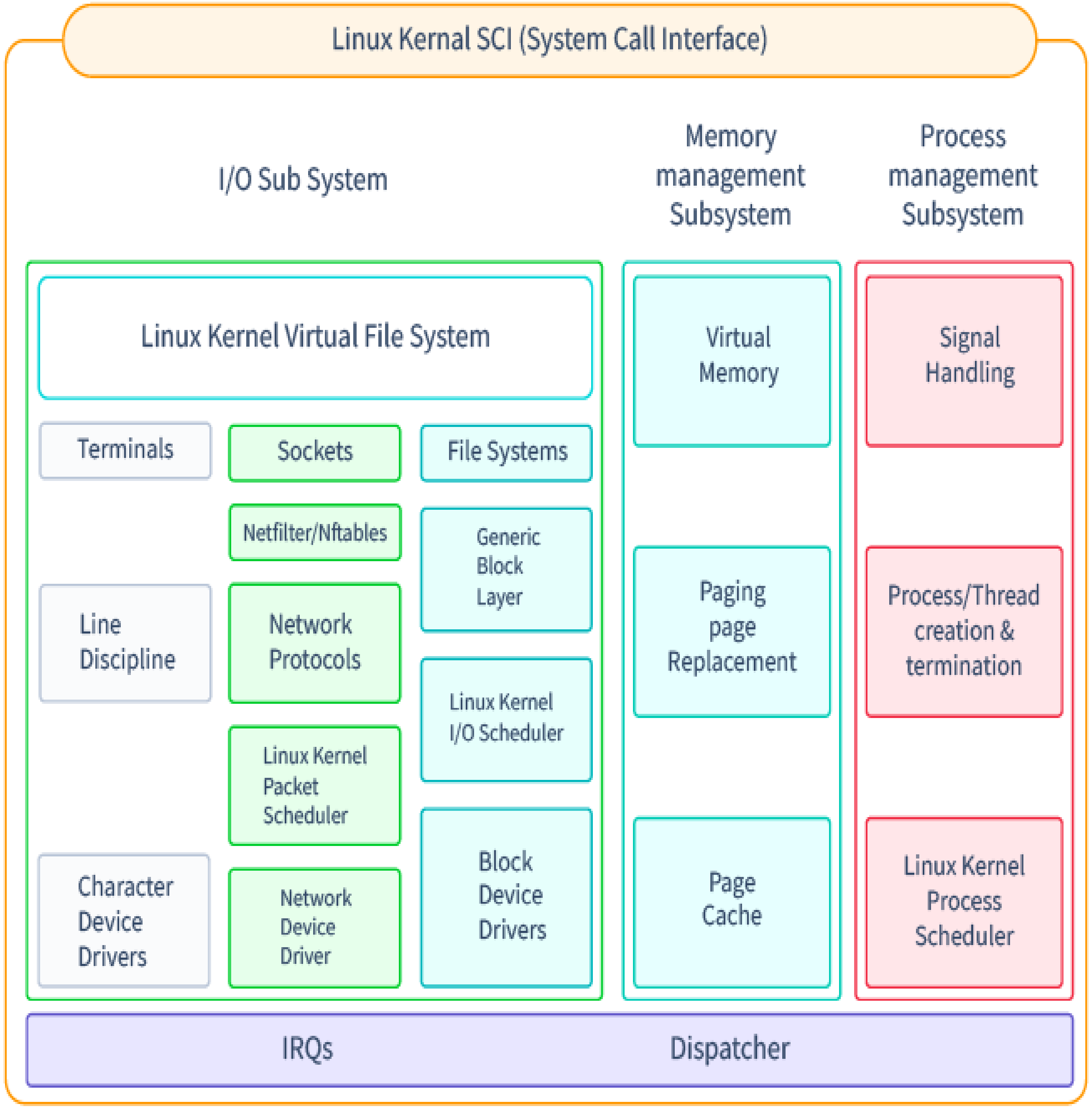
Process Scheduling

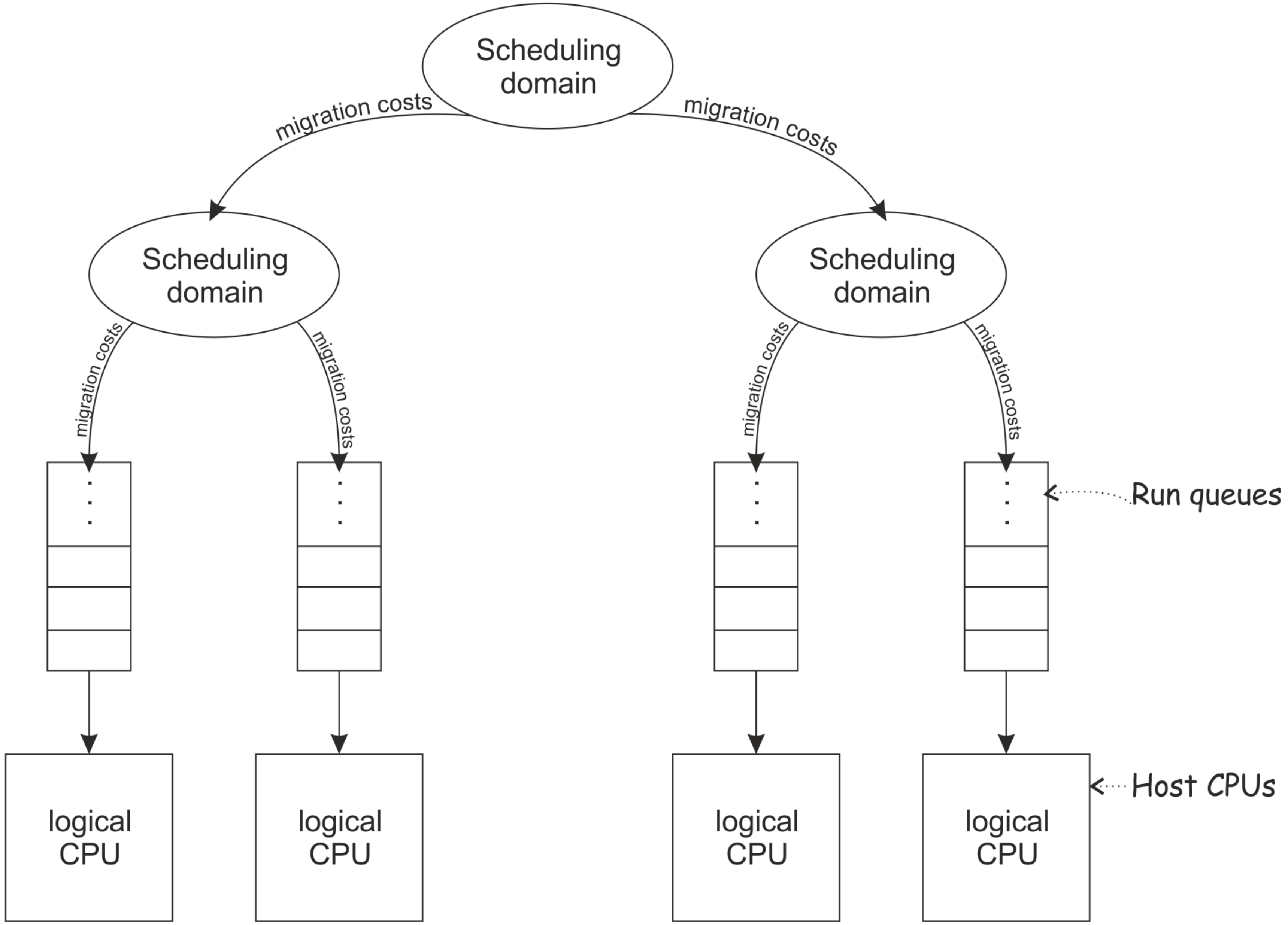
Introduction

- Like any time-sharing system, Linux achieves the effect of simultaneous execution of multiple processes by switching from one process to another in a very short time frame.
- The kernel subsystem that puts those processes to work is by, process scheduler.
- The process scheduler decides which process runs, when, and for how long.
- The process scheduler divides the finite resource of processor time between the runnable processes on a system.
- The scheduler is the basis of a multitasking operating system such as Linux.
- By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously.


