Cpuset Library and Linux Kernel Support

This document describes the 'C' library **libcpuset** interface to Linux cpusets.

Cpusets provide system-wide control of the CPUs on which tasks may execute, and the memory nodes on which they allocate memory. Each cpuset defines a list of allowed CPUs and memory nodes, and each process in the system is attached to a cpuset. Cpusets are represented in a hierarchical virtual file system. Cpusets can be nested and they have file-like permissions.

The efficient administration of large multi-processor systems depends on dynamically allocating portions of the systems CPU and memory resources to different users and purposes. The optimum performance of NUMA systems depends optimizing CPU and memory placement of critical applications, and minimizing interference between applications. Cpusets provides a convenient means to control such CPU and memory placement and usage.

Author: Paul Jackson

Address: pj@sgi.com

Date: 14 November 2006

This document is written using the outline processor Leo, and version controlled using CSSC. It is rendered using Python Docutils on reStructuredText extracted from Leo, directly into both html and LaTeX. The LaTeX is converted into pdf using the pdflatex utility. The html is converted into plain text using the lynx utility.

Table of Contents

1 Why Cpusets?

The essential purpose of cpusets is to provide CPU and memory containers or "soft partitions" within which to run sets of related tasks.

On an SMP (multiple CPU) system without some means of CPU placement, any task can run on any CPU. On a NUMA (multiple memory node) system, any memory page can be allocated on any node. This can cause both poor cache locality and poor memory access times, substantially reducing performance and run-time repeatability. By restraining all other jobs from using any of the CPUs or memory nodes assigned to critical jobs, interference with critical jobs can be minimized.

For example, some multi-threaded high performance computing (HPC) jobs consist of a number of threads that communicate via message passing interfaces (MPI) and other such jobs rely on multi-platform shared-memory parallel programming (OpenMP) that can tightly couple parallel computation threads using special language directives. It is common that threads in such jobs need to be executing at the same time to make optimum progress. In such cases, if a single thread loses a CPU, all threads stop making forward progress and spin at a barrier. Cpusets can eliminate the need for a gang scheduler, provide isolation of one such job from other tasks on a system, and facilitate providing equal resources to each thread in such a job. This results in both optimum and repeatable performance.

This document focuses on the 'C' API provided by the user level **libcpuset** library. This library depends on the following Linux 2.6 kernel facilities:

- sched_setaffinity (for CPU binding),
- mbind and set_mempolicy (for memory node binding), and
- kernel cpusets support.

The sched_setaffinity, mbind and set_mempolicy calls enable specifying the CPU and memory placement for individual tasks. On smaller or limited use systems, these calls may be sufficient.

The kernel cpuset facility provides additional support for system wide management of CPU and memory resources, by related sets of tasks. It provides a hierarchical structure to the resources, with file system like name-space and permissions, and support for guaranteed exclusive use of resources.

The Linux kernel provides the following support for cpusets:

- Each task has a link to a cpuset structure that specifies the CPUs and memory nodes available for its use.
- A hook in the sched_setaffinity and mbind system calls ensures that any requested CPU or memory node is available in that tasks cpuset.

- Tasks sharing the same placement constraints reference the same cpuset.
- These kernel cpusets are arranged in a hierarchical virtual file system, reflecting the possible nesting of "soft partitions".
- The kernel task scheduler is constrained to only schedule a task on the CPUs in that task's cpuset.
- The kernel memory allocator is constrained to only allocate physical memory to a task from the memory nodes in that tasks cpuset.
- The kernel memory allocator provides an economical per-cpuset metric of the aggregate memory pressure (frequency of requests for a free memory page not easily satisfied by an available free page) of the tasks in a cpuset (see the per-cpuset 'memory_pressure' file.)
- The kernel memory allocator provides the option to request that memory pages used for file I/O (the kernel page cache) and associated kernel data structures for file inodes and directories be evenly spread across all the memory nodes in a cpuset, rather than preferentially allocated on whatever memory node the task that first accessed the page was first running (see the per-cpuset 'memory_spread_page' and 'memory_spread_slab' files).
- The memory migration facility in the kernel can be controlled using per-cpuset files, so that when the memory nodes allowed to a task by cpusets changes, any pages it had on no longer allowed nodes are migrated to nodes now allowed.

A cpuset constrains the jobs (set of related tasks) running in it to a subset of the systems memory and CPUs. They enable administrators and system service software to:

- Create and delete named cpusets.
- Decide which CPUs and memory nodes are available to a cpuset.
- Attach a task to a particular cpuset.
- Identify all tasks sharing the same cpuset.
- Exclude any other cpuset from overlapping a given cpuset, giving the tasks running in that cpuset exclusive use of those CPUs and memory nodes.
- Perform bulk operations on all tasks associated with a cpuset, such as varying the resources available to that cpuset, or hibernating those tasks in temporary favor of some other job.
- Perform sub-partitioning with hierarchical permissions and resource management.

Cpusets are exposed by the kernel to user space by mounting the `cpuset` virtual file system (VFS) at `/dev/cpuset`, rather than by additional system calls. Such a VFS is a natural way to represent nested resource allocations and the associated hierarchical permission model.

Within a single cpuset, other facilities such as `dplace`, first-touch memory placement, pthreads, `sched_setaffinity` and `mbind` can be used to manage processor and memory placement to a more fine-grained level.

There is a single set of kernel mechanisms that supports all these facilities and provides a consistent processor and memory model regardless of what mix of utilities and API's you use to manage it. This provides a consistent execution model for all users.

