NAME

sched − overview of CPU scheduling

DESCRIPTION

Since Linux 2.6.23, the default scheduler is CFS, the "Completely Fair Scheduler". The CFS scheduler replaced the earlier "O(1)" scheduler.

API summary

Linux provides the following system calls for controlling the CPU scheduling behavior, policy, and priority of processes (or, more precisely, threads).

nice(2) Set a new nice value for the calling thread, and return the new nice value.

getpriority(2)
Return the nice value of a thread, a process group, or the set of threads owned by a specified user.

setpriority(2)
Set the nice value of a thread, a process group, or the set of threads owned by a specified user.

sched_setscheduler(2)
Set the scheduling policy and parameters of a specified thread.

sched_getscheduler(2)
Return the scheduling policy of a specified thread.

sched_setparam(2)
Set the scheduling parameters of a specified thread.

sched_getparam(2)
Fetch the scheduling parameters of a specified thread.

sched_get_priority_max(2)
Return the maximum priority available in a specified scheduling policy.

sched_get_priority_min(2)
Return the minimum priority available in a specified scheduling policy.

sched_rr_get_interval(2)
Fetch the quantum used for threads that are scheduled under the "round-robin" scheduling policy.

sched_yield(2)
Cause the caller to relinquish the CPU, so that some other thread be executed.

sched_setaffinity(2)
(Linux-specific) Set the CPU affinity of a specified thread.

sched_getaffinity(2)
(Linux-specific) Get the CPU affinity of a specified thread.

sched_setattr(2)
Set the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of **sched_setscheduler**(2) and **sched_setparam**(2).

sched_getattr(2)
Fetch the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of **sched_getscheduler**(2) and **sched_getparam**(2).

Scheduling policies

The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a *static* scheduling priority, *sched_priority*. The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all threads on the system.

For threads scheduled under one of the normal scheduling policies (**SCHED_OTHER**, **SCHED_IDLE**, **SCHED_BATCH**), *sched_priority* is not used in scheduling decisions (it must be specified as 0).

Processes scheduled under one of the real-time policies (**SCHED_FIFO**, **SCHED_RR**) have a *sched_priority* value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always have higher priority than normal threads.) Note well: POSIX.1 requires an implementation to support only a minimum 32 distinct priority levels for the real-time policies, and some systems supply just this minimum. Portable programs should use **sched_get_priority_min**(2) and **sched_get_priority_max**(2) to find the range of priorities supported for a particular policy.

Conceptually, the scheduler maintains a list of runnable threads for each possible *sched_priority* value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and selects the thread at the head of this list.

A thread's scheduling policy determines where it will be inserted into the list of threads with equal static priority and how it will move inside this list.

All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. The scheduling policy determines the ordering only within the list of runnable threads with equal static priority.

SCHED_FIFO: First in-first out scheduling

SCHED_FIFO can be used only with static priorities higher than 0, which means that when a **SCHED_FIFO** thread becomes runnable, it will always immediately preempt any currently running **SCHED_OTHER**, **SCHED_BATCH**, or **SCHED_IDLE** thread. **SCHED_FIFO** is a simple scheduling algorithm without time slicing. For threads scheduled under the **SCHED_FIFO** policy, the following rules apply:

1) A running **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 blocked **SCHED_FIFO** thread becomes runnable, it will be inserted at the end of the list for its priority.

3) If a call to **sched_setscheduler**(2), **sched_setparam**(2), **sched_setattr**(2), **pthread_setschedparam**(3), or **pthread_setschedprio**(3) changes the priority of the running or runnable **SCHED_FIFO** thread identified by *pid* the effect on the thread's position in the list depends on the direction of the change to threads priority:

   - If the thread's priority is raised, it is placed at the end of the list for its new priority. As a consequence, it may preempt a currently running thread with the same priority.
   - If the thread's priority is unchanged, its position in the run list is unchanged.
   - If the thread's priority is lowered, it is placed at the front of the list for its new priority.

   According to POSIX.1-2008, changes to a thread's priority (or policy) using any mechanism other than **pthread_setschedprio**(3) should result in the thread being placed at the end of the list for its priority.

4) A thread calling **sched_yield**(2) will be put at the end of the list.

No other events will move a thread scheduled under the **SCHED_FIFO** policy in the wait list of runnable threads with equal static priority.

A **SCHED_FIFO** thread runs until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls **sched_yield**(2).

