manner.

- sitting on the run queue for 20 seconds in real time amounts to 5 seconds in virtual time. Four tasks executing for 5 seconds each would keep the CPU occupied for 20 seconds in real time.

- suppose that the virtual time of the run queue is given by fair_clock, while the waiting time of a process is stored in wait_runtime. To sort tasks on the red-black tree, the kernel uses the difference fair_clock - wait_runtime.

- while fair_clock is a measure for the CPU time a task would have gotten if scheduling were completely fair, wait_runtime is a direct measure for the unfairness caused by the imperfection of real systems.

- well, that is how theoretically we can present it – in reality, implementation is much more involved

- this strategy is complicated by a number of real-world issues:
  - different priority levels for tasks (i.e., nice values) must be taken into account, and more important processes must get a higher share of CPU time than less important ones.

  - tasks must not be switched too often because a context switch, that is, changing from one task to another, has a certain overhead. When switching happens too often, too much time is spent with exchanging tasks that is not available for effective work anymore.

  - on the other hand, the time that goes by between task switches must not be too long because large unfairness values could accumulate in this case. Letting tasks run for too long can also lead to larger latencies than desired for multimedia systems.

- the above scheduling algorithm is known as CFS (completely fair scheduling)

- currently(!!!), Linux supports real-time scheduling algorithms mandated by POSIX – FIFO and RR

- we are going to see the details below

Scheduler - generic scheduler code/data-structures

- scheduling can be activated in two ways:

  - either directly if a task goes to sleep or wants to yield the CPU for other reasons, or by a periodic mechanism that is run with constant frequency and that checks from time to time if switching tasks is necessary.

  - we may call these two components as generic scheduler or core scheduler; it has two major functions:

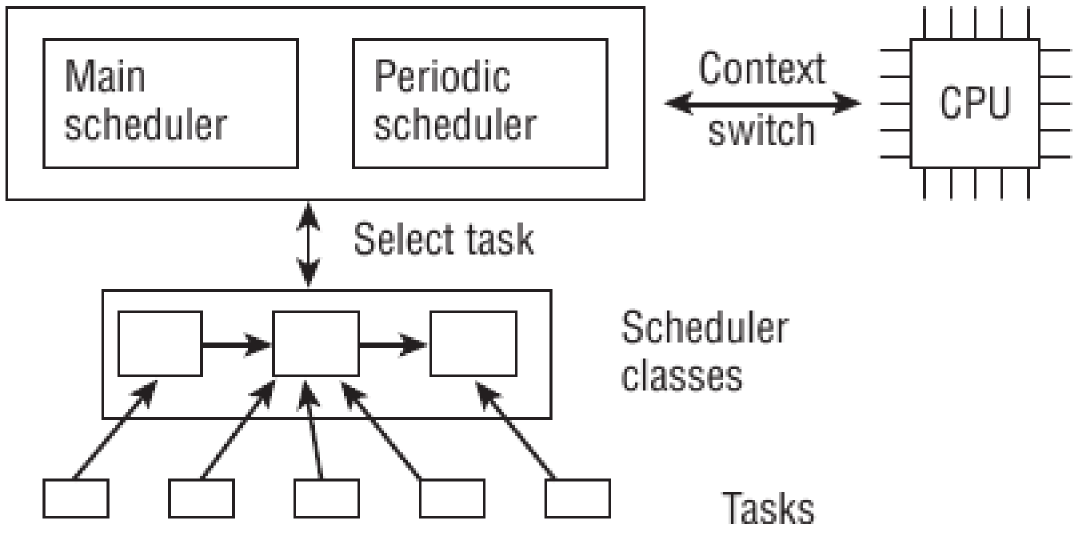
  - essentially, the generic scheduler is a dispatcher that interacts with two other components:

    - scheduling classes are used to decide which task runs next. The kernel supports different scheduling policies (completely fair scheduling, real-time scheduling, and scheduling of the idle task when there is nothing to do), and scheduling classes allow for implementing these policies in a modular way: Code from one class does not need to interact with code from other classes.

    - when the scheduler is invoked, it queries the scheduler classes which task is supposed to run next.

    - after a task has been selected to run, a low-level task switch must be performed. This requires close interaction with the underlying CPU – context-switching is really done here

Illustration of the core scheduler components



- every task belongs to exactly one of the scheduling classes, and each scheduling class is responsible to manage their tasks. The generic scheduler itself is not involved in managing tasks at all; this is completely delegated to the scheduler classes.

