NAME

sched_setscheduler, sched_getscheduler − set and get scheduling policy/parameters

SYNOPSIS

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);

int sched_getscheduler(pid_t pid);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

DESCRIPTION

sched_setscheduler() sets both the scheduling policy and the associated parameters for the process whose ID is specified in *pid*. If *pid* equals zero, the scheduling policy and parameters of the calling process will be set. The interpretation of the argument *param* depends on the selected policy. Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies:

SCHED_OTHER
the standard round-robin time-sharing policy;

SCHED_BATCH
for "batch" style execution of processes; and

SCHED_IDLE for running *very* low priority background jobs.

The following "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable processes are selected for execution:

SCHED_FIFO a first-in, first-out policy; and

SCHED_RR a round-robin policy.

The semantics of each of these policies are detailed below.

sched_getscheduler() queries the scheduling policy currently applied to the process identified by *pid*. If *pid* equals zero, the policy of the calling process will be retrieved.

Scheduling Policies

The scheduler is the kernel component that decides which runnable process will be executed by the CPU next. Each process has an associated scheduling policy and a *static* scheduling priority, *sched_priority*; these are the settings that are modified by **sched_setscheduler**(). The scheduler makes it decisions based on knowledge of the scheduling policy and static priority of all processes on the system.

For processes 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 processes always have higher priority than normal processes.) Note well: POSIX.1-2001 only requires an implementation to support 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 processes for each possible *sched_priority* value.

In order to determine which process runs next, the scheduler looks for the non-empty list with the highest static priority and selects the process at the head of this list.

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

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

SCHED_FIFO: First In-First Out scheduling

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

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

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

* A call to **sched_setscheduler**() or **sched_setparam**(2) will put the **SCHED_FIFO** (or **SCHED_RR**) process identified by *pid* at the start of the list if it was runnable. As a consequence, it may preempt the currently running process if it has the same priority. (POSIX.1-2001 specifies that the process should go to the end of the list.)

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

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

A **SCHED_FIFO** process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, 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 process is only allowed to run for a maximum time quantum. If a **SCHED_RR** process 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** process that has been preempted by a higher priority process and subsequently resumes execution as a running process 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_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can only be used at static priority 0. **SCHED_OTHER** is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms. The process 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 (set by **nice**(2) or **setpriority**(2)) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all **SCHED_OTHER** processes.

SCHED_BATCH: Scheduling batch processes

(Since Linux 2.6.16.) **SCHED_BATCH** can only be used at static priority 0. This policy is similar to **SCHED_OTHER** in that it schedules the process 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 process is CPU-intensive. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behaviour, so that this process is mildly disfavored in scheduling decisions.

This policy is useful for workloads that are non-interactive, 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 only be used 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).

Privileges and resource limits

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

Since Linux 2.6.12, the **RLIMIT_RTPRIO** resource limit defines a ceiling on an unprivileged process'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 process has a non-zero **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 process can also make these changes, as long as the effective user ID of the process making the change matches the real or effective user ID of the target process.

* Special rules apply for the **SCHED_IDLE**: an unprivileged process operating under this policy cannot change its policy, regardless of the value of its **RLIMIT_RTPRIO** resource limit.

Privileged (**CAP_SYS_NICE**) processes 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**.

Response time

A blocked high priority process waiting for the 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).

Since a non-blocking infinite loop in a process scheduled under **SCHED_FIFO** or **SCHED_RR** will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected. See also the description of the **RLIMIT_RTTIME** resource limit in **getrlimit**(2).

POSIX systems on which **sched_setscheduler**() and **sched_getscheduler**() are available define **_POSIX_PRIORITY_SCHEDULING** in *<unistd.h>*.

RETURN VALUE

On success, **sched_setscheduler**() returns zero. On success, **sched_getscheduler**() returns the policy for the process (a non-negative integer). On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EINVAL

The scheduling *policy* is not one of the recognized policies, or *param* does not make sense for the *policy*.

EPERM

The calling process does not have appropriate privileges.

ESRCH

The process whose ID is *pid* could not be found.

CONFORMING TO

POSIX.1-2001 (but see BUGS below). The **SCHED_BATCH** and **SCHED_IDLE** policies are Linux-specific.

NOTES

POSIX.1 does not detail the permissions that an unprivileged process requires in order to call **sched_setscheduler**(), and details vary across systems. For example, the Solaris 7 manual page says that the real or effective user ID of the calling process must match the real user ID or the save set-user-ID of the target process.

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.

Real-time features in the mainline Linux kernel

From kernel version 2.6.18 onwards, however, Linux is gradually becoming equipped with real-time capabilities, most of which are derived from the former *realtime-preempt* patches developed by Ingo Molnar, Thomas Gleixner, Steven Rostedt, and others. Until the patches have been completely merged into the mainline kernel (this is expected to be around kernel version 2.6.30), 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 that can be selected using **sched_setscheduler**() are then used to run a process with true real-time priority and a minimum worst-case scheduling latency.

BUGS

POSIX says that on success, **sched_setscheduler**() should return the previous scheduling policy. Linux **sched_setscheduler**() does not conform to this requirement, since it always returns 0 on success.

SEE ALSO

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_rr_get_interval**(2), **sched_setaffinity**(2), **sched_setparam**(2), **sched_yield**(2), **setpriority**(2), **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 kernel source file *Documentation/scheduler/sched-rt-group.txt* (since kernel 2.6.25).

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.