SCHED_RR: Round-robin scheduling

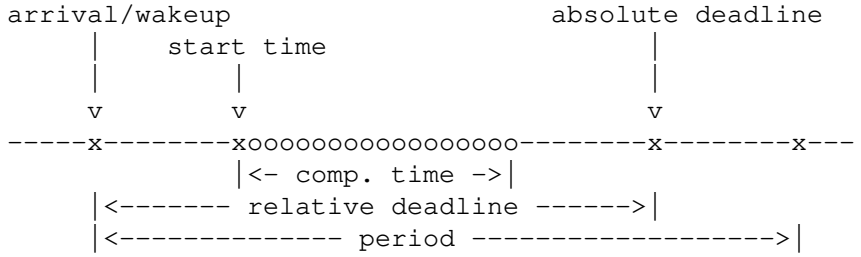
SCHED_RR is a simple enhancement of **SCHED_FIFO**. Everything described above for **SCHED_FIFO** also applies to **SCHED_RR**, except that each thread is allowed to run only for a maximum time quantum. If a **SCHED_RR** thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A **SCHED_RR** thread that has been preempted by a higher priority thread and subsequently resumes execution as a running thread will complete the unexpired portion of its round-robin time quantum. The length of the time quantum can be retrieved using **sched_rr_get_interval**(2).

SCHED_DEADLINE: Sporadic task model deadline scheduling

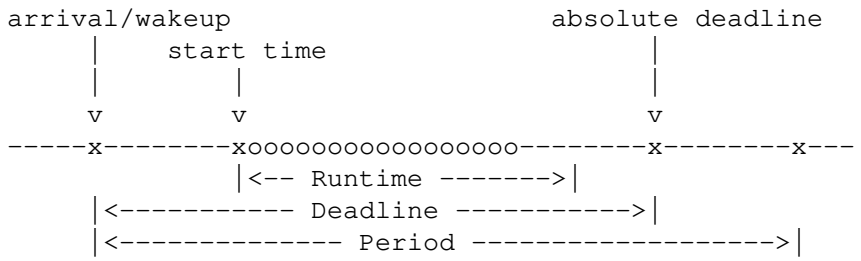
Since version 3.14, Linux provides a deadline scheduling policy (**SCHED_DEADLINE**). This policy is currently implemented using GEDF (Global Earliest Deadline First) in conjunction with CBS (Constant Bandwidth Server). To set and fetch this policy and associated attributes, one must use the Linux-specific **sched_setattr**(2) and **sched_getattr**(2) system calls.

A sporadic task is one that has a sequence of jobs, where each job is activated at most once per period. Each job also has a *relative deadline*, before which it should finish execution, and a *computation time*, which is the CPU time necessary for executing the job. The moment when a task wakes up because a new job has to be executed is called the *arrival time* (also referred to as the request time or release time). The *start time* is the time at which a task starts its execution. The *absolute deadline* is thus obtained by adding the relative deadline to the arrival time.

The following diagram clarifies these terms:



When setting a **SCHED_DEADLINE** policy for a thread using **sched_setattr**(2), one can specify three parameters: *Runtime*, *Deadline*, and *Period*. These parameters do not necessarily correspond to the aforementioned terms: usual practice is to set Runtime to something bigger than the average computation time (or worst-case execution time for hard real-time tasks), Deadline to the relative deadline, and Period to the period of the task. Thus, for **SCHED_DEADLINE** scheduling, we have:



The three deadline-scheduling parameters correspond to the *sched_runtime*, *sched_deadline*, and *sched_period* fields of the *sched_attr* structure; see **sched_setattr**(2). These fields express values in nanoseconds. If *sched_period* is specified as 0, then it is made the same as *sched_deadline*.

The kernel requires that:

sched_runtime <= sched_deadline <= sched_period

In addition, under the current implementation, all of the parameter values must be at least 1024 (i.e., just over one microsecond, which is the resolution of the implementation), and less than 2^63. If any of these checks fails, **sched_setattr**(2) fails with the error **EINVAL**.

The CBS guarantees non-interference between tasks, by throttling threads that attempt to over-run their specified Runtime.

To ensure deadline scheduling guarantees, the kernel must prevent situations where the set of **SCHED_DEADLINE** threads is not feasible (schedulable) within the given constraints. The kernel thus performs an admittance test when setting or changing **SCHED_DEADLINE** policy and attributes. This admission test calculates whether the change is feasible; if it is not, **sched_setattr**(2) fails with the error **EBUSY**.