2 Linux Cpuset Kernel Support

Several developers in the Open Source community provided the Linux 2.6 kernel support for CPU and memory placement, including the following:

- Robert Love - `Tech9, Novell` (USA) - Scheduler attributes such as CPU affinity
- Simon Derr - `Bull` (France) - CPU placement, core architecture and file system interface to cpusets
- Andi Kleen - `SUSE` (Germany) - NUMA memory placement, mbind and mempolicy
- Paul Jackson - `SGI` (USA) - kernel bitmask improvements, kernel cpuset integration, **libbitmask**, **libcpuset**

At least a couple of command line utilities have been developed that use these affinity calls to allow placing a process on a specific CPU. Robert Love has a package *schedutils* with a command `taskset`. The `numactl` command in Andi Kleen's work has options to run a specified command on specified CPUs and memory nodes.

Andi Kleen led a session at the 2003 Kernel Summit in Ottawa in NUMA memory management, and has been developing a NUMA library and kernel support for memory placement. Minutes from that session are available at: http://lwn.net/Articles/40626/.

The cpuset kernel facility and file system for Linux 2.6 kernels is based on the work of Simon Derr, with integration and refinements by Paul Jackson. In addition this document of the **libcpuset** user library, the kernel cpuset facility is documented in the kernel source file Documentation/cpusets.txt, and Simon Derr has provided a document of cpusets at http://www.bullopensource.org/cpuset/.

The user level bitmask library supports convenient manipulation of multi-word bitmasks useful for CPUs and memory nodes. This bitmask library is required by and designed to work with the cpuset library. The design notes of **libbitmask** are available in a separate document, Bitmask_Library.html or Bitmask_Library.pdf.

Unlike `sched_setaffinity()` and `mbind()`, which are implemented as additional kernel system calls, the primary kernel interface for accessing the cpuset facility is The Cpuset File System, usually mounted at `/dev/cpuset`. The cpuset library **libcpuset** provides convenient access to these facilities from 'C' programs.

Cpusets extend the usefulness of the Linux 2.6 kernel mechanisms `sched_setaffinity()` for CPU placement, and `mbind()` and `set_mempolicy()` for memory placement. On smaller or dedicated use systems, these other mechanisms are often sufficient. The **libcpuset** library provides a convenient API to these other mechanisms that has the added advantage of being robustly adapting to memory migration.

On larger NUMA systems, running more than one, performance critical, job, it is necessary to be able to manage jobs in their entirety. This includes providing a job with exclusive CPU and memory that no other job can use and being able to list all tasks currently in a cpuset.

You can use both these other placement mechanisms and cpusets together, using the Advanced Cpuset Library Functions to manage overall job placement, and using the other mechanisms, perhaps via the Basic Cpuset Library Functions within each given job to manage the details of thread and memory page placement.

2.1 The Cpuset File System

Cpusets are named, nested sets of CPUs and memory nodes. Each cpuset is represented by a directory in the cpuset virtual file system, normally mounted at `/dev/cpuset`.

New cpusets are created using the `mkdir` system call or shell command. The properties of a cpuset, such as its flags, allowed CPUs and memory nodes, and attached tasks, are queried and modified by reading or writing to the appropriate file in that cpusets directory.

The state of each cpuset is represented by small text files in that cpusets directory. These files may be read and written using traditional shell utilities such as `cat(1)` and `echo(1)`, or using ordinary file access routines from programmatic languages, such as `open(2)`, `read(2)`, `write(2)` and `close(2)` from the 'C' library.

These per-cpuset files represent internal kernel state and do not have any persistent image on disk. These files are automatically created when the cpuset is created, as a result of the `mkdir` invocation. It is not allowed to add or remove files from a cpuset directory.

Each of these per-cpuset files is listed and described below:

tasks: List of the process ID's (PIDs) of the tasks in that cpuset. The list is formatted as a series of ASCII decimal numbers, each followed by a newline. A task may be added to a cpuset (removing it from the cpuset previously containing it) by writing its PID to that cpusets `tasks` file (with or without

a trailing newline.)

Beware that only one PID may be written to the `tasks` file at a time. If a string is written that contains more than one PID, all but the first will be ignored.

notify_on_release: Flag (0 or 1). If set (1), then that cpuset will receive special handling whenever its last using task and last child cpuset goes away. For more information, see the *Notify On Release* section, below.

cpus: List of CPUs on which tasks in that cpuset are allowed to execute. See List Format below for a description of the format of `cpus`.

The CPUs allowed to a cpuset may be changed by writing a new list to its `cpus` file. Note however, such a change does not take affect until the PIDs of the tasks in the cpuset are rewritten to the cpusets `tasks` file.

cpu_exclusive: Flag (0 or 1). If set (1), then the cpuset has exclusive use of its CPUs (no sibling or cousin cpuset may overlap CPUs). By default this is off (0). Newly created cpusets also initially default this to off (0).

mems: List of memory nodes on which tasks in that cpuset are allowed to allocate memory. See List Format below for a description of the format of `mems`.

mem_exclusive: Flag (0 or 1). If set (1), then the cpuset has exclusive use of its memory nodes (no sibling or cousin may overlap). By default this is off (0). Newly created cpusets also initially default this to off (0).

memory_migrate: Flag (0 or 1). If set (1), then memory migration is enabled. For more information, see the *Memory Migration* section, below.

