**NAME**

priocntl – process scheduler control

**SYNOPSIS**

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);
```

**DESCRIPTION**

**priocntl( )** provides for control over the scheduling of an active light weight process (**LWP**).

LWPs fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the real-time class and the time-sharing class. The characteristics of these classes are described under the corresponding headings below. The class attribute of an LWP is inherited across the **fork**(2), **exec**(2) and **_lwp_create**(2) system calls. **priocntl( )** can be used to dynamically change the class and other scheduling parameters associated with a running LWP or set of LWPs given the appropriate permissions as explained below.

In the default configuration, a runnable real-time LWP runs before any other LWP. Therefore, inappropriate use of real-time LWP can have a dramatic negative impact on system performance.

**priocntl( )** provides an interface for specifying a process, set of processes or an LWP to which the function is to apply. The **priocntlset**(2) system call provides the same functions as **priocntl( )**, but allows a more general interface for specifying the set of LWPs to which the function is to apply.

For **priocntl( )**, the *idtype* and *id* arguments are used together to specify the set of LWPs. The interpretation of *id* depends on the value of *idtype*. The possible values for *idtype* and corresponding interpretations of *id* are as follows:

| | |
|---|---|
| **P_LWPID** | *id* is an LWP ID. The **priocntl ( )** system call applies to the LWP with the specified ID within the calling process. |
| **P_PID** | *id* is a process ID specifying a single process. The **priocntl( )** system call applies to all LWPs currently associated with the specified process. |
| **P_PPID** | *id* is a parent process ID. The **priocntl( )** system call applies to all LWPs currently associated with processes with the specified parent process ID. |
| **P_PGID** | *id* is a process group ID. The **priocntl( )** system call applies to all LWPs currently associated with processes in the specified process group. |
| **P_SID** | *id* is a session ID. The **priocntl( )** system call applies to all LWPs currently associated with processes in the specified session. |
| **P_CID** | *id* is a class ID (returned by **priocntl( ) PC_GETCID** as explained below). The **priocntl( )** system call applies to all LWPs in the specified class. |
| **P_UID** | *id* is a user ID. The **priocntl( )** system call applies to all LWPs with this effective user ID. |

**P_GID** *id* is a group ID. The **priocntl( )** system call applies to all LWPs with this effective group ID.

**P_ALL** The **priocntl( )** system call applies to all existing LWPs. The value of *id* is ignored. The permission restrictions described below still apply.

An *id* value of **P_MYID** can be used in conjunction with the *idtype* value to specify the calling LWP's LWP ID, parent process ID, process group ID, session ID, class ID, user ID, or group ID.

In order to change the scheduling parameters of an LWP (using the **PC_SETPARMS** command as explained below) the real or effective user ID of the LWP calling **priocntl( )** must match the real or effective user ID of the receiving LWP or the effective user ID of the calling LWP must be super-user. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting LWPs to that class and/or when setting class-specific scheduling parameters.

A special **sys** scheduling class exists for the purpose of scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any LWP to **sys**. In addition, any processes in the **sys** class that are included in a specified set of processes are disregarded by **priocntl( )**. For example, an *idtype* of **P_UID** and an *id* value of zero would specify all processes with a user ID of zero except processes in the **sys** class and (if changing the parameters using **PC_SETPARMS**) the **init**(1M) process.

The **init** process is a special case. In order for a **priocntl( )** call to change the class or other scheduling parameters of the **init** process (process ID 1), it must be the only process specified by *idtype* and *id*. The **init** process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices may be highly undesirable; see the *System Administration Guide*, *Volume II* for more information.)

The data type and value of *arg* are specific to the type of command specified by *cmd*.

A structure with the following members is used by the **PC_GETCID** and **PC_GETCLINFO** commands.

```
id_t    pc_cid;                 /* Class id */
char    pc_clname[PC_CLNMSZ];   /* Class name */
long    pc_clinfo[PC_CLINFOSZ]; /* Class information */
```

**pc_cid** is a class ID returned by **priocntl( ) PC_GETCID**. **pc_clname** is a buffer of size **PC_CLNMSZ** (defined in **<sys/priocntl.h>**) used to hold the class name (**RT** for real-time or **TS** for time-sharing).

**pc_clinfo** is a buffer of size **PC_CLINFOSZ** (defined in **<sys/priocntl.h>**) used to return data describing the attributes of a specific class. The format of this data is class-specific and is described under the appropriate heading (REAL-TIME CLASS or TIME-SHARING CLASS) below.

A structure with the following elements is used by the **PC_SETPARMS** and **PC_GETPARMS** commands.