For example, it is required (but not necessarily sufficient) for the total utilization to be less than or equal to the total number of CPUs available, where, since each thread can maximally run for Runtime per Period,

that thread's utilization is its Runtime divided by its Period.

In order to fulfill the guarantees that are made when a thread is admitted to the **SCHED_DEADLINE** policy, **SCHED_DEADLINE** threads are the highest priority (user controllable) threads in the system; if any **SCHED_DEADLINE** thread is runnable, it will preempt any thread scheduled under one of the other policies.

A call to **fork**(2) by a thread scheduled under the **SCHED_DEADLINE** policy fails with the error **EAGAIN**, unless the thread has its reset-on-fork flag set (see below).

A **SCHED_DEADLINE** thread that calls **sched_yield**(2) will yield the current job and wait for a new period to begin.

SCHED_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can be used at only static priority 0 (i.e., threads under real-time policies always have priority over **SCHED_OTHER** processes). **SCHED_OTHER** is the standard Linux time-sharing scheduler that is intended for all threads that do not require the special real-time mechanisms.

The thread to run is chosen from the static priority 0 list based on a *dynamic* priority that is determined only inside this list. The dynamic priority is based on the nice value (see below) and is increased for each time quantum the thread is ready to run, but denied to run by the scheduler. This ensures fair progress among all **SCHED_OTHER** threads.

In the Linux kernel source code, the **SCHED_OTHER** policy is actually named **SCHED_NORMAL**.

The nice value

The nice value is an attribute that can be used to influence the CPU scheduler to favor or disfavor a process in scheduling decisions. It affects the scheduling of **SCHED_OTHER** and **SCHED_BATCH** (see below) processes. The nice value can be modified using **nice**(2), **setpriority**(2), or **sched_setattr**(2).

According to POSIX.1, the nice value is a per-process attribute; that is, the threads in a process should share a nice value. However, on Linux, the nice value is a per-thread attribute: different threads in the same process may have different nice values.

The range of the nice value varies across UNIX systems. On modern Linux, the range is −20 (high priority) to +19 (low priority). On some other systems, the range is −20..20. Very early Linux kernels (Before Linux 2.0) had the range −infinity..15.

The degree to which the nice value affects the relative scheduling of **SCHED_OTHER** processes likewise varies across UNIX systems and across Linux kernel versions.

With the advent of the CFS scheduler in kernel 2.6.23, Linux adopted an algorithm that causes relative differences in nice values to have a much stronger effect. In the current implementation, each unit of difference in the nice values of two processes results in a factor of 1.25 in the degree to which the scheduler favors the higher priority process. This causes very low nice values (+19) to truly provide little CPU to a process whenever there is any other higher priority load on the system, and makes high nice values (−20) deliver most of the CPU to applications that require it (e.g., some audio applications).

On Linux, the **RLIMIT_NICE** resource limit can be used to define a limit to which an unprivileged process's nice value can be raised; see **setrlimit**(2) for details.

For further details on the nice value, see the subsections on the autogroup feature and group scheduling, below.

SCHED_BATCH: Scheduling batch processes

(Since Linux 2.6.16.) **SCHED_BATCH** can be used only at static priority 0. This policy is similar to **SCHED_OTHER** in that it schedules the thread according to its dynamic priority (based on the nice value). The difference is that this policy will cause the scheduler to always assume that the thread is CPU-intensive. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behavior, so that this thread is mildly disfavored in scheduling decisions.

This policy is useful for workloads that are noninteractive, but do not want to lower their nice value, and for workloads that want a deterministic scheduling policy without interactivity causing extra preemptions

(between the workload's tasks).

SCHED_IDLE: Scheduling very low priority jobs

(Since Linux 2.6.23.) **SCHED_IDLE** can be used only at static priority 0; the process nice value has no influence for this policy.

This policy is intended for running jobs at extremely low priority (lower even than a +19 nice value with the **SCHED_OTHER** or **SCHED_BATCH** policies).

Resetting scheduling policy for child processes

Each thread has a reset-on-fork scheduling flag. When this flag is set, children created by **fork**(2) do not inherit privileged scheduling policies. The reset-on-fork flag can be set by either:

* ORing the **SCHED_RESET_ON_FORK** flag into the *policy* argument when calling **sched_setscheduler**(2) (since Linux 2.6.32); or
* specifying the **SCHED_FLAG_RESET_ON_FORK** flag in *attr.sched_flags* when calling **sched_setattr**(2).