memory_pressure: A measure of how much memory pressure the tasks in this cpuset are causing. For more information, see the *Memory Pressure* section, below. Always has value zero (0) unless `memory_pressure_enabled` is enabled in the top cpuset. This file is read-only.

memory_pressure_enabled: Flag (0 or 1). This file is only present in the root cpuset, normally `/dev/cpuset`. If set (1), then `memory_pressure` calculations are enabled for all cpusets in the system. For more information, see the *Memory Pressure* section, below.

memory_spread_page: Flag (0 or 1). If set (1), then the kernel page cache (file system buffers) are uniformly spread across the cpuset. For more information, see the *Memory Spread* section, below.

memory_spread_slab: Flag (0 or 1). If set (1), then the kernel slab caches for file i/o (directory and inode structures) are uniformly spread across the cpuset. For more information, see the *Memory Spread* section, below.

In addition one new file type is added to the `/proc` file system:

/proc/<pid>/cpuset: For each task (pid), list its cpuset path, relative to the root of the cpuset file system. This file is read-only.

Finally, the two control fields actually used by the kernel scheduler and memory allocator to constrain scheduling and allocation to the allowed CPUs are exposed as two more fields in the `status` file of each task:

/proc/<pid>/status:

`Cpus_allowed:` bit vector of CPUs on which this task may be scheduled.

`Mems_allowed:` bit vector of memory nodes on which this task may obtain memory.

There are several reasons why a tasks `Cpus_allowed` and `Mems_allowed` values may differ from the `cpus` and `mems` that are allowed in its current cpuset, as follows:

A. A task might use `sched_setaffinity`, `mbind` or `set_mempolicy` to restrain its placement to less than its cpuset.

B. Various temporary changes to `Cpus_allowed` are done by kernel internal code.

C. Attaching a task to a cpuset doesn't change its `Mems_allowed` until the next time that task needs kernel memory.

D. Changing a cpusets `cpus` doesn't change the `Cpus_allowed` of the tasks attached to it until those tasks are reattached to that cpuset (to avoid a hook in the hotpath scheduler code in the kernel).

E. If hotplug is used to remove all the CPUs, or all the memory nodes, in a cpuset, then the tasks attached to that cpuset will have their `Cpus_allowed` or `Mems_allowed` altered to the CPUs or memory nodes of the closest ancestor to that cpuset that is not empty.

Beware of items D and E, above. Due to item D, user space action is required to update a tasks `Cpus_allowed` after changing its cpuset. Use the routine cpuset_reattach to perform this update after a changing the `cpus` allowed to a cpuset.

Due to item E, the confines of a cpuset can be **violated** after a hotplug removal that empties a cpuset. To avoid having a cpuset without CPU or memory resources, update your system's cpuset configuration to reflect the new hardware configuration. The kernel prefers misplacing a task, over starving a task of essential compute resources.

There is one other condition under which the confines of a cpuset may be violated. A few kernel critical internal memory allocation requests, marked `GFP_ATOMIC`, must be satisfied immediately. The kernel may drop some request or malfunction if one of these allocations fail. If such a request cannot be satisfied within the current tasks cpuset, then the kernel relaxes the cpuset, and looks for memory anywhere it can find it. It's better to violate the cpuset than stress the kernel.

New cpusets are created using `mkdir` (at the shell or in C). Old ones are removed using `rmdir`. The above files are accessed using `read(2)` and `write(2)` system calls, or shell commands such as `cat(1)` and `echo(1)`.

The CPUs and memory nodes in a given cpuset are always a subset of its parent. The root cpuset has all possible CPUs and memory nodes in the system. A cpuset may be exclusive (cpu or memory) only if its parent is similarly exclusive.

Each task has a pointer to a cpuset. Multiple tasks may reference the same cpuset. Requests by a task, using the `sched_setaffinity(2)` system call to include CPUs in its CPU affinity mask, and using the `mbind(2)` and `set_mempolicy(2)` system calls to include memory nodes in its memory policy, are both filtered through that tasks cpuset, filtering out any CPUs or memory nodes not in that cpuset. The scheduler will not schedule a task on a CPU that is not allowed in its cpus_allowed vector, and the kernel page allocator will not allocate a page on a node that is not allowed in the requesting tasks mems_allowed vector.

If a cpuset is cpu or mem exclusive, no other cpuset, other than a direct ancestor or descendant, may share any of the same CPUs or memory nodes.

User level code may create and destroy cpusets by name in the cpuset virtual file system, manage the attributes and permissions of these cpusets and which CPUs and memory nodes are assigned to each cpuset, specify and query to which cpuset a task is assigned, and list the task pids assigned to a cpuset.

Cpuset names are limited in length by the kernel's VFS implementation. No single component of a cpuset name may exceed 255 characters, and the full pathname of a cpuset including the `/dev/cpuset` mount point may not exceed 4095 characters in length.

2.2 Exclusive Cpusets

If a cpuset is marked `cpu_exclusive` or `mem_exclusive`, no other cpuset, other than a direct ancestor or descendant, may share any of the same CPUs or memory nodes.

A cpuset that is `cpu_exclusive` has a scheduler (sched) domain associated with it. The sched domain consists of all CPUs in the current cpuset that are not part of any exclusive child cpusets. This ensures that the scheduler load balancing code only balances against the CPUs that are in the sched domain as defined above and not all of the CPUs in the system. This removes any overhead due to load balancing code trying to pull tasks outside of the `cpu_exclusive` cpuset only to be prevented by the `Cpus_allowed` mask of the task.