Linux Kernal SCI (System Call Interface)
I/O Sub System
Memory management Subsystem
Process management Subsystem
Linux Kernel Virtual File System
Terminals
Sockets
File Systems
Netfilter/Nftables
Generic Block Layer
Line Discipline
Network Protocols
Linux Kernel I/O Scheduler
Linux Kernel Packet Scheduler
Character Device Drivers
Network Device Driver
Block Device Drivers
Virtual Memory
Paging page Replacement
Page Cache
Signal Handling
Process/Thread creation & termination
Linux Kernel Process Scheduler
IRQs
Dispatcher

Scheduling Policy

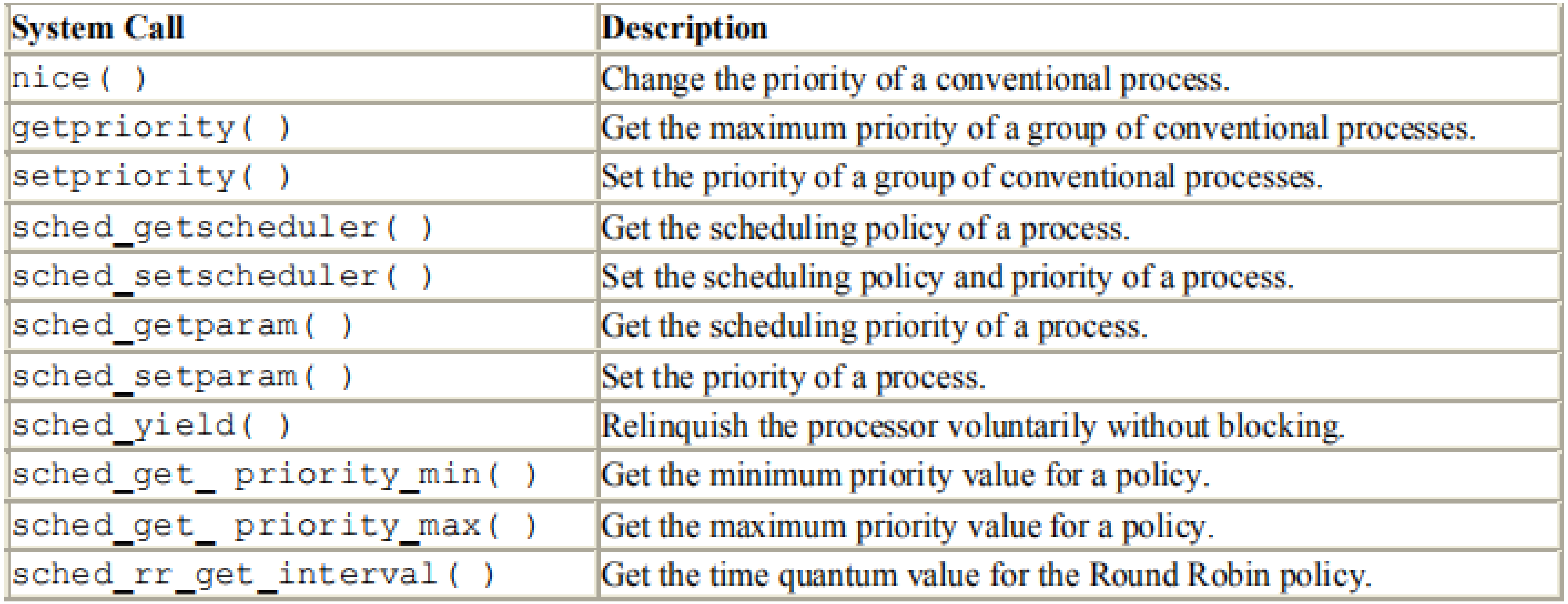
- The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on.
- The set of rules used to determine when and how selecting a new process to run is called scheduling policy.
- Linux scheduling is based on the time-sharing technique.
- Several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process.
- A single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or quantum expires, a process switch may take place.
- Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing.
- The scheduling policy is also based on ranking processes according to their priority. Each process is associated with a value that denotes how appropriate it is to be assigned to the CPU.