Note that the constants used with these two APIs have different names. The state of the reset-on-fork flag can analogously be retrieved using **sched_getscheduler**(2) and **sched_getattr**(2).

The reset-on-fork feature is intended for media-playback applications, and can be used to prevent applications evading the **RLIMIT_RTTIME** resource limit (see **getrlimit**(2)) by creating multiple child processes.

More precisely, if the reset-on-fork flag is set, the following rules apply for subsequently created children:

* If the calling thread has a scheduling policy of **SCHED_FIFO** or **SCHED_RR**, the policy is reset to **SCHED_OTHER** in child processes.
* If the calling process has a negative nice value, the nice value is reset to zero in child processes.

After the reset-on-fork flag has been enabled, it can be reset only if the thread has the **CAP_SYS_NICE** capability. This flag is disabled in child processes created by **fork**(2).

Privileges and resource limits

In Linux kernels before 2.6.12, only privileged (**CAP_SYS_NICE**) threads can set a nonzero static priority (i.e., set a real-time scheduling policy). The only change that an unprivileged thread can make is to set the **SCHED_OTHER** policy, and this can be done only if the effective user ID of the caller matches the real or effective user ID of the target thread (i.e., the thread specified by *pid*) whose policy is being changed.

A thread must be privileged (**CAP_SYS_NICE**) in order to set or modify a **SCHED_DEADLINE** policy.

Since Linux 2.6.12, the **RLIMIT_RTPRIO** resource limit defines a ceiling on an unprivileged thread's static priority for the **SCHED_RR** and **SCHED_FIFO** policies. The rules for changing scheduling policy and priority are as follows:

* If an unprivileged thread has a nonzero **RLIMIT_RTPRIO** soft limit, then it can change its scheduling policy and priority, subject to the restriction that the priority cannot be set to a value higher than the maximum of its current priority and its **RLIMIT_RTPRIO** soft limit.
* If the **RLIMIT_RTPRIO** soft limit is 0, then the only permitted changes are to lower the priority, or to switch to a non-real-time policy.
* Subject to the same rules, another unprivileged thread can also make these changes, as long as the effective user ID of the thread making the change matches the real or effective user ID of the target thread.
* Special rules apply for the **SCHED_IDLE** policy. In Linux kernels before 2.6.39, an unprivileged thread operating under this policy cannot change its policy, regardless of the value of its **RLIMIT_RTPRIO** resource limit. In Linux kernels since 2.6.39, an unprivileged thread can switch to either the **SCHED_BATCH** or the **SCHED_OTHER** policy so long as its nice value falls within the range permitted by its **RLIMIT_NICE** resource limit (see **getrlimit**(2)).

Privileged (**CAP_SYS_NICE**) threads ignore the **RLIMIT_RTPRIO** limit; as with older kernels, they can make arbitrary changes to scheduling policy and priority. See **getrlimit**(2) for further information on

RLIMIT_RTPRIO.

Limiting the CPU usage of real-time and deadline processes

A nonblocking infinite loop in a thread scheduled under the **SCHED_FIFO**, **SCHED_RR**, or **SCHED_DEADLINE** policy can potentially block all other threads from accessing the CPU forever. Prior to Linux 2.6.25, the only way of preventing a runaway real-time process from freezing the system was to run (at the console) a shell scheduled under a higher static priority than the tested application. This allows an emergency kill of tested real-time applications that do not block or terminate as expected.

Since Linux 2.6.25, there are other techniques for dealing with runaway real-time and deadline processes. One of these is to use the **RLIMIT_RTTIME** resource limit to set a ceiling on the CPU time that a real-time process may consume. See **getrlimit**(2) for details.

Since version 2.6.25, Linux also provides two */proc* files that can be used to reserve a certain amount of CPU time to be used by non-real-time processes. Reserving CPU time in this fashion allows some CPU time to be allocated to (say) a root shell that can be used to kill a runaway process. Both of these files specify time values in microseconds:

/proc/sys/kernel/sched_rt_period_us

This file specifies a scheduling period that is equivalent to 100% CPU bandwidth. The value in this file can range from 1 to **INT_MAX**, giving an operating range of 1 microsecond to around 35 minutes. The default value in this file is 1,000,000 (1 second).