```
id_t    pc_cid;                        /* LWP class */
long    pc_clparms[PC_CLPARMSZ];       /* Class-specific params */
```

**pc_cid** is a class ID (returned by **priocntl( ) PC_GETCID**). The special class ID **PC_CLNULL** can also be assigned to **pc_cid** when using the **PC_GETPARMS** command as explained below.

The **pc_clparms** buffer holds class-specific scheduling parameters. The format of this parameter data for a particular class is described under the appropriate heading below. **PC_CLPARMSZ** is the length of the **pc_clparms** buffer and is defined in **<sys/priocntl.h>**.

**Commands**

Available **priocntl( )** commands are:

**PC_GETCID**

Get class ID and class attributes for a specific class given class name. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. The **pc_clname** buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in **pc_cid**, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl( )** call returns the total number of classes configured in the system (including the **sys** class). If the class specified by **pc_clname** is invalid or is not currently configured the **priocntl( )** call returns –1 with **errno** set to **EINVAL**. The format of the attribute data returned for a given class is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl( )** call still returns the number of configured classes.

**PC_GETCLINFO**

Get class name and class attributes for a specific class given class ID. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. **pc_cid** is the class ID of the class whose attributes you are getting.

On success, the class name is returned in the **pc_clname** buffer, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl( )** call returns the total number of classes configured in the system (including the **sys** class). The format of the attribute data returned for a given class is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl( )** call still returns the number of configured classes.

**PC_SETPARMS**
Set the class and class-specific scheduling parameters of the specified LWP(s) associated with the specified process(es). When this command is used with the *idtype* of P_LWPID, it will set the class and class-specific scheduling parameters of the LWP. *arg* points to a structure of type **pcparms_t**. **pc_cid** specifies the class you are setting and the **pc_clparms** buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header and described under the appropriate class heading below.

When setting parameters for a set of LWPs, **priocntl( )** acts on the LWPs in the set in an implementation-specific order. If **priocntl( )** encounters an error for one or more of the target processes, it may or may not continue through the set of LWPs, depending on the nature of the error. If the error is related to permissions (**EPERM**), **priocntl( )** continues through the LWP set, resetting the parameters for all target LWPs for which the calling LWP has appropriate permissions. **priocntl( )** then returns –1 with **errno** set to **EPERM** to indicate that the operation failed for one or more of the target LWPs. If **priocntl( )** encounters an error other than permissions, it does not continue through the set of target LWPs but returns the error immediately.

**PC_GETPARMS**
Get the class and/or class-specific scheduling parameters of an LWP. *arg* points to a structure of type **pcparms_t**.

If **pc_cid** specifies a configured class and a single LWP belonging to that class is specified by the *idtype* and *id* values or the **procset** structure, then the scheduling parameters of that LWP are returned in the **pc_clparms** buffer. If the LWP specified does not exist or does not belong to the specified class, the **priocntl( )** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** specifies a configured class and a set of LWPs is specified, the scheduling parameters of one of the specified LWP belonging to the specified class are returned in the **pc_clparms** buffer and the **priocntl( )** call returns the process ID of the selected LWP. The criteria for selecting an LWP to return in this case is class dependent. If none of the specified LWPs exist or none of them belong to the specified class the **priocntl( )** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** is **PC_CLNULL** and a single LWP is specified the class of the specified LWP is returned in **pc_cid** and its scheduling parameters are returned in the **pc_clparms** buffer.

**PC_ADMIN**
This command provides functionality needed for the implementation of the **dispadmin**(1M) command. It is not intended for general use by other applications.

**REAL-TIME CLASS**

The real-time class provides a fixed priority preemptive scheduling policy for those LWPS requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the real-time class is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable real-time LWP is given CPU service before any LWP belonging to any other class.

The real-time class has a range of real-time priority (**rt_pri**) values that may be assigned to an LWP within the class. Real-time priorities range from 0 to *x*, where the value of *x* is configurable and can be determined for a specific installation by using the **priocntl( ) PC_GETCID** or **PC_GETCLINFO** command.

The real-time scheduling policy is a fixed priority policy. The scheduling priority of a real-time LWP is never changed except as the result of an explicit request by the user/application to change the **rt_pri** value of the LWP.

For an LWP in the real-time class, the **rt_pri** value is, for all practical purposes, equivalent to the scheduling priority of the LWP. The **rt_pri** value completely determines the scheduling priority of a real-time LWP relative to other LWPs within its class. Numerically higher **rt_pri** values represent higher priorities. Since the real-time class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable real-time LWP with the highest **rt_pri** value is always selected to run before any other LWPs in the system.

In addition to providing control over priority, **priocntl( )** provides for control over the length of the time quantum allotted to the LWP in the real-time class. The time quantum value specifies the maximum amount of time an LWP may run assuming that it does not complete or enter a resource or event wait state (**sleep**). Note that if another LWP becomes runnable at a higher priority, the currently running LWP may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable real-time LWPs on a set of scheduling queues. There is a separate queue for each configured real-time priority and all real-time LWPs with a given **rt_pri** value are kept together on the appropriate queue. The LWPs on a given queue are ordered in FIFO order (that is, the LWP at the front of the queue has been waiting longest for service and receives the CPU first). Real-time LWPs that wake up after sleeping, LWPs which change to the real-time class from some other class, LWPs which have used their full time quantum, and runnable LWPs whose priority is reset by **priocntl( )** are all placed at the back of the appropriate queue for their priority. An LWP that is preempted by a higher priority LWP remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other LWP at this priority. Following a **fork**(2) or **_lwp_create**(2) system call by a real-time LWP, the parent LWP continues to run while the child LWP (which inherits its parent's **rt_pri** value) is placed at the back of the queue.

A structure with the following members (defined in **<sys/rtpriocntl.h>**) defines the format used for the attribute data for the real-time class.