A cpuset that is `mem_exclusive` restricts kernel allocations for page, buffer, and other data commonly shared by the kernel across multiple users. All cpusets, whether `mem_exclusive` or not, restrict allocations of memory for user space. This enables configuring a system so that several independent jobs can share common kernel data, such as file system pages, while iso-

lating each jobs user allocation in its own cpuset. To do this, construct a large `mem_exclusive` cpuset to hold all the jobs, and construct child, non-mem_exclusive cpusets for each individual job. Only a small amount of typical kernel memory, such as requests from interrupt handlers, is allowed to be taken outside even a `mem_exclusive` cpuset.

2.3 Notify On Release

If the `notify_on_release` flag is enabled (1) in a cpuset, then whenever the last task in the cpuset leaves (exits or attaches to some other cpuset) and the last child cpuset of that cpuset is removed, the kernel runs the command `/sbin/cpuset_release_agent`, supplying the pathname (relative to the mount point of the cpuset file system) of the abandoned cpuset. This enables automatic removal of abandoned cpusets.

The default value of `notify_on_release` in the root cpuset at system boot is disabled (0). The default value of other cpusets at creation is the current value of their parents `notify_on_release` setting.

The command `/sbin/cpuset_release_agent` is invoked, with the name (`/dev/cpuset` relative path) of that cpuset in `argv[1]`. This supports automatic cleanup of abandoned cpusets.

The usual contents of the command `/sbin/cpuset_release_agent` is simply the shell script:

```
#!/bin/sh
rmdir /dev/cpuset/$1
```

By default `notify_on_release` is off (0). Newly created cpusets inherit their `notify_on_release` setting from their parent cpuset.

As with other flag values below, this flag can be changed by writing an ASCII number 0 or 1 (with optional trailing newline) into the file, to clear or set the flag, respectively.

2.4 Memory Pressure

The `memory_pressure` of a cpuset provides a simple per-cpuset metric of the rate that the tasks in a cpuset are attempting to free up in use memory on the nodes of the cpuset to satisfy additional memory requests.

This enables batch schedulers monitoring jobs running in dedicated cpusets to efficiently detect what level of memory pressure that job is causing.

This is useful both on tightly managed systems running a wide mix of submitted jobs, which may choose to terminate or re-prioritize jobs that are trying to use more memory than allowed

on the nodes assigned them, and with tightly coupled, long running, massively parallel scientific computing jobs that will dramatically fail to meet required performance goals if they start to use more memory than allowed to them.

This mechanism provides a very economical way for the batch scheduler to monitor a cpuset for signs of memory pressure. It's up to the batch scheduler or other user code to decide what to do about it and take action.

If the `memory_pressure_enabled` flag in the top cpuset is not set (0), then the kernel does not compute this filter, and the per-cpuset files `memory_pressure` always contain the value zero (0).

If the `memory_pressure_enabled` flag in the top cpuset is set (1), then the kernel computes this filter for each cpuset in the system, and the memory_pressure file for each cpuset reflects the recent rate of such low memory page allocation attempts by tasks in said cpuset.

Reading the `memory_pressure` file of a cpuset is very efficient. The expectation is that batch schedulers can poll these files and detect jobs that are causing memory stress, so that action can be taken to avoid impacting the rest of the system with a job that is trying to aggressively exceed its allowed memory.

Note well: unless enabled by setting `memory_pressure_enabled` in the top cpuset, `memory_pressure` is not computed for any cpuset, and always reads a value of zero.

Why a per-cpuset, running average:

> Because this meter is per-cpuset, rather than per-task or memory region, the system load imposed by a batch scheduler monitoring this metric is sharply reduced on large systems, because a scan of the system-wide tasklist can be avoided on each set of queries.
>
> Because this meter is a running average, instead of an accumulating counter, a batch scheduler can detect memory pressure with a single read, instead of having to read and accumulate results for a period of time.
>
> Because this meter is per-cpuset rather than per-task or memory region, the batch scheduler can obtain the key information, memory pressure in a cpuset, with a single read, rather than having to query and accumulate results over all the (dynamically changing) set of tasks in the cpuset.

A per-cpuset simple digital filter is kept within the kernel, and updated by any task attached to that cpuset, if it enters the synchronous (direct) page reclaim code.

The per-cpuset `memory_pressure` file provides an integer number representing the recent (half-life of 10 seconds) rate of direct page reclaims caused by the tasks in the cpuset, in units of reclaims attempted per second, times 1000.

The kernel computes this value using a single-pole low-pass recursive (IIR) digital filter coded with 32 bit integer arithmetic. The value decays at an exponential rate.

Given the simple 32 bit integer arithmetic used in the kernel to compute this value, this meter works best for reporting page reclaim rates between one per millisecond (msec) and one per 32 (approx) seconds. At constant rates faster than one per msec it maxes out at values just under 1,000,000. At constant rates between one per msec, and one per second it will stabilize to a value N*1000, where N is the rate of events per second. At constant rates between one per second and one per 32 seconds, it will be choppy, moving up on the seconds that have an event, and then decaying until the next event. At rates slower than about one in 32 seconds, it decays all the way back to zero between each event.

2.5 Memory Spread

There are two Boolean flag files per cpuset that control where the kernel allocates pages for the file system buffers and related in kernel data structures. They are called `memory_spread_page` and `memory_spread_slab`.