Scheduling related fields in the process descriptor

```
<linux/sched.h>
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;
...
}
```

- let us explore these fields in our discussions

- each scheduling class will have one such structure in the system

```
<linux/sched.h>

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);
};
```

Central Runqueue

- 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.

- the run queue is the starting point for many actions of the global scheduler.
- however, the processes are not directly managed by the general elements of the run queue this is the responsibility of the individual scheduler classes, and a class-specific sub-run queue is therefore embedded in each run-queue.

```
kernel/sched.c

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;
...
};
```

- let us discuss some of the key fields

scheduling entity

- since the scheduler can operate with more general entities than tasks, an appropriate data structure is required to describe such an entity. It is defined as follows:

- we only consider the case of task as the scheduling entity(not a group of tasks and so on....)

- the fields are use by the scheduling algorithms

<linux/sched.h>

```
 struct sched_entity {
         struct load_weight load; /* for load-balancing */
         struct rb_node run_node;
         unsigned int on_rq;
         u64  exec_start;
         u64  sum_exec_runtime;
         u64  vruntime;
         u64  prev_sum_exec_runtime;
 ...
 }
```

Scheduling priorities - user-space vs kernel-space

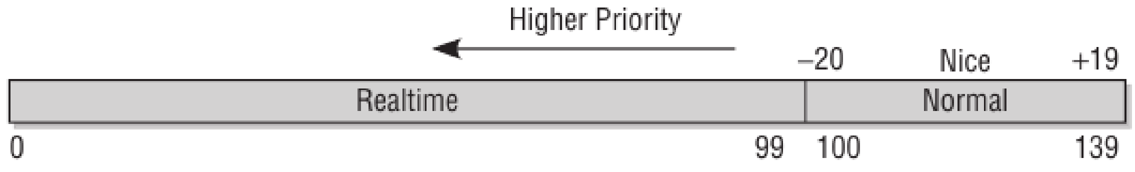
- for the CFS scheduling class, user-space can set the priorities in the range -20 to +19 – popularly known as nice values (welcome to Unix/Linux !!!)

- in the kernel, these are mapped to 100 to 139 (kernel view is different)

- real-time scheduling classes support user-space priorities in the range 1-99

- in the kernel, the real-time user-space priorities are mapped to 0-98(99 is mapped to 0 &
  1 is mapped to 98)

- summarized in the diagram below

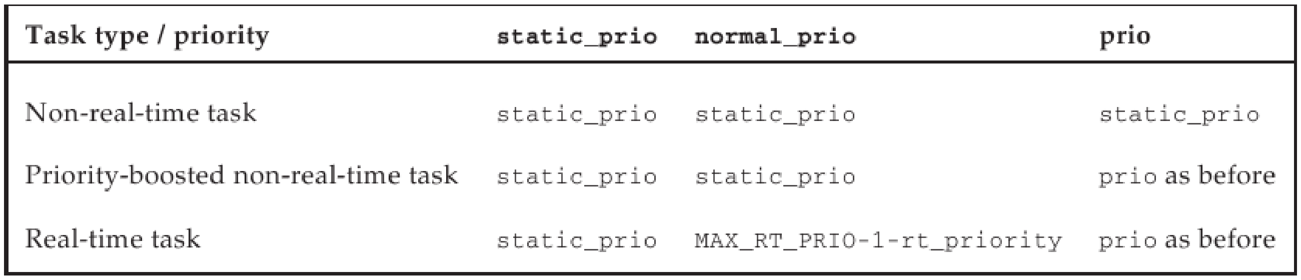


- some helpful macros that can help us understand the macros

```
<linux/sched.h>

#define   MAX_USER_RT_PRIO                    100
#define   MAX_RT_PRIO                         MAX_USER_RT_PRIO
#define   MAX_PRIO                            (MAX_RT_PRIO + 40)
#define   DEFAULT_PRIO                        (MAX_RT_PRIO + 20)

kernel/sched.c


#define NICE_TO_PRIO(nice)                    (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)                    ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)                          PRIO_TO_NICE((p)->static_prio)
```

- the three priority fields in each process descriptor are explained below:

- let us not bother about every case now – in the appropriate context, the special cases will get clear



Task type / priority	`static_prio`	`normal_prio`	`prio`
Non-real-time task	`static_prio`	`static_prio`	`static_prio`
Priority-boosted non-real-time task	`static_prio`	`static_prio`	`prio` as before
Real-time task	`static_prio`	`MAX_RT_PRIO-1-rt_priority`	`prio` as before

load weight – used for influencing the CFS with priorities

- 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 load weight is contained in the data structure load_weight:

```
<linux/sched.h>

 struct load_weight {
        unsigned long weight, inv_weight;  //used in CFS algorithm
};
```

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.