```
short     rt_maxpri;     /* Maximum real-time priority */
```

The **priocntl( ) PC_GETCID** and **PC_GETCLINFO** commands return real-time class attributes in the **pc_clinfo** buffer in this format.

**rt_maxpri** specifies the configured maximum **rt_pri** value for the real-time class (if **rt_maxpri** is *x*, the valid real-time priorities range from 0 to *x*).

A structure with the following members (defined in **<sys/rtpriocntl.h>**) defines the format used to specify the real-time class-specific scheduling parameters of an LWP.

```
short   rt_pri;      /* Real-Time priority */
ulong   rt_tqsecs;   /* Seconds in time quantum */
long    rt_tqnsecs;  /* Additional nanoseconds in quantum */
```

When using the **priocntl( ) PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the real-time class, the data in the **pc_clparms** buffer is in this format.

The above commands can be used to set the real-time priority to the specified value or get the current **rt_pri** value. Setting the **rt_pri** value of an LWP that is currently running or runnable (not sleeping) causes the LWP to be placed at the back of the scheduling queue for the specified priority. The LWP is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous **rt_pri** value of the LWP. Note that a running LWP can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its **rt_pri** value to its current real-time priority value. In order to change the time quantum of an LWP without setting the priority or affecting the LWP's position on the queue, the **rt_pri** field should be set to the special value **RT_NOCHANGE** (defined in **<sys/rtpriocntl.h>**). Specifying **RT_NOCHANGE** when changing the class of an LWP to real-time from some other class results in the real-time priority being set to zero.

For the **priocntl( ) PC_GETPARMS** command, if **pc_cid** specifies the real-time class and more than one real-time LWP is specified, the scheduling parameters of the real-time LWP with the highest **rt_pri** value among the specified LWPs are returned and the LWP ID of this LWP is returned by the **priocntl( )** call. If there is more than one LWP sharing the highest priority, the one returned is implementation-dependent.

The **rt_tqsecs** and **rt_tqnsecs** fields are used for getting or setting the time quantum associated with an LWP or group of LWPs. **rt_tqsecs** is the number of seconds in the time quantum and **rt_tqnsecs** is the number of additional nanoseconds in the quantum. For example setting **rt_tqsecs** to 2 and **rt_tqnsecs** to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the **rt_tqnsecs** field results in an error return with **errno** set to **EINVAL**. Although the resolution of the **tq_nsecs** field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. The maximum time quantum that can be specified is implementation-specific and equal to **LONG_MAX** ticks (defined in **<limits.h>**). Requesting a quantum greater than this maximum results in an error return with **errno** set to **ERANGE** (although infinite quantums may be requested using a special value as explained below). Requesting a time quantum of zero (setting both **rt_tqsecs** and **rt_tqnsecs** to 0) results in an error return with **errno** set to **EINVAL**.

The **rt_tqnsecs** field can also be set to one of the following special values (defined in **<sys/rtpriocntl.h>**), in which case the value of **rt_tqsecs** is ignored.

| | |
|---|---|
| **RT_TQINF** | Set an infinite time quantum. |
| **RT_TQDEF** | Set the time quantum to the default for this priority (see **rt_dptbl**(4)). |

**RT_NOCHANGE** Do nott set the time quantum. This value is useful when you wish to change the real-time priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to real-time from some other class is equivalent to specifying **RT_TQDEF**.

In order to change the class of an LWP to real-time (from any other class) the LWP invoking **priocntl( )** must have super-user privileges. In order to change the priority or time quantum setting of a real-time LWP, the LWP invoking **priocntl( )** must have super-user privileges or must itself be a real-time LWP whose real or effective user ID matches the real of effective user ID of the target LWP.

The real-time priority and time quantum are inherited across the **fork**(2) and **exec**(2) system calls.

**TIME-SHARING CLASS**

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among LWPs with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive LWPs and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see **ts_upri** below) values that may be assigned to LWPs within the class. A **ts_upri** value of zero is defined as the default base priority for the time-sharing class. User priorities range from $-x$ to $+x$ where the value of $x$ is configurable and can be determined for a specific installation by using the **priocntl( ) PC_GETCID** or **PC_GETCLINFO** command.

The purpose of the user priority is to provide some degree of user/application control over the scheduling of LWPs in the time-sharing class. Raising or lowering the **ts_upri** value of an LWP in the time-sharing class raises or lowers the scheduling priority of the LWP. It is not guaranteed, however, that an LWP with a higher **ts_upri** value will run before one with a lower **ts_upri** value. This is because the **ts_upri** value is just one factor used to determine the scheduling priority of a time-sharing LWP. The system may dynamically adjust the internal scheduling priority of a time-sharing LWP based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the **PC_GETCID** and **PC_GETCLINFO** commands) there is a per LWP user priority limit (see **ts_uprilim** below), which specifies the maximum **ts_upri** value that may be set for a given LWP; by default, **ts_uprilim** is zero.

A structure with the following members (defined in **<sys/tspriocntl.h>**) defines the format used for the attribute data for the time-sharing class.

