# SUSE Linux Enterprise Real Time 10 SP1 Quick Start

## SUSE Linux Enterprise 10 SP1

NOVELL® QUICK START CARD

SUSE Linux Enterprise Real Time is an add-on to SUSE® Linux Enterprise that allows you to run tasks which require deterministic real-time processing, in a SUSE Linux Enterprise environment. SUSE Linux Enterprise Real Time meets this requirement by offering several different options for CPU and IO scheduling, CPU shielding and setting CPU affinities to processes.

## Installing SUSE Linux Enterprise Real Time

There are two ways to set up SUSE Linux Enterprise Real Time:

- Install it as an add-on product when installing the SUSE Linux Enterprise Server 10 SP1.
- Install it on top of an already installed SUSE Linux Enterprise Server 10 SP1.

SUSE Linux Enterprise Real Time always needs a SUSE Linux Enterprise Server SP1 base, it cannot be installed in standalone mode. Refer to the SUSE Linux Enterprise Server *Installation and Administration* manual, Section "Installing Add-On Products" at `http://www.novell.com/documentation/sles10/sles_admin/index.html?page=/documentation/sles10/sles_admin/data/sec_yast2_sw.html` to learn more about installing add-on products.

The following sections provide a brief introduction to the tools and possibilities of SUSE Linux Enterprise Real Time.

## Using CPU Sets

In some circumstances, it is beneficial to be able to run specific tasks only on defined CPUs. For this reason, the linux kernel provides a feature called cpuset. Cpusets provide the means to do a so called "soft partitioning" of the system. Dedicated CPUs, together with some predefined memory, work on a number of tasks.

All systems have at least one cpuset that is called `/`. To retrieve the cpuset of a specific task with a certain process id *`pid`*, use the command `cat /proc/pid/cpuset`. To add, remove, or manage cpusets, a special file system with file system type `cpuset` is available. Before you can use this file system type, mount it to `/dev/cpuset` with the following commands:

```
mkdir /dev/cpuset
mount -t cpuset none /dev/cpuset
```

Every cpuset has the following entries:

cpus
: A list of CPUs available for the current cpuset. Ranges of CPUs are displayed with a dash between the first and the last CPU, else CPUs are represented by a comma separated list of CPU numbers.

mems
: A list of memory nodes available to the current cpuset.

memory_migrate
: This flag determines if memory pages should be moved to the new configuration, in case the memory configuration of the cpuset changes.

cpu_exclusive
: Defines if this cpuset becomes a scheduling domain, that shares properties and policies.

mem_exclusive
: Determines if userspace tasks in this cpuset can only get their memory from the memory assigned to this cpuset.

tasks
: Contains the process ids of all tasks running in this cpuset.

notify_on_release
: If this is set to 1, `/sbin/cpuset_release_agent` will be called when the last process leaves this cpuset. Note, that it is up to the administrator to create a script or binary that matches the local needs.

memory_pressure
: Provides the means to determine how often a cpuset is running short of memory. Only calculated if `memory_pressure_enabled` is enabled in the top cpuset.

memory_spread_page and memory_spread_slab
: Determines if file system buffers and I/O buffers are uniformly spread across the cpuset.

In addition to these entries, the top cpuset also contains the entry `memory_pressure_enabled`, which must be set to `1` if you want to make use of the `memory_pressure` entries in the different cpusets.

In order to make use of cpusets, you need detailed hardware information for several reasons: on big machines, memory that is local to a CPU will be much faster than memory that is only available on a different node. If you want to create cpusets from several nodes, you should try to combine CPUs that are close together. Otherwise, task switches and memory access may slow down your system noticeably.

To find out which node a CPU belongs to, use the `/sys` file system. The kernel provides information about available CPUs to a specific node by creating links in `/sys/devices/system/node/nodeX/`.

If several CPUs are to be combined to a cpuset, check the distance of the CPUs from each other with the command `numactl --hardware`. This command is available after installing the package `numactl`.

The actual configuration and manipulation of cpusets is done by modifying the file system below `/dev/cpuset`. Tasks are performed in the following way:

Create a Cpuset
: To create a cpuset with the name `exampleset`, just run `mkdir /dev/cpuset/exampleset` to create the respective directory. The newly created set will contain several entries that reflect the current status of the set.

Remove a Cpuset
: To remove a cpuset, you only need to remove the cpuset directory. For example, use `rmdir /dev/cpuset/exampleset` to remove the previously generated cpuset named `exampleset`. In contrast to normal file systems, this works even if there are still entries in the directory.