Let's first see the table:
kernel/sched.c

```
static const int prio_to_weight[40] = {

 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /* -5 */       3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
  ...........................................
}
```

- the array contains one entry for each nice level in the range [0, 39] as used by the kernel. The multiplier between the entries is 1.25
- to see why this is required, consider the following example. Two processes A and B run at nice level 0, so each one gets the same share of the CPU, namely, 50 percent. The weight for a nice 0 task is 1,024 as can be deduced from the table. the share for each task is 1024/ 1024+1024 = 0.5, is, 50 percent as expected.
- If task B is re-niced by one priority level, it is supposed to get 10 percent less CPU share. In other words, this means that A will get 55 percent and B will get 45 percent of the total CPU time. Increasing the priority by 1 leads to a decrease of its weight, which is then 1, 024/1.25 ≈ 820. The CPU share A will get now is therefore 1024/1024+820 ≈ 0.55, whereas B will have 820/1024+820 ≈ 0 .45 a 10 percent difference is required.

Periodic scheduler

- the periodic scheduler is implemented in scheduler_tick. the function is automatically called by the kernel with the frequency HZ if system activity is going on.

- the mechanism underlying periodic actions is discussed under time-management

- the function has two principal tasks.
  - to manage the kernel scheduling-specific statistics relating to the whole system and to the individual processes. The main actions performed involve incrementing counters and we we are not bothering about them here
  - to activate the periodic scheduling method of the scheduling class responsible for the current process.
- thanks to the modular structure of the scheduler, the main work is really simple, as it can be completely delegated to the scheduler-class-specific method:

  kernel/sched.c

```
scheduler_tick()
{

    ..................


        if (curr != rq->idle)
                curr->sched_class->task_tick(rq, curr);
}
```

- how task_tick is implemented depends on the underlying scheduler class. The completely fair scheduler, for instance, will in this method check if a process has been running for too long to avoid large latencies

- If the current task is supposed to be rescheduled, the scheduler class methods set the TIF_NEED_RESCHED flag in the task structure to express this request, and the kernel invokes the scheduler at next opportunity

Main scheduler

- the main scheduler function (schedule) is invoked directly at many points in the kernel to allocate the CPU to a process other than the currently active one. After returning from system calls, the kernel also checks whether the reschedule flag - TIF_NEED_RESCHED of the current process is set — for example, the flag is set by scheduler_tick as mentioned above. If it is, the kernel invokes schedule.

- the function then assumes that the currently active task is definitely to be replaced with another task. (not really – things are too practical)

kernel/sched.c

```
    asmlinkage void __sched schedule(void)
    {
            struct task_struct *prev, *next;
            struct rq *rq;
            int cpu;
      need_resched:
            cpu = smp_processor_id();
            rq = cpu_rq(cpu);
            prev = rq->curr;
    ...
```

As in the periodic scheduler, the kernel takes the opportunity to update the run queue clock and clears the reschedule flag TIF_NEED_RESCHED in the task structure of the currently running task.

```
            __update_rq_clock(rq);
            clear_tsk_need_resched(prev);
    ...


            prev->sched_class->put_prev_task(rq, prev);
            next = pick_next_task(rq, prev);
    ...
            if (likely(prev != next)) {
               rq->curr = next;
               context_switch(rq, prev, next);
            }
...
```

context-switching

The context switch proper is performed by invoking two processor-specific functions:

- switch_mm changes the memory context described in task_struct->mm. Depending on the processor, this is done by loading the page tables, flushing the translation lookaside buffers (partially or fully), and supplying the MMU with new information. Because these actions go deep into CPU details – pursue it , if required

- switch_to switches the processor register contents and the kernel stack (the virtual user address space is changed in the first step, and as it includes the user mode stack, it is not necessary to change the latter explicitly). This task also varies greatly from architecture to architecture and is usually coded entirely in assembly language. Again, I ignore implementation details.