Scheduling domain
migration costs
migration costs
Scheduling domain
Scheduling domain
migration costs
migration costs
migration costs
migration costs
Run queues
logical CPU
logical CPU
logical CPU
logical CPU
Host CPUs

- In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of the CPU for a long time interval are boosted by dynamically increasing their priority.
- Correspondingly, processes running for a long time are penalized by decreasing their priority.
- When speaking about scheduling, processes are traditionally classified as "I/O-bound" or "CPU-bound."
- The former make heavy use of I/O devices and spend much time waiting for I/O operations to complete; the latter are number-crunching applications that require a lot of CPU time.
- An alternative classification distinguishes three classes of processes:

- **Interactive processes** - These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.
- **Batch processes** - These do not need user interaction, and hence they often run in the background. Since such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.
- **Real-time processes** - These have very strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should have a short response time and, most important, such response time should have a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

- A batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program).
- While in Linux real-time programs are explicitly recognized as such by the scheduling algorithm, there is no way to distinguish between interactive and batch programs.
- In order to offer a good response time to interactive applications, Linux (like all Unix kernels) implicitly favors I/O-bound processes over CPU-bound ones.



System Call	**Description**
`nice( )`	Change the priority of a conventional process.
`getpriority( )`	Get the maximum priority of a group of conventional processes.
`setpriority( )`	Set the priority of a group of conventional processes.
`sched_getscheduler( )`	Get the scheduling policy of a process.
`sched_setscheduler( )`	Set the scheduling policy and priority of a process.
`sched_getparam( )`	Get the scheduling priority of a process.
`sched_setparam( )`	Set the priority of a process.
`sched_yield( )`	Relinquish the processor voluntarily without blocking.
`sched_get_ priority_min( )`	Get the minimum priority value for a policy.
`sched_get_ priority_max( )`	Get the maximum priority value for a policy.
`sched_rr_get_interval( )`	Get the time quantum value for the Round Robin policy.

Process Premption

- Linux processes are preemptive. If a process enters the TASK_RUNNING state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process.
- If it is, the execution of current is interrupted and the scheduler is invoked to select another process to run. Of course, a process may also be preempted when its time quantum expires.
- When this occurs, the **need_resched** field of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.
- Be aware that a preempted process is not suspended, since it remains in the TASK_RUNNING state; it simply no longer uses the CPU.
- Some real-time operating systems feature preemptive kernels, which means that a process running in Kernel Mode can be interrupted after any instruction, just as it can in User Mode.
- The Linux kernel is not preemptive, which means that a process can be preempted only while running in User Mode; nonpreemptive kernel design is much simpler, since most synchronization problems involving the kernel data structures are easily avoided.

How Long must a Quantum Last?

- The quantum duration is critical for system performances: it should be neither too long nor too short.
- If the quantum duration is too short, the system overhead caused by task switches becomes excessively high. For instance, suppose that a task switch requires 10 milliseconds; if the quantum is also set to 10 milliseconds, then at least 50% of the CPU cycles will be dedicated to task switch.
- If the quantum duration is too long, processes no longer appear to be executed concurrently. For instance, let's suppose that the quantum is set to five seconds; each runnable process makes progress for about five seconds, but then it stops for a very long time.
- It is often believed that a long quantum duration degrades the response time of interactive applications. This is usually false.
- Interactive processes have a relatively high priority, therefore they quickly preempt the batch processes, no matter how long the quantum duration is.
- In some cases, a quantum duration that is too long degrades the responsiveness of the system.
- The choice of quantum duration is always a compromise. The rule of thumb adopted by Linux is: choose a duration as long as possible, while keeping good system response time.

Scheduling Algorithm

- The Linux scheduling algorithm works by dividing the CPU time into epochs .
- In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
- In general, different processes have different time quantum durations.
- The time quantum value is the maximum CPU time portion assigned to the process in that epoch.
- When a process has exhausted its time quantum, it is preempted and replaced by another runnable process.
- Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted—for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch.
- The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.
- Each process has a base time quantum: it is the time-quantum value assigned by the scheduler to the process if it has exhausted its quantum in the previous epoch.
- The users can change the base time quantum of their processes by using the **nice( )** and **setpriority( )** system calls. A new process always inherits the base time quantum of its parent.

- The **INIT_TASK** macro sets the value of the base time quantum of process (swapper) to **DEF_PRIORITY**; that macro is defined as follows:

```
#define DEF_PRIORITY (20*HZ/100)
```

- Since HZ, which denotes the frequency of timer interrupts, is set to 100 for IBM PCs, the value of **DEF_PRIORITY** is 20 ticks, that is, about 210 ms.
- Users rarely change the base time quantum of their processes, so **DEF_PRIORITY** also denotes the base time quantum of most processes in the system.
- In order to select a process to run, the Linux scheduler must consider the priority of each process. Actually, there are two kinds of priority:
- **Static priority** - This kind is assigned by the users to real-time processes and ranges from 1 to 99. It is never changed by the scheduler.
- **Dynamic priority** - This kind applies only to conventional processes; it is essentially the sum of the base time quantum (which is therefore also called the base priority of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch.