/proc/sys/kernel/sched_rt_runtime_us

The value in this file specifies how much of the "period" time can be used by all real-time and deadline scheduled processes on the system. The value in this file can range from −1 to **INT_MAX**−1. Specifying −1 makes the run time the same as the period; that is, no CPU time is set aside for non-real-time processes (which was the Linux behavior before kernel 2.6.25). The default value in this file is 950,000 (0.95 seconds), meaning that 5% of the CPU time is reserved for processes that don't run under a real-time or deadline scheduling policy.

Response time

A blocked high priority thread waiting for I/O has a certain response time before it is scheduled again. The device driver writer can greatly reduce this response time by using a "slow interrupt" interrupt handler.

Miscellaneous

Child processes inherit the scheduling policy and parameters across a **fork**(2). The scheduling policy and parameters are preserved across **execve**(2).

Memory locking is usually needed for real-time processes to avoid paging delays; this can be done with **mlock**(2) or **mlockall**(2).

The autogroup feature

Since Linux 2.6.38, the kernel provides a feature known as autogrouping to improve interactive desktop performance in the face of multiprocess, CPU-intensive workloads such as building the Linux kernel with large numbers of parallel build processes (i.e., the **make**(1) **−j** flag).

This feature operates in conjunction with the CFS scheduler and requires a kernel that is configured with **CONFIG_SCHED_AUTOGROUP**. On a running system, this feature is enabled or disabled via the file */proc/sys/kernel/sched_autogroup_enabled*; a value of 0 disables the feature, while a value of 1 enables it. The default value in this file is 1, unless the kernel was booted with the *noautogroup* parameter.

A new autogroup is created when a new session is created via **setsid**(2); this happens, for example, when a new terminal window is started. A new process created by **fork**(2) inherits its parent's autogroup membership. Thus, all of the processes in a session are members of the same autogroup. An autogroup is automatically destroyed when the last process in the group terminates.

When autogrouping is enabled, all of the members of an autogroup are placed in the same kernel scheduler "task group". The CFS scheduler employs an algorithm that equalizes the distribution of CPU cycles across task groups. The benefits of this for interactive desktop performance can be described via the following example.

Suppose that there are two autogroups competing for the same CPU (i.e., presume either a single CPU system or the use of **taskset**(1) to confine all the processes to the same CPU on an SMP system). The first group contains ten CPU-bound processes from a kernel build started with *make −j10*. The other contains a single CPU-bound process: a video player. The effect of autogrouping is that the two groups will each receive half of the CPU cycles. That is, the video player will receive 50% of the CPU cycles, rather than just 9% of the cycles, which would likely lead to degraded video playback. The situation on an SMP system is more complex, but the general effect is the same: the scheduler distributes CPU cycles across task groups such that an autogroup that contains a large number of CPU-bound processes does not end up hogging CPU cycles at the expense of the other jobs on the system.

A process's autogroup (task group) membership can be viewed via the file */proc/[pid]/autogroup*:

```
$ cat /proc/1/autogroup
/autogroup-1 nice 0
```

This file can also be used to modify the CPU bandwidth allocated to an autogroup. This is done by writing a number in the "nice" range to the file to set the autogroup's nice value. The allowed range is from +19 (low priority) to −20 (high priority). (Writing values outside of this range causes **write**(2) to fail with the error **EINVAL**.)

The autogroup nice setting has the same meaning as the process nice value, but applies to distribution of CPU cycles to the autogroup as a whole, based on the relative nice values of other autogroups. For a process inside an autogroup, the CPU cycles that it receives will be a product of the autogroup's nice value (compared to other autogroups) and the process's nice value (compared to other processes in the same autogroup.

The use of the **cgroups**(7) CPU controller to place processes in cgroups other than the root CPU cgroup overrides the effect of autogrouping.

The autogroup feature groups only processes scheduled under non-real-time policies (**SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE**). It does not group processes scheduled under real-time and deadline policies. Those processes are scheduled according to the rules described earlier.

The nice value and group scheduling

When scheduling non-real-time processes (i.e., those scheduled under the **SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE** policies), the CFS scheduler employs a technique known as "group scheduling", if the kernel was configured with the **CONFIG_FAIR_GROUP_SCHED** option (which is typical).