- remember, however, that kernel threads do not have their own userspace memory context and execute on top of the address space of a random task; their task_struct->mm is NULL. the address space ‘‘borrowed’’ from the current task is noted in active_mm instead

CFS scheduler - insight

- one really crucial point of the completely fair scheduler is that sorting processes on the red-black tree is based on the following key:

kernel/sched_fair.c

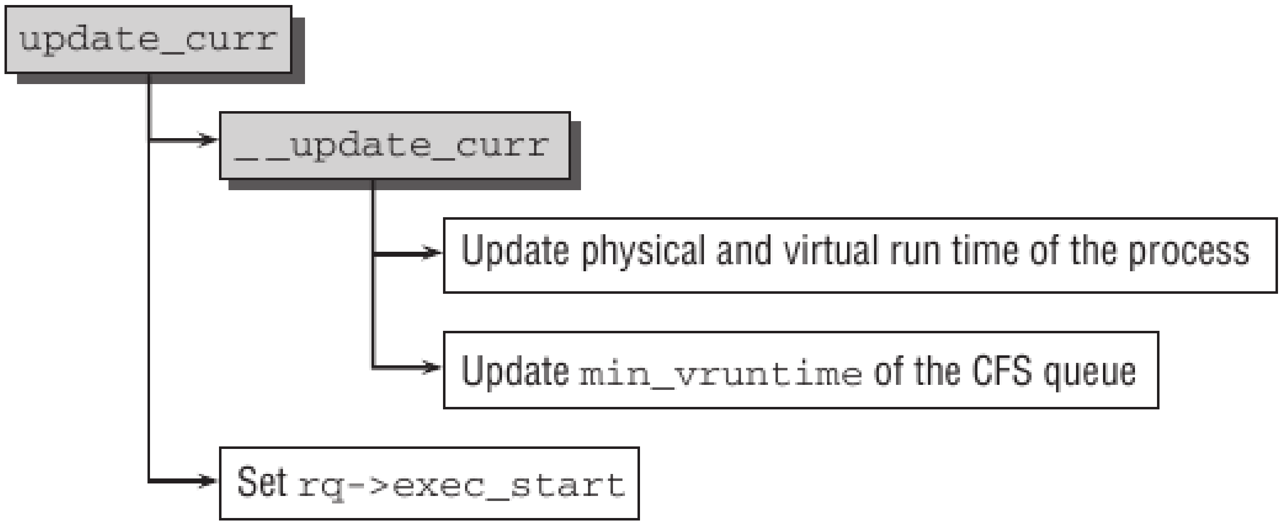
```
static inline s64 entity_key(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        return se->vruntime    -   cfs_rq->min_vruntime;
}
```

- in the above, se->vruntime represents the virtual time consumed by the process
- in the above cfs_rq->min_vruntime is that of the process with the least value
- it indirectly represents the waiting time of a process – smaller this is, the process is favoured

- elements with a smaller key will be placed more to the left, and thus be scheduled more quickly.
  this way, the kernel implements two antagonistic mechanisms:
  - when a process is running, its vruntime will steadily increase, so it will finally move rightward in the red-black tree.
    because vruntime will increase more slowly for more important processes, they will also move rightward more slowly, so their chance to be scheduled is bigger than for a less important process — just as required.
  - if a process sleeps, its vruntime will remain unchanged. Because the per-queue min_vruntime increases in the meantime (recall that it is monotonic!), the sleeper will be placed more to the left after waking up because the key got smaller.

Virtual clock emulation

- update_curr is called whenever required at different places / routines


update_curr
__update_curr
Update physical and virtual run time of the process
Update min_vruntime of the CFS queue
Set rq->exec_start

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_of(cfs_rq)->clock;
        unsigned long delta_exec;
        if (unlikely(!curr))
                return;
...
```

```
        delta_exec = (unsigned long)(now - curr->exec_start);
        __update_curr(cfs_rq, curr, delta_exec);
        curr->exec_start = now;
}
```

- based on this information, __update_curr has to update the physical and virtual time that the current process has spent executing on the CPU. This is simple for the physical time. The time difference just needs to be added to the previously accounted time:

```
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
        unsigned long delta_exec_weighted;
        u64 vruntime;
        curr->sum_exec_runtime += delta_exec;
...
```

- the interesting thing is how the non-existing virtual clock is emulated using the given information. once more, the kernel is clever and saves some time in the common case: For processes that run at nice level 0, virtual and physical time are identical by definition.

- when a different priority is used, the time must be weighted according to the load weight of the process

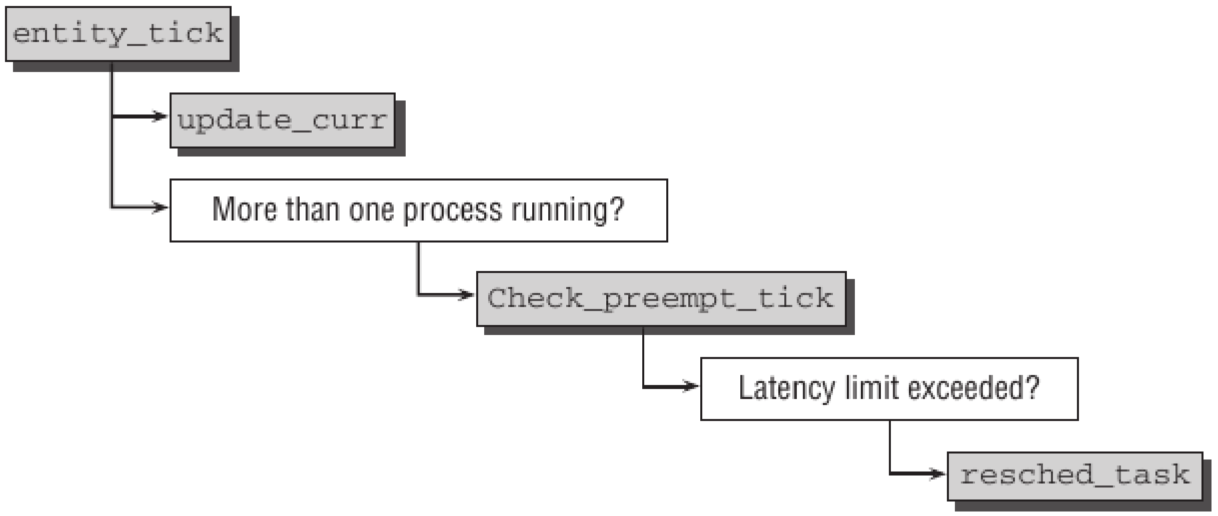
```
        delta_exec_weighted = delta_exec;
        if (unlikely(curr->load.weight != NICE_0_LOAD)) {
                delta_exec_weighted = calc_delta_fair(delta_exec_weighted,
                                                      &curr->load);
        }
        curr->vruntime += delta_exec_weighted;
...
```

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

- the inverse weight values mentioned above can be brought to good use in this calculation. recall 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 small.

- scheduler tick and CFS scheduler class

whenever scheduler_tick() is called, the scheduling class tick method of the current process is called - in our case, it is task_tick_fair - it calls many functions – the most interesting is entity_tick() and the flow is below:



kernel/sched_fair.c

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;
        ideal_runtime = sched_slice(cfs_rq, curr);
        delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
        if (delta_exec > ideal_runtime)
                resched_task(rq_of(cfs_rq)->curr);   //set TIF_NEED_RESCHED
}
```

Real-time scheduling class

- Real-time processes differ from normal processes in one essential way: If a real-time process exists in the system and is runnable, it will always be selected by the scheduler — unless there is another real-time process with a higher priority.

- normal priority of a real-time class process is calculated as follows:

```
kernel/sched.c
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;
}
```

- otherwise, it is calculated as follows:

```
kernel/sched.c
static inline int __normal_prio(struct task_struct *p)
{
        return p->static_prio;
}
```

- that shows that the kernel-priority of a real-time task will be lower than that of the kernel-priority of a non-real-time process