Data Structure used by the scheduler()

- The process list links together all process descriptors, while the runqueue list links together the process descriptors of all runnable processes—that is, of those in a TASK_RUNNING state.
- In both cases, the **init_task** process descriptor plays the role of list header. Each process descriptor includes several fields related to scheduling:
- **need_resched -** A flag checked by **ret_from_intr( )** to decide whether to invoke the schedule( )function.
- **policy -** The scheduling class. The values permitted are:
    - **SCHED_FIFO -** A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real-time process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.
    - **SCHED_RR -** A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all SCHED_RR real-time processes that have the same priority.
    - **SCHED_OTHER -** A conventional, time-shared process. The policy field also encodes a SCHED_YIELD binary flag. This flag is set when the process invokes the **sched_ yield( )** system call a way of voluntarily relinquishing the processor without the need to start an I/O operation or go to sleep; The scheduler puts the process descriptor at the bottom of the runqueue list.

- **rt_priority** - The static priority of a real-time process. Conventional processes do not make use of this field.
- **priority** - The base time quantum (or base priority) of the process.
- **counter** - The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process.

- Recall that the **update_process_times( )** function decrements the counter field of the current process by 1 at every tick.
- When a new process is created, **do_fork( )** sets the counter field of both current (the parent) and p (the child) processes in the following way:

```
current->counter >>= 1;
p->counter = current->counter;
```

- In other words, the number of ticks left to the parent is split in two halves, one for the parent and one for the child.
- This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires.
- This programming trick does not work since the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting lots of background processes in a shell or by opening a lot of windows on a graphical desktop.
- More generally speaking, a process cannot hog resources by forking multiple descendents.

The schedule() function

- **schedule( )** implements the scheduler. Its objective is to find a process in the runqueue list and then assign the CPU to it.
- It is invoked, directly or in a lazy way, by several kernel routines.
- **Direct Invocation** - The scheduler is invoked directly when the current process must be blocked right away because the resource it needs is not available.
- In this case, the kernel routine that wants to block it proceeds as follows:

    1. Inserts current in the proper wait queue

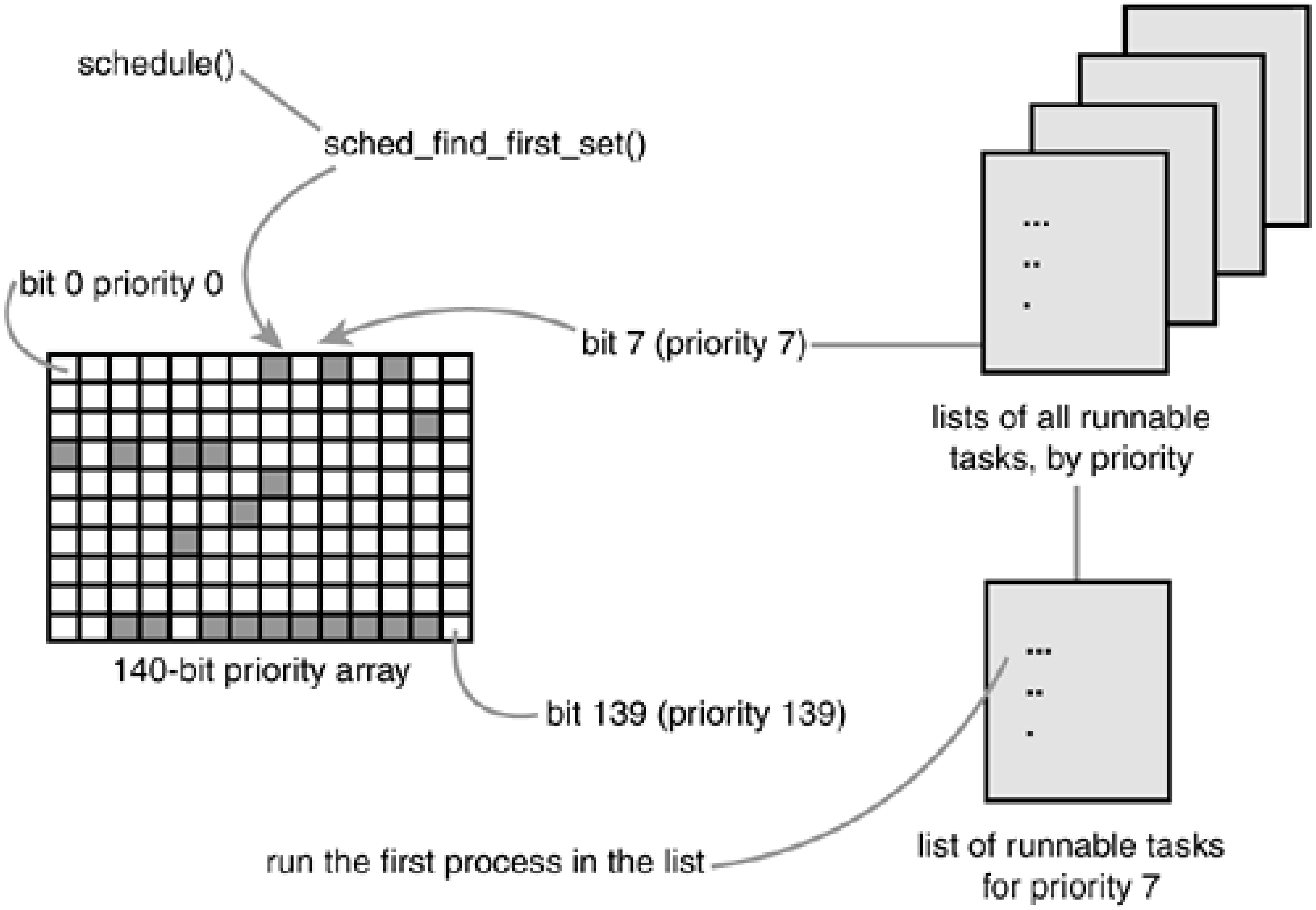
    2. Changes the state of current either to TASK_INTERRUPTIBLE or to TASK_UNINTERRUPTIBLE

    3. Invokes schedule( )

    4. Checks if the resource is available; if not, goes to step 2

    5. Once the resource is available, removes current from the wait queue

- The kernel routine checks repeatedly whether the resource needed by the process is available; if not, it yields the CPU to some other process by invoking schedule().
- Later, when the scheduler once again grants the CPU to the process, the availability of the resource is again checked.
- The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the **need_resched** field and, if necessary, invokes **schedule( )** to voluntarily relinquish the CPU.


schedule()
sched_find_first_set()
bit 0 priority 0
bit 7 (priority 7)
140-bit priority array
bit 139 (priority 139)
lists of all runnable tasks, by priority
list of runnable tasks for priority 7
run the first process in the list

- **Lazy Invocation** - The scheduler can also be invoked in a lazy way by setting the **need_resched** field of current to 1.
- Since a check on the value of this field is always made before resuming the execution of a User Mode process , **schedule( )** will definitely be invoked at some close future time.
- Lazy invocation of the scheduler is performed in the following cases:
  - When current has used up its quantum of CPU time; this is done by the **update_process_times( )** function. When a process is woken up and its priority is higher than that of the current process; this task is performed by the **reschedule_idle( )** function, which is invoked by the **wake_up_process( )** function.

    ```
    If (goodness(current, p) > goodness(current, current))
    current->need_resched = 1;
    ```
  - When a **sched_setscheduler( )** or **sched_ yield( )** system call is issued.

Action performed by schedule()

- Before actually scheduling a process, the schedule( ) function starts by running the functions left by other kernel control paths in various queues. The function invokes **run_task_queue( )** on the **tq _scheduler** task queue.
- Linux puts a function in that task queue when it must defer its execution until the next **schedule( )** invocation:

```
run_task_queue(&tq_scheduler);
```

- The function then executes all active unmasked bottom halves. These are usually present to perform tasks requested by device drivers.

```
if (bh_active & bh_mask)
do_bottom_half( );
```

- Now comes the actual scheduling, and therefore a potential process switch.
- The value of current is saved in the prev local variable and the **need_resched** field of previous set to 0. The key outcome of the function is to set another local variable called next so that it points to the descriptor of the process selected to replace prev.