If the per-cpuset Boolean flag file `memory_spread_page` is set, the kernel will spread the file system buffers (page cache) evenly over all the nodes that the faulting task is allowed to use, instead of preferring to put those pages on the node where the task is running.

If the per-cpuset Boolean flag file `memory_spread_slab` is set, the kernel will spread some file system related slab caches, such as for inodes and directory entries, evenly over all the nodes that the faulting task is allowed to use; instead of preferring to put those pages on the node where the task is running.

The setting of these flags does not affect anonymous data segment or stack segment pages of a task.

By default, both kinds of memory spreading are off, and memory pages are allocated on the node local to where the task is running, except perhaps as modified by the tasks NUMA mempolicy or cpuset configuration, as long as sufficient free memory pages are available.

When new cpusets are created, they inherit the memory spread settings of their parent.

Setting memory spreading causes allocations for the affected page or slab caches to ignore the tasks NUMA mempolicy and be spread instead. Tasks using mbind() or set_mempolicy() calls to set NUMA mempolicies will not notice any change in these calls as a result of their containing tasks memory spread settings. If memory spreading is turned off, the currently specified NUMA mempolicy once again applies to memory page allocations.

Both `memory_spread_page` and `memory_spread_slab` are Boolean flag files. By default they contain “0”, meaning that the feature is off for that cpuset. If a “1” is written to that file, that

turns the named feature on.

This memory placement policy is also known (in other contexts) as round-robin or interleave.

This policy can provide substantial improvements for jobs that need to place thread local data on the corresponding node, but that need to access large file system data sets that need to be spread across the several nodes in the jobs cpuset in order to fit. Without this policy, especially for jobs that might have one thread reading in the data set, the memory allocation across the nodes in the jobs cpuset can become very uneven.

2.6 Memory Migration

Normally, under the default setting (disabled) of `memory_migrate`, once a page is allocated (given a physical page of main memory) that page stays on whatever node it was allocated, as long as it remains allocated. If the cpuset memory placement policy `mems` subsequently changes, currently allocated pages are not moved. If pages are swapped out to disk and back, then on return to main memory, they may be allocated on different nodes, depending on the cpuset `mems` setting in affect at the time the page is swapped back in.

When memory migration is enabled in a cpuset, if the `mems` setting of the cpuset is changed, then any memory page in use by any task in the cpuset that is on a memory node no longer allowed will be migrated to a memory node that is allowed.

Also if a task is moved into a cpuset with `memory_migrate` enabled, any memory pages it uses that were on memory nodes allowed in its previous cpuset, but which are not allowed in its new cpuset, will be migrated to a memory node allowed in the new cpuset.

The relative placement of a migrated page within the cpuset is preserved during these migration operations if possible. For example, if the page was on the second valid node of the prior cpuset, the page will be placed on the second valid node of the new cpuset, if possible.

In order to maintain the cpuset relative position of pages, even pages on memory nodes allowed in both the old and new cpusets may be migrated. For example, if `memory_migrate` is enabled in a cpuset, and that cpusets `mems` file is written, changing it from say memory nodes "4-7", to memory nodes "5-8", then the following page migrations will be done, in order, for all pages in the address space of tasks in that cpuset:

```
First, migrate pages on node 7 to node 8
Second, migrate pages on node 6 to node 7
Third, migrate pages on node 5 to node 6
Fourth, migrate pages on node 4 to node 5
```

In this example, pages on any memory node other than "4-7" will not be migrated. The order

in which nodes are handled in a migration is intentionally chosen so as to avoid migrating memory *to* a node until any migrations *from* that node have first been accomplished.

2.7 Mask Format

The Mask Format is used to represent CPU and memory node bitmasks in the **/proc/<pid>/status** file.

It is hexadecimal, using ASCII characters "0" - "9" and "a" - "f". This format displays each 32-bit word in hex (zero filled), and for masks longer than one word, uses a comma separator between words. Words are displayed in big-endian order most significant first. And hex digits within a word are also in big-endian order.

The number of 32-bit words displayed is the minimum number needed to display all bits of the bitmask, based on the size of the bitmask.

Examples of the Mask Format:

```
00000001                        # just bit 0 set
80000000,00000000,00000000      # just bit 95 set
00000001,00000000,00000000      # just bit 64 set
000000ff,00000000               # bits 32-39 set
00000000,000E3862               # bits 1,5,6,11-13,17-19 set
```

A mask with bits 0, 1, 2, 4, 8, 16, 32 and 64 set displays as "00000001,00000001,00010117". The first "1" is for bit 64, the second for bit 32, the third for bit 16, the fourth for bit 8, the fifth for bit 4, and the "7" is for bits 2, 1 and 0.

2.8 List Format

The List Format is used to represent CPU and memory node bitmasks (sets of CPU and memory node numbers) in the `/dev/cpuset` file system.

It is a comma separated list of CPU or memory node numbers and ranges of numbers, in ASCII decimal.

Examples of the List Format:

```
0-4,9           # bits 0, 1, 2, 3, 4, and 9 set
0-3,7,12-15     # bits 0, 1, 2, 3, 7, 12, 13, 14, and 15 set
```

3 Using Cpusets at the Shell Prompt

There are multiple ways to use cpusets, including:

- They can be queried and changed from a shell prompt, using such command line utilities as `echo`, `cat`, `mkdir` and `ls`.
- They can be queried and changed via the **libcpuset** 'C' programming API. The primary emphasis of this document is on the 'C' API.