  Note that you will get an error like `rmdir: exampleset: Device or resource busy`, if there are still tasks active in that set. To remove these tasks from the set, just move them to another set.

Add CPUs to a Cpuset
: To add CPUs to a set, you may either specify a comma separated list of CPU numbers, or give a range of CPUs. For example, to add CPUs with the numbers `2`, `3` and `7` to `exampleset`, you can use one of the following commands: `/bin/echo 2,3,7 > /dev/cpuset/exampleset/cpus` or `/bin/echo 2-3,7 > /dev/cpuset/exampleset/cpus`.

Add Memory to a Cpuset
: You cannot move tasks to a cpuset without giving the cpuset access to some system memory. To do so, echo a node number into `/dev/cpuset/exampleset/mems`. If possible, use a node that is close to the used CPUs in this set.

Moving Tasks to Cpusets
: A cpuset is just a useless structure, unless it handles some tasks. To add a task to `/dev/cpuset/exampleset/`, simply echo the task number into `/dev/cpuset/exampleset/`. The following script moves all user space processes to `/dev/cpuset/exampleset/` and leaves all kernel threads untouched:

```
cd /dev/cpuset/exampleset; \
for pid in $(cat ../tasks); do \
test -e /proc/$pid/exe && \
echo $pid > tasks; done
```

  Note, that for a clean solution, you would have to stop all processes, move them to the new cpuset, and let them continue afterward. Otherwise, the process may finish before the *for loop* finishes, or other processes may start during moving.

  This loop liberates all CPUs not contained in the exampleset from all processes. Check the result with the command `cat /dev/cpuset/tasks`, which then should not have any entries.

  Of course, you can move all tasks from a special cpuset to the top level set, if you intend to remove this special cpuset.

Automatically Remove Unused Cpusets
: In case a cpuset is not used any longer by any process, you might want to clean up such unused cpusets automatically. To initialize the removal, you can use the `notify_on_release` flag. If this is set to `1`, the kernel will run `/sbin/cpuset_release_agent` when the last process exits. To remove an unused script, you may,

for example, add the following script in `/sbin/cpuset_release_agent`:

```
#!/bin/sh
logger cpuset: releasing $1
rmdir /dev/cpuset/$1
```

After adding the script to your system, run `chmod 755 /sbin/cpuset_release_agent` to make the script executable.

Determine the Cpuset of a Specific Process

All processes with the process id `PID` have an entry in `/proc/PID/cpuset`. If you run the command `cat /proc/PID/cpuset` on a PID that runs in the cpuset `exampleset`, you will find the results in `/exampleset`.

## Specifying a CPU Affinity with `taskset`

The default behavior of the kernel, is to keep a process running on the same CPU, if the system load is balanced over the available CPUs. Otherwise, the kernel tries to improve the load balancing by moving processes to an idling CPU. In some situations, however, it is desirable to set a CPU affinity for a given process. In this case, the kernel will not move the process away from the selected CPUs. For example, if you use shielding, the shielded CPUs will not run any process that does not have an affinity to the shielded CPUs. Another possibility is to run all low priority tasks on a selected CPU to remove load from the other CPUs.

Note, that if a task is running inside a specific cpuset, the affinity mask must match at least one of the CPUs available in this set. The `taskset` command will not move a process outside the cpuset it is running in.

To set or retrieve the CPU affinity of a task, a bitmask is used, that is represented by a hexadecimal number. If you count the bits of this bitmask, the lowest bit represents the first logical CPU as they are found in `/proc/cpuinfo`. For example:

`0x00000001`
: is processor #0.

`0x00000002`
: is processor #1.

`0x00000003`
: is processor #0 and processor #1.

`0xFFFFFFFE`
: all but the first CPU.

If a given mask does not contain any valid CPU on the system, an error is returned. If taskset returns without an error, the given program has been scheduled to the specified list of CPUs.

The command `taskset` can either be used to start a new process with a given CPU affinity, or to redefine the CPU affinity of a already running process.

Examples

`taskset -p` *`pid`*
: Retrieves the current CPU affinity of the process with PID `pid`.

`taskset -p` *`mask pid`*
: Sets the CPU affinity of the process with PID `pid` to `mask`.

`taskset` *`mask command`*
: Runs `command` with a CPU affinity of `mask`.