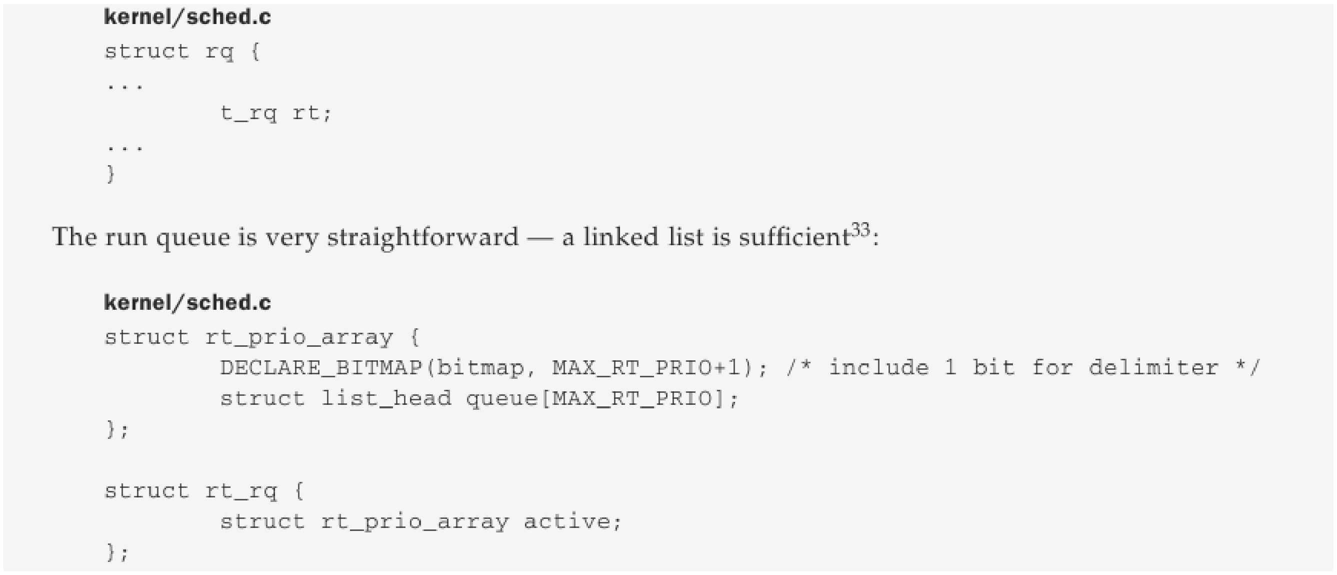
- The two available real-time classes differ as follows:

  - Round robin processes (SCHED_RR) 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.

  - First-in, first-out processes (SCHED_FIFO) do not have a time slice and are permitted to run as long as they want once they have been selected.

- It is evident that the system can be rendered unusable by badly programmed real-time processes — all that is needed is an endless loop whose loop body never sleeps. Extreme care should therefore be taken when writing real-time applications.

slide26



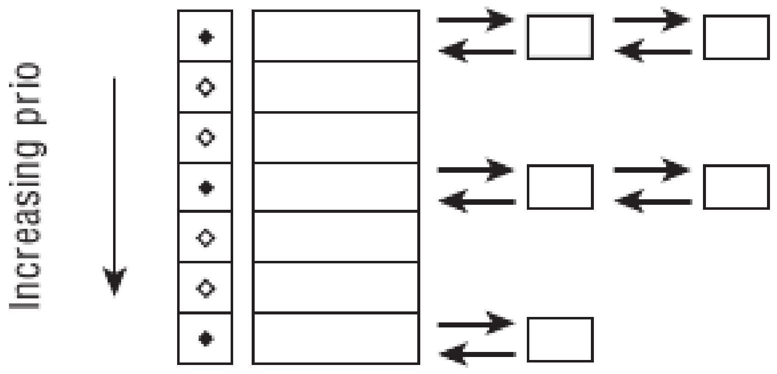
kernel/sched.c
```
struct rq {
...
        t_rq rt;
...
}
```

The run queue is very straightforward — a linked list is sufficient[33]:

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

struct rt_rq {
        struct rt_prio_array active;
};
```

- time-slices are decremented at HZ rate

real-time scheduling class run-queue and bit-map



kernel/sched-rt.c

```
        if (--p->time_slice)
                return;
        p->time_slice = DEF_TIMESLICE;
        /*
         * Requeue to the end of queue if we are not the only element
         * on the queue:
         */
        if (p->run_list.prev != p->run_list.next) {

                        requeue_task_rt(rq, p);
                        set_tsk_need_resched(p);
            }
    }
```

- The sched_setscheduler system call must be used to convert a process into a real-time process. This call performs only the following simple tasks:

  - It removes the process from its current queue using deactivate_task.
  - It sets the real-time priority and the scheduling class in the task data structure.
  - It reactivates the task.

- must be a priviliged user to do so

---------------------intentionally left blank------------------------

Kernel preemption related

- Kernel preemption was added during the development of kernel 2.5. Although astonishingly few changes were required to make the kernel preemptive, the mechanism is not as easy to implement as preemption of tasks running in user-space.

- If the kernel cannot complete certain actions in a single operation — manipulation of data structures, for instance — race conditions may occur and render the system inconsistent. The same problems arise on multiprocessor systems.

- kernel preemption counter

How does the kernel keep track of whether it can be preempted or not? Recall that each task in the system is equipped with an instance of struct thread_info. The structure also includes a preemption counter:

```
<asm-arch/thread_info.h>
struct thread_info {
...
        int preempt_count;      // 0 => preemptable
...
}
```