- First, a check is made to determine whether prev is a Round Robin real-time process (policyfield set to SCHED_RR) that has exhausted its quantum. If so, **schedule( )** assigns a new quantum to prev and puts it at the bottom of the runqueue list:

```
if (!prev->counter && prev->policy == SCHED_RR) {
        prev->counter = prev->priority;
        move_last_runqueue(prev);
        }
```

- Now **schedule( )** examines the state of prev. If it has nonblocked pending signals and its state is TASK_INTERRUPTIBLE, the function wakes up the process as follows. This action is not the same as assigning the processor to prev; it just gives prev a chance to be selected for execution:

```
if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
```

- If prev is not in the TASK_RUNNING state, **schedule( )** was directly invoked by the process itself because it had to wait on some external resource; therefore, prev must be removed from the runqueue list:

```
if (prev->state != TASK_RUNNING)
    del_from_runqueue(prev);
```

- Next, **schedule( )** must select the process to be executed in the next time quantum.
- To that end, the function scans the runqueue list. It starts from the process referenced by the **next_run** field of init_task, which is the descriptor of process (swapper).
- The objective is to store in next the process descriptor pointer of the highest priority process. In order to do this, next is initialized to the first runnable process to be checked, and c is initialized to its "goodness".

```
        if (prev->state == TASK_RUNNING) {
                next = prev;
         if (prev->policy & SCHED_YIELD) {
                prev->policy &= ~SCHED_YIELD;
                 c = 0;
         } else
         c = goodness(prev, prev);
        } else {
        c = -1000;
        next = &init_task;
}
```

- If the SCHED_YIELD flag of prev->policy is set, prev has voluntarily relinquished the CPU by issuing a sched_ yield( ) system call. In this case, the function assigns a zero goodness to it.

- Now **schedule( )** repeatedly invokes the **goodness( )** function on the runnable processes to determine the best candidate:

```
p = init_task.next_run;
while (p != &init_task) {
          weight = goodness(prev, p);
if (weight > c) {
          c = weight;
          next = p;
}
          p = p->next_run;
}
```

- The while loop selects the first process in the runqueue having maximum weight. If the previous process is runnable, it is preferred with respect to other runnable processes having the same weight.

- If the runqueue list is empty (no runnable process exists except for swapper), the cycle is not entered and next points to init_task. Moreover, if all processes in the runqueue list have a priority lesser than or equal to the priority of prev, no process switch will take place and the old process will continue to be executed.
- A further check must be made at the exit of the loop to determine whether c is 0. This occurs only when all the processes in the runqueue list have exhausted their quantum, that is, all of them have a zero counter field.
- When this happens, a new epoch begins, therefore schedule( ) assigns to all existing processes (not only to the TASK_RUNNING ones) a fresh quantum, whose duration is the sum of the priority value plus half the counter value:

```
if (!c) {
        for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}
```

- In this way, suspended or stopped processes have their dynamic priorities periodically increased. As stated earlier, the rationale for increasing the counter value of suspended or stopped processes is to give preference to I/O-bound processes. However, even after an infinite number of increases, the value of counter can never become larger than twice the priority value.

- The concluding part of **schedule( )**: if a process other than prev has been selected, a process switch must take place. Before performing it, however, the context_swtch field of kstat is increased by 1 to update the statistics maintained by the kernel:

```
if (prev != next) {
          kstat.context_swtch++;
          switch_to(prev,next);
          }
          return;
```

- Notice that the return statement that exits from schedule( ) will not be performed right away by the next process but at a later time by the prev one when the scheduler selects it again for execution.

How Good Is a Runnable Process?

- The heart of the scheduling algorithm includes identifying the best candidate among all processes in the runqueue list. This is what the **goodness( )** function does.
- It receives as input parameters prev (the descriptor pointer of the previously running process) and p (the descriptor pointer of the process to evaluate). The integer value c returned by **goodness( )** measures the "goodness" of p and has the following meanings:
- c = -1000 - p must never be selected; this value is returned when the runqueue list contains only init_task.
- c = 0 - p has exhausted its quantum. Unless p is the first process in the runqueue list and all runnable processes have also exhausted their quantum, it will not be selected for execution.
- 0 < c < 1000 - p is a conventional process that has not exhausted its quantum; a higher value of c denotes a higher level of goodness.
- c >= 1000 - p is a real-time process; a higher value of c denotes a higher level of goodness.