## Changing I/O Priorities with `ionice`

Handling I/O is one of the critical issues for all high-performance systems. If a task has lots of CPU power available, but must wait for the disk, it will not work as efficient as it could. The Linux kernel provides three different scheduling classes to determine the I/O handling for a process. All of these classes can be fine-tuned with a nice level.

The *Best Effort* Scheduler

The *Best Effort* scheduler is the default I/O scheduler, and is used for all processes that do not specify a different I/O scheduler class. By default, this scheduler sets its niceness according to the nice value of the running process.

There are eight different nice levels available for this scheduler. The lowest priority is represented by a nice level of seven, the highest priority is zero.

This scheduler has the scheduling class number `2`.

The *Real Time* Scheduler

The real-time I/O class always gets the highest priority for disk access. The other schedulers will only be served, if no real-time request is present. This scheduling class may easily lock up the system if not implemented with care.

The real-time scheduler defines nice levels just like the *Best Effort* scheduler.

This scheduler has the scheduling class number `1`.

The *Idle* Scheduler

The *Idle* scheduler does not define any nice levels. I/O is only done in this class, if no other scheduler runs an I/O request. This scheduler has the lowest available priority and can be used for processes that are not time-critical at all.

This scheduler has the scheduling class number `3`.

To change I/O schedulers and nice values, use the `ionice` command. This provides a means to tune the scheduler of

already running processes, or to start new processes with specific I/O settings.

Examples

`ionice -c3 -p$$`
: Sets the scheduler of the current shell to `Idle`.

`ionice`
: Without additional parameters, this prints the I/O scheduler settings of the current shell.

`ionice -c1 -p42 -n2`
: Sets the scheduler of the process with process id `42` to `Real Time`, and its nice value to `2`.

`ionice -c3 /bin/bash`
: Starts the Bash shell with the `Idle` I/O scheduler.

## Changing the I/O Scheduler for Block Devices

The Linux kernel provides several block device schedulers that can be selected individually for each block device. All but the `noop` scheduler perform a kind of ordering of requested blocks to reduce head movements on the hard disk. If you use an external storage system that has its own scheduler, you may want to disable the Linux internal reordering by selecting the `noop` scheduler.

The Linux I/O Schedulers

noop
: The *noop* scheduler is a very simple scheduler, that performs basic merging and sorting on I/O requests. This scheduler is mainly used for specialized environments that run their own schedulers optimized for the used hardware, such as storage systems or hardware RAID controllers.

anticipatory
: The main principle of *anticipatory* scheduling is, that after a read, the scheduler simply expects further reads from userspace. For this reason, after a read completes, the anticipatory scheduler will do nothing at all for a few milliseconds, giving userspace the possibility to ask for another read. If such a read is requested, it will be performed immediately. Otherwise the scheduler continues with doing writes after a short time-out.

  The advantage of this procedure is a major reduction of seeks and thus a decreased read latency. This also increases read and write bandwidth.

deadline
: The main point of *deadline* scheduling is to try hard to answer a request before a given deadline. This results in very good I/O for a random single I/O in real-time environments.

  In principle, the *deadline* uses two lists with all requests. One is sorted by block sequences to reduce seeking latencies, the other is sorted by expire times for each request. Normally, requests are served according to the block sequence, but if a request reaches its deadline, the scheduler starts to work on this request.

cfq
: The *Completely Fair Queuing* scheduler uses a separate I/O queue for each process. All of these queues get a similar time slice for disk access. With this procedure, the *CFQ* tries to divide the bandwidth evenly between all requesting processes. This scheduler has a similar throughput as the *anticipatory* scheduler, but the maximum latency is much shorter.

  For the average system, this scheduler yields the best results, and thus is the default I/O scheduler on SUSE Linux Enterprise systems.