This section describes the use of cpusets using shell commands.

One convenient way to learn how cpusets work is to experiment with them at the shell prompt, before doing extensive 'C' coding.

Note that there is one significant difference between these two interfaces.

Modifying the CPUs in a cpuset the shell prompt requires an additional step, due to intentional limitations in the kernel support for cpusets. The cpuset_reattach routine can be used to perform this step when using **libcpuset**. The extra step consists of writing the pid of each task attached to that cpuset back into the cpusets `tasks` file:

> In order to minimize the impact of cpusets on critical kernel code, such as the scheduler, and due to the fact that the kernel does not support one task updating the memory placement of another task directly, the impact on a task of changing its cpuset CPU or memory node placement, or of changing to which cpuset a task is attached, is subtle.
>
> If a cpuset has its memory nodes modified, then for each task attached to that cpuset, the next time that the kernel attempts to allocate a page of memory for that task, the kernel will notice the change in the tasks cpuset, and update its per-task memory placement to remain within the new cpusets memory placement. If the task was using mempolicy MPOL_BIND, and the nodes to which it was bound overlap with its new cpuset, then the task will continue to use whatever subset of MPOL_BIND nodes are still allowed in the new cpuset. If the task was using MPOL_BIND and now none of its MPOL_BIND nodes are allowed in the new cpuset, then the task will be essentially treated as if it was MPOL_BIND bound to the new cpuset (even though its NUMA placement, as queried by get_mempolicy(), doesn't change). If a task is moved from one cpuset to another, then the kernel will adjust the tasks memory placement, as above, the next time that the kernel attempts to allocate a page of memory for that task.
>
> If a cpuset has its CPUs modified, then each task using that cpuset does _not_ change its behavior automatically. In order to minimize the impact on the critical scheduling code in the kernel, tasks will continue to use their prior CPU placement

until they are rebound to their cpuset, by rewriting their pid to the 'tasks' file of their cpuset. If a task had been bound to some subset of its cpuset using the sched_setaffinity() call, the effect of this is lost on the rebinding. The rebound tasks `cpus_allowed` is set to include all `cpus` in the tasks new cpuset. If a task is moved from one cpuset to another, its CPU placement is updated in the same way as if the tasks pid is rewritten to the 'tasks' file of its current cpuset.

In summary, the memory placement of a task whose cpuset is changed is automatically updated by the kernel, on the next allocation of a page for that task, but the processor placement is not updated, until that tasks pid is rewritten to the 'tasks' file of its cpuset. The delay in rebinding a tasks memory placement is necessary because the kernel does not support one task changing another tasks memory placement. The added user level step in rebinding a tasks CPU placement is necessary to avoid impacting the scheduler code in the kernel with a check for changes in a tasks processor placement.

To create a new cpuset and attach the current command shell to it, the steps are:

1) mkdir /dev/cpuset (if not already done)
2) mount -t cpuset cpuset /dev/cpuset (if not already done)
3) Create the new cpuset using `mkdir(1)`.
4) Assign CPUs and memory nodes to the new cpuset.
5) Attach the shell to the new cpuset.

For example, the following sequence of commands will setup a cpuset named "Charlie", containing just CPUs 2 and 3, and memory node 1, and then attach the current shell to that cpuset:

```
mkdir /dev/cpuset
mount -t cpuset cpuset /dev/cpuset
cd /dev/cpuset
mkdir Charlie
cd Charlie
/bin/echo 2-3 > cpus
/bin/echo 1 > mems
/bin/echo $$ > tasks
# The current shell is now running in cpuset Charlie
# The next line should display '/Charlie'
cat /proc/self/cpuset
```

Migrating a job (the set of tasks attached to a cpuset) to different CPUs and memory nodes in the system, including moving the memory pages currently allocated to that job, can be done

as follows. Lets say you want to move the job in cpuset *alpha* (CPUs 4-7 and memory nodes 2-3) to a new cpuset *beta* (CPUs 16-19 and memory nodes 8-9).

1) First create the new cpuset *beta*.
2) Then allow CPUs 16-19 and memory nodes 8-9 in *beta*.
3) Then enable `memory_migration` in *beta*.
4) Then move each task from *alpha* to *beta*.

The following sequence of commands accomplishes this:

```
cd /dev/cpuset
mkdir beta
cd beta
/bin/echo 16-19 > cpus
/bin/echo 8-9 > mems
/bin/echo 1 > memory_migrate
while read i; do /bin/echo $i; done < ../alpha/tasks > tasks
```

The above should move any tasks in *alpha* to *beta*, and any memory held by these tasks on memory nodes 2-3 to memory nodes 8-9, respectively.

Notice that the last step of the above sequence did **not** do:

```
cp ../alpha/tasks tasks     # Doesn't work (ignores all but first task)
```

The **while** loop, rather than the seemingly easier use of the `cp(1)` command, is necessary because only one task PID at a time may be written to the `tasks` file.

The same affect (writing one pid at a time) as the **while** loop can be accomplished more efficiently, in fewer keystrokes and in syntax that works in any shell, but alas more obscurely, by using the **sed -u** [**unbuffered**] option:

```
sed -un p < ../alpha/tasks > tasks
```

4 Cpuset Programming Model

The **libcpuset** programming model for cpusets provides a hierarchical cpuset name space that integrates smoothly with Linux 2.6 kernel support for simple processor and memory placement.