Under group scheduling, threads are scheduled in "task groups". Task groups have a hierarchical relationship, rooted under the initial task group on the system, known as the "root task group". Task groups are formed in the following circumstances:

* All of the threads in a CPU cgroup form a task group. The parent of this task group is the task group of the corresponding parent cgroup.
* If autogrouping is enabled, then all of the threads that are (implicitly) placed in an autogroup (i.e., the same session, as created by **setsid**(2)) form a task group. Each new autogroup is thus a separate task group. The root task group is the parent of all such autogroups.
* If autogrouping is enabled, then the root task group consists of all processes in the root CPU cgroup that were not otherwise implicitly placed into a new autogroup.
* If autogrouping is disabled, then the root task group consists of all processes in the root CPU cgroup.
* If group scheduling was disabled (i.e., the kernel was configured without **CONFIG_FAIR_GROUP_SCHED**), then all of the processes on the system are notionally placed in a single task group.

Under group scheduling, a thread's nice value has an effect for scheduling decisions *only relative to other threads in the same task group*. This has some surprising consequences in terms of the traditional semantics of the nice value on UNIX systems. In particular, if autogrouping is enabled (which is the default in various distributions), then employing **setpriority**(2) or **nice**(1) on a process has an effect only for

scheduling relative to other processes executed in the same session (typically: the same terminal window).

Conversely, for two processes that are (for example) the sole CPU-bound processes in different sessions (e.g., different terminal windows, each of whose jobs are tied to different autogroups), *modifying the nice value of the process in one of the sessions has no effect* in terms of the scheduler's decisions relative to the process in the other session. A possibly useful workaround here is to use a command such as the following to modify the autogroup nice value for *all* of the processes in a terminal session:

```
$ echo 10 > /proc/self/autogroup
```

Real-time features in the mainline Linux kernel

Since kernel version 2.6.18, Linux is gradually becoming equipped with real-time capabilities, most of which are derived from the former *realtime-preempt* patch set. Until the patches have been completely merged into the mainline kernel, they must be installed to achieve the best real-time performance. These patches are named:

patch-*kernelversion*-rt*patchversion*

and can be downloaded from ⟨http://www.kernel.org/pub/linux/kernel/projects/rt/⟩.

Without the patches and prior to their full inclusion into the mainline kernel, the kernel configuration offers only the three preemption classes **CONFIG_PREEMPT_NONE**, **CONFIG_PREEMPT_VOLUNTARY**, and **CONFIG_PREEMPT_DESKTOP** which respectively provide no, some, and considerable reduction of the worst-case scheduling latency.

With the patches applied or after their full inclusion into the mainline kernel, the additional configuration item **CONFIG_PREEMPT_RT** becomes available. If this is selected, Linux is transformed into a regular real-time operating system. The FIFO and RR scheduling policies are then used to run a thread with true real-time priority and a minimum worst-case scheduling latency.

NOTES

The **cgroups**(7) CPU controller can be used to limit the CPU consumption of groups of processes.

Originally, Standard Linux was intended as a general-purpose operating system being able to handle background processes, interactive applications, and less demanding real-time applications (applications that need to usually meet timing deadlines). Although the Linux kernel 2.6 allowed for kernel preemption and the newly introduced O(1) scheduler ensures that the time needed to schedule is fixed and deterministic irrespective of the number of active tasks, true real-time computing was not possible up to kernel version 2.6.17.

SEE ALSO

chcpu(1), **chrt**(1), **lscpu**(1), **ps**(1), **taskset**(1), **top**(1), **getpriority**(2), **mlock**(2), **mlockall**(2), **munlock**(2), **munlockall**(2), **nice**(2), **sched_get_priority_max**(2), **sched_get_priority_min**(2), **sched_getaffinity**(2), **sched_getparam**(2), **sched_getscheduler**(2), **sched_rr_get_interval**(2), **sched_setaffinity**(2), **sched_setparam**(2), **sched_setscheduler**(2), **sched_yield**(2), **setpriority**(2), **pthread_getaffinity_np**(3), **pthread_getschedparam**(3), **pthread_setaffinity_np**(3), **sched_getcpu**(3), **capabilities**(7), **cpuset**(7)

Programming for the real world − POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0.

The Linux kernel source files *Documentation/scheduler/sched-deadline.txt*, *Documentation/scheduler/sched-rt-group.txt*, *Documentation/scheduler/sched-design-CFS.txt*, and *Documentation/scheduler/sched-nice-design.txt*

COLOPHON

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.