To print the current scheduler of a block device like `/dev/sda`, use the following command:

```
cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

In this case, the scheduler for `/dev/sda` is set to `cfq`, the `Completely Fair Queuing` scheduler. This is the default scheduler on SUSE Linux Enterprise Real Time.

To change the schedulers, echo one of the names `noop`, `anticipatory`, `deadline`, or `cfq` into `/sys/block/<device>/scheduler`. For example, if you want to set the I/O scheduler of the device `/dev/sda` to `noop`, use the command `echo "noop" > /sys/block/sda/scheduler`. To set other variables in the `/sys` file system, use a similar approach.

## Tuning the Block Device I/O Scheduler

All schedulers, except for the *noop* scheduler, have several common parameters that may be tuned for each block device. You can access these parameters with `sysfs` in the `/sys/block/<device>/queue/iosched/` directory. The following parameters are tuneable for the respective scheduler:

Anticipatory Scheduler

`antic_expire`
: Time in milliseconds that the *anticipatory* scheduler waits for another read request close to the last read request performed. The *anticipatory* scheduler will not wait for upcoming read requests, if this value is set to zero.

`read_expire`
: Deadline of a read request in milliseconds. This scheduler also controls the interval between expired requests. By default, read_expire is set to 125 milliseconds. Until a read request is served which is next on the list, it can thus take up to 250 milliseconds.

`write_expire`
: Similar to `read_expire` for write requests.

`read_batch_expire`
: If write requests are scheduled, this is the time in milliseconds that reads are served before pending writes get a time slice. If writes are more important than reads, set this value lower than `read_expire`.

`write_batch_expire`
: Similar to `read_batch_expire` for write requests.

Deadline Scheduler

`read_expire`
: The main focus of this scheduler is to limit the start latency for a request to a given time. Therefore, for each request, a deadline is calculated from the current time plus the value of `read_expire` in milliseconds.

`write_expire`
: Similar to `read_expire` for write requests.

`fifo_batch`
: If a request hits its deadline, it is necessary to move the request from the sorted I/O scheduler list to the dispatch queue. The variable `fifo_batch` controls how many requests are moved, depending on the cost of each request.

`front_merges`
: The scheduler normally tries to find contiguous I/O requests and merges them. There are two kinds of merges: The new I/O request may be in front of the existing I/O request (front merge), or it may follow behind the existing request (back merge). Because most merges are back merges, you can disable the front merge functionality by setting `front_merges` to `0`.

`write_starved`
: In case some read or write requests hit their deadline, the scheduler prefers the read requests by default. To prevent write requests from being postponed forever, the variable `write_starved` controls how often read requests are preferred until write requests are preferred over read requests.

CFQ Scheduler

`back_seek_max` and `back_seek_penalty`
: The *CFQ* scheduler normally uses a strict ascending elevator. When needed, it also allows small backward seeks, but it puts some penalty on them. The maximum backward sector seek is defined with `back_seek_max`, and the multiplier for the penalty is set by `back_seek_penalty`.

`fifo_expire_async` and `fifo_expire_sync`
: The `fifo_expire_*` variables define the timeout in milliseconds for asynchronous and synchronous I/O requests. Typically, `fifo_expire_async` affects write and `fifo_expire_sync` affects both read and write operations.

`quantum`
: Defines the number of I/O requests to dispatch when the block device is idle.

`slice_async`, `slice_async_rq`, `slice_sync`, and `slice_idle`
: These variables define the time slices a block device gets for synchronous or asynchronous operations.

  - `slice_async` and `slice_sync` represent the length of an asynchronous or synchronous disk slice in milliseconds.
  - `slice_async_rq` defines for how many requests an asynchronous disk slice lasts.
  - `slice_idle` defines how long a sync slice may idle.

## For More Information

A lot of information about real-time implementations and administration can be found on the Internet. The following list contains a number of selected links:

- The cpuset feature of the kernel is explained in `/usr/src/linux/Documentation/cpusets.txt`. More detailed documentation is available from `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/linux/bks/SGI_Admin/books/LX_Resource_AG/sgi_html/ch04.html`, `http://www.bullopensource.org/cpuset/`, and `http://lwn.net/Articles/127936/`.
- An overview of CPU and I/O schedulers available in Linux can be found at `http://aplawrence.com/Linux/linux26_features.html`.
- Detailed information about the anticipatory I/O scheduler is available at `http://www.cs.rice.edu/~ssiyer/r/antsched/antio.html` and `http://www.cs.rice.edu/~ssiyer/r/antsched/`.
- For more information about the deadline I/O scheduler, refer to `http://lwn.net/2002/0110/a/io-scheduler.php3`, or `http://kerneltrap.org/node/431`. In your installed system, find further information in `/usr/src/linux/Documentation/block/deadline-iosched.txt`.
- The CFQ I/O scheduler is covered in detail in `http://en.wikipedia.org/wiki/CFQ`.
- General information about I/O scheduling in Linux is available at `http://lwn.net/Articles/101029/`, `http://lwn.net/Articles/114273/`, and `http://donami.com/118`.
- A lot of information about real-time can be found at `http://linuxdevices.com/articles/AT6476691775.html`.

# Legal Notice