```
short    ts_maxupri;    /* Limits of user priority range */
```

The **priocntl( ) PC_GETCID** and **PC_GETCLINFO** commands return time-sharing class attributes in the **pc_clinfo** buffer in this format.

**ts_maxupri** specifies the configured maximum user priority value for the time-sharing class. If **ts_maxupri** is $x$, the valid range for both user priorities and user priority limits is from $-x$ to $+x$.

A structure with the following members (defined in **<sys/tspriocntl.h>**) defines the format used to specify the time-sharing class-specific scheduling parameters of an LWP.

```
short    ts_uprilim;    /* Time-Sharing user priority limit */
short    ts_upri;       /* Time-Sharing user priority */
```

When using the **priocntl( ) PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the time-sharing class, the data in the **pc_clparms** buffer is in this format.

For the **priocntl( ) PC_GETPARMS** command, if **pc_cid** specifies the time-sharing class and more than one time-sharing LWP is specified, the scheduling parameters of the time-sharing LWP with the highest **ts_upri** value among the specified LWPs is returned and the LWP ID of this LWP is returned by the **priocntl( )** call. If there is more than one LWP sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing LWP may lower its own **ts_uprilim** (or that of another LWP with the same user ID). Only a time-sharing LWP with super-user privileges may raise a **ts_uprilim**. When changing the class of an LWP to time-sharing from some other class, super-user privileges are required in order to set the initial **ts_uprilim** to a value greater than zero. Attempts by a non-super-user LWP to raise a **ts_uprilim** or set an initial **ts_uprilim** greater than zero fail with a return value of –1 and **errno** set to **EPERM**.

Any time-sharing LWP may set its own **ts_upri** (or that of another LWP with the same user ID) to any value less than or equal to the LWP's **ts_uprilim**. Attempts to set the **ts_upri** above the **ts_uprilim** (and/or set the **ts_uprilim** below the **ts_upri**) result in the **ts_upri** being set equal to the **ts_uprilim**.

Either of the **ts_uprilim** or **ts_upri** fields may be set to the special value **TS_NOCHANGE** (defined in **<sys/tspriocntl.h>**) in order to set one of the values without affecting the other. Specifying **TS_NOCHANGE** for the **ts_upri** when the **ts_uprilim** is being set to a value below the current **ts_upri** causes the **ts_upri** to be set equal to the **ts_uprilim** being set. Specifying **TS_NOCHANGE** for a parameter when changing the class of an LWP to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the **ts_uprilim** is **0** and the default for the **ts_upri** is to set it equal to the **ts_uprilim** which is being set.

The time-sharing user priority and user priority limit are inherited across the **fork** and **exec** functions.

## RETURN VALUES

Unless otherwise noted above, **priocntl( )** returns a value of 0 on success. **priocntl( )** returns –1 on failure and sets **errno** to indicate the error.

## ERRORS

**priocntl( )** fails if one or more of the following are true :

**EAGAIN** An attempt to change the class of an LWP failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

**EFAULT** One of the arguments points to an illegal address.

**EINVAL** The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.

| | |
|---|---|
| **ENOMEM** | An attempt to change the class of an LWP failed because of insufficient memory. |
| **EPERM** | The effective user of the calling LWP is not super-user. |
| **ERANGE** | The requested time quantum is out of range. |
| **ESRCH** | None of the specified LWPs exist. |

**SEE ALSO**

**priocntl**(1), **dispadmin**(1M), **init**(1M), **_lwp_create**(2), **exec**(2), **fork**(2), **nice**(2), **priocntlset**(2), **rt_dptbl**(4)

*System Interfaces Guide*