- The goodness( ) function is equivalent to:

```
if (p->policy != SCHED_OTHER)
return 1000 + p->rt_priority;
if (p->counter == 0)
        return 0;
if (p->mm == prev->mm)
        return p->counter + p->priority + 1;
        return p->counter + p->priority;
```

- If the process is real-time, its goodness is set to at least 1000. If it is a conventional process that has exhausted its quantum, its goodness is set to 0; otherwise, it is set to p->counter + p->priority.
- A small bonus is given to p if it shares the address space with prev (i.e., if their process descriptors' mm fields point to the same memory descriptor). The rationale for this bonus is that if p runs right after prev, it will use the same page tables, hence the same memory; some of the valuable data may still be in the hardware cache.

System Calls Related to Scheduling

- **nice() system call** - The **nice( )** system call allows processes to change their base priority. The integer value contained in the increment parameter is used to modify the priority field of the process descriptor.
- The nice Unix command, which allows users to run programs with modified scheduling priority, is based on this system call.
- The **sys_nice( )** service routine handles the **nice( )** system call.
- The increment parameter may have any value, absolute values larger than 40 are trimmed down to 40.
- Negative values correspond to requests for priority increments, while positive ones correspond to requests for priority decrements.

```
increase = 0
newprio = increment;
if (increment < 0) {
if (!capable(CAP_SYS_NICE))
 return -EPERM;
 newprio = -increment;
 increase = 1;
}
```

```
                if (newprio > 40)
                newprio = 40;
        newprio = (newprio * DEF_PRIORITY + 10) / 20;
                increment = newprio;
                if (increase)
         increment = -increment;
if (current->priority - increment < 1)
        current->priority = 1;
        else if (current->priority > DEF_PRIORITY*2)
current->priority = DEF_PRIORITY*2;
else
current->priority -= increment;
        return 0;
```

The getpriority( ) and setpriority( ) System Calls

- The **nice( )** system call affects only the process that invokes it.
- Two other system calls, denoted as **getpriority( )** and **setpriority( )**, act on the base priorities of all processes in a given group.
- **getpriority( )** returns 20 plus the highest base priority among all processes in a given group;
- **setpriority( )** sets the base priority of all processes in a given group to a given value.
- The kernel implements these system calls by means of the sys_getpriority( ) and **sys_setpriority( )** service routines. Both of them act essentially on the same group of parameters:

- **which -** Identifies the group of processes; it can assume one of the following values:
- **PRIO_PROCESS -** Select the processes according to their process ID (pid field of the process descriptor).
- **PRIO_PGRP -** Select the processes according to their group ID (pgrp field of the process descriptor).
- **PRIO_USER -** Select the processes according to their user ID (uid field of the process descriptor)

- **who -** Value of the pid, pgrp, or uid field (depending on the value of which) to be used for selecting the processes. If who is 0, its value is set to that of the corresponding field of the current process.

- **niceval -** The new base priority value (needed only by sys_setpriority( )). It should range between -20 (highest priority) and +20 (minimum priority).

Performance of the Scheduling Algorithm

- **The algorithm does not scale well** - If the number of existing processes is very large, it is inefficient to recompute all dynamic priorities at once. The dynamic priorities were recomputed every second, which made the problem worse. Linux tries to minimize the overhead of the scheduler by recomputing the priorities of all runnable processes which have exhausted their time quantum. Therefore, when the number of processes is large, the recomputation phase is more expensive but is executed less frequently.
- **The predefined quantum is too large for high system loads** - The system responsiveness experienced by users depends heavily on the system load, which is the average number of processes that are runnable, and hence waiting for CPU time.
- **I/O-bound process boosting strategy is not optimal** - The preference for I/O-bound processes is a good strategy to ensure a short response time for interactive programs, but it is not perfect. Interactive programs that are also CPU-bound may appear unresponsive to the users.
- **Support for real-time applications is weak** – Non-preemptive kernels are not well suited for real-time applications, since processes may spend several milliseconds in Kernel Mode while handling an interrupt or exception. During this time, a real-time process that becomes runnable cannot be resumed. This is unacceptable for real-time applications, which require predictable and low response times.