As systems become larger, with more complex memory, processor and bus architectures, the hierarchical cpuset model for managing processor and memory resources will become increasingly important.

The cpuset name space remains visible to all tasks on a system. Once created, a cpuset remains in existence until it is deleted or until the system is rebooted, even if no tasks are currently running in that cpuset.

The key properties of a cpuset are its pathname, the list of which CPUs and memory nodes it contains, and whether the cpuset has exclusive rights to these resources.

Every task (process) in the system is attached to (running inside) a cpuset. Tasks inherit their parents cpuset attachment when forked. This binding of task to a cpuset can subsequently be changed, either by the task itself, or externally from another task, given sufficient authority.

Tasks have their CPU and memory placement constrained to whatever their containing cpuset allows. A cpuset may have exclusive rights to its CPUs and memory, which provides certain guarantees that other cpusets will not overlap.

At system boot, a top level root cpuset is created, which includes all CPUs and memory nodes on the system. The usual mount point of the cpuset file system, and hence the usual file system path to this root cpuset, is `/dev/cpuset`.

Changing the cpuset binding of a task does not by default move the memory the tasks might have currently allocated, even if that memory is on memory nodes no longer allowed in the tasks cpuset. On kernels that support such memory migration, use the **[optional]** *cpuset_migrate* function to move allocated memory as well.

To create a cpuset from 'C' code, one obtains a handle to a new `struct cpuset`, sets the desired attributes via that handle, and issues a `cpuset_create()` to actually create the desired cpuset and bind it to the specified name. One can also list by name what cpusets exist, query and modify their properties, move tasks between cpusets, list what tasks are currently attached to a cpuset, and delete cpusets.

The `cpuset_alloc()` call applies a hidden *undefined* mark to each attribute of the allocated struct cpuset. Calls to the various `cpuset_set*()` routines mark the attribute being set as *defined*. Calls to `cpuset_create()` and `cpuset_modify()` only set the attributes of the cpuset marked *defined*. This is primarily noticable when creating a cpuset. Code in the kernel sets some attributes of new cpusets, such as `memory_spread_page`, `memory_spread_slab` and `notify_on_release`, by default to the value inherited from their parent. Unless the application using `libcpuset` explicitly overrides the setting of these attributes in the struct cpuset, between the calls to `cpuset_alloc()` and `cpuset_create()`, the kernel default settings will prevail. These hidden marks have no noticeable affect when modifying an existing cpuset using the sequence of calls `cpuset_alloc()`, `cpuset_query()`, and `cpuset_modify()`, because the `cpuset_query()` call sets all attributes and marks them *defined*, while reading the attributes from the cpuset.

The names of cpusets in this 'C' library are either relative to the root cpuset mount point (typically `/dev/cpuset`) if the name starts with a slash '/' character or else relative to the current tasks cpuset.

Cpusets can be renamed using the `rename(2)` system call. The per-cpuset files within each cpuset directory cannot be renamed, and the `rename(2)` system call cannot be used to modify the cpuset hierarchy. You cannot change the parent cpuset of a cpuset using `rename(2)`.

Despite its name, the `pid` parameter to various `libcpuset` routines is actually a thread id, and each thread in a threaded group can be attached to a different cpuset. The value returned from a call to `gettid(2)` can be passed in the argument `pid`.

4.1 Permission Model

Cpusets have a permission structure which determines which users have rights to query, modify and attach to any given cpuset. The permissions of a cpuset are determined by the permissions of the special files and directories in the cpuset file system. The cpuset filesystem is normally mounted at `/dev/cpuset`.

The directory representing each cpuset, and the special per-cpuset files in each such directory, both have traditional `Unix` hierarchical file system permissions for read, write and execute (or search), for each of the owning user, the owning group, and all others.

For instance, a task can put itself in some other cpuset (than its current one) if it can write the tasks file for that cpuset (requires execute permission on the encompassing directories and write permission on that tasks file).

Because the cpuset file system is not persistent, changes in permissions, and even the existence of cpusets other than the root cpuset, are not preserved across system reboots.

An additional constraint is applied to requests to place some other task in a cpuset. One task may not attach another to a cpuset unless it would have permission to send that task a signal.

A task may create a child cpuset if it can access and write the parent cpuset directory. It can modify the CPUs or memory nodes in a cpuset if it can access that cpusets directory (execute permissions on the encompassing directories) and write the corresponding `cpus` or `mems` file.

It should be noted, however, that changes to the CPUs of a cpuset do not apply to any task in that cpuset until the task is reattached to that cpuset. If a task can write the cpus file, it should also be able to write the tasks file and might be expected to have permission to reattach the tasks therein (equivalent to permission to send them a signal).

Some utilities and libraries read the cpuset special files of the cpuset in which the calling task is placed, and may be hindered if a task is placed in a cpuset that is not owned by the task's user-id, unless special care is taken to allow that task continued read access to its own

cpuset special files (and search access to the directory containing them). For example, the Message Passing Toolkit's MPI library's placement functionality activated through the use of MPI_DSM_DISTRIBUTE will fail to place ranks correctly if it cannot read the cpuset special files through insufficient permissions on the cpuset directory or the special files within it.

There is one minor difference between the manner in which cpuset path permissions are evaluated by `libcpuset` and the manner in which file system operation permissions are evaluated by direct system calls. System calls that operate on file pathnames, such as the `open(2)` system call, rely on direct kernel support for a tasks current directory. Therefore such calls can successfully operate on files in or below a tasks current directory even if the task lacks search permission on some ancestor directory. Calls in `libcpuset` that operate on cpuset pathnames, such as the `cpuset_query()` call, rely on `libcpuset` internal conversion of all cpuset pathnames to full, root-based paths, so cannot successfully operate on a cpuset unless the task has search permission on all ancestor directories, starting with the usual cpuset mount point (`/dev/cpuset`.)

4.2 Other Placement Mechanisms

The Linux 2.6 kernel supports additional processor and memory placement mechanisms.

- The `sched_setaffinity(2)` and `sched_getaffinity(2)` system calls set and get a process's CPU affinity mask, which determines the set of CPUs on which it is eligible to run. The `taskset(1)` command provides a command line utility for manipulating a process's CPU affinity mask using these system calls. Only CPUs within a tasks cpuset are allowed, but CPUs are numbered using system wide CPU numbers, not cpuset relative numbers.
- The `set_mempolicy(2)`, and `get_mempolicy(2)` system calls set and get a tasks NUMA memory policy for a process and its children, which defines on which nodes memory is allocated for the process. The `numactl(8)` command provides a command line utility and the `libnuma` library provides a 'C' API for manipulating a process's NUMA memory policy using these system calls. Only memory nodes within a tasks cpuset are allowed, but nodes are numbered using system wide node numbers, not cpuset relative numbers.
- The `mbind(2)` system call sets the NUMA memory policy for the pages in a specific range of a tasks virtual address space.

Cpusets are designed to interact cleanly with these other mechanisms, to support for example having a batch scheduler use cpusets to control the CPU and memory placement of various jobs, while within each job, these other mechanisms are used to manage placement in more detail. It is possible for a batch scheduler to change a jobs cpuset placement while preserving the internal CPU affinity and NUMA memory placement policy.

The CPU and memory node placement constraints imposed by cpusets always constrain those of these other mechanisms. You can use these other mechanisms to reduce further the set of CPUs or memory nodes allowed to a task by cpusets, but you can not use these other mechanisms to escape a tasks cpuset confinement.

Calls to `sched_setaffinity(2)` to modify a tasks CPU affinity automatically mask off CPUs that are not allowed by the affected tasks cpuset. If that results in all the requested CPUs being masked off, then the call fails with errno set to `EINVAL`. If some of the requested CPUs are allowed by the tasks cpuset, then the call proceeds as if only the allowed CPUs were requested, silently ignoring the other, unallowed CPUs. If a task is moved to a different cpuset, or if the `cpus` of a cpuset are changed, then the CPU affinity of the affected task or tasks is lost. If a batch scheduler needs to preserve the CPU affinity of the tasks in a job being moved, it should use the `sched_getaffinity(2)` and `sched_setaffinity(2)` calls to save and restore each affected tasks CPU affinity across the move, relative to the cpuset. The **cpu_set_t** mask data type supported by the 'C' library for use with the CPU affinity calls is different then the **libbitmask** bitmasks used by **libcpuset**, so some coding is required to convert between the two, in order to calculate and preserve cpuset relative CPU affinity.

Similar to CPU affinity, calls to modify a tasks NUMA memory policy silently mask off requested memory nodes outside the tasks allowed cpuset, and will fail if that results in requesting an empty set of memory nodes. Unlike CPU affinity, the NUMA memory policy system calls do not support one task querying or modifying another tasks policy. So the kernel automatically handles preserving cpuset relative NUMA memory policy when either a task is attached to a different cpuset, or a cpusets `mems` setting is changed. If the old and new `mems` sets have the same size, then the cpuset relative offset of affected NUMA memory policies is preserved. If the new `mems` is smaller, then the old `mems` relative offsets are folded onto the new `mems`, modulo the size of the new `mems`. If the new `mems` is larger, then just the first `N` nodes are used, where `N` is the size of the old `mems`.

4.3 Cpuset Aware Thread Pinning

If a job intends to use the Other Placement Mechanisms described above, then that job cannot be *guaranteed* safe operation under the control of a batch scheduler if that job might be migrated to different CPUs or memory nodes. This is because these Other Placement Mechanisms use system wide numbering of CPUs and memory nodes, not cpuset relative numbering, and the job might be migrated without its knowledge while it is trying to adjust its placement.

That is, between the point where such an application computes the CPU or memory node on which it wants to place a thread, and the point where it issues the `sched_setaffinity(2)`, `mbind(2)` or `set_mempolicy(2)` call to direct such a placement, the thread might be migrated to a different cpuset, or its cpuset changed to different CPUs or memory nodes, invalidating the CPU or memory node number it just computed.

This potential race condition is not a significant problem for applications that only use these

Other Placement Mechanisms early in the job run for initial placement setup, if the job is only migrated by a batch scheduler after it has been running for a while.

This **libcpuset** library provides the following mechanisms to support cpuset relative thread placement that is robust even if the job is being concurrently migrated such as by a batch scheduler.

If a job needs to pin a thread to a single CPU, then it can use the convenient cpuset_pin function. This is the most common case.

If a job needs to implement some other variation of placement, such as to specific memory nodes, or to more than one CPU, then it can use the following functions to safely guard such code from placement changes caused by job migration:

- cpuset_get_placement
- cpuset_equal_placement
- cpuset_free_placement

The **libcpuset** functions cpuset_c_rel_to_sys_cpu and variations provide a convenient means to convert between system wide and cpuset relative CPU and memory node numbering.

4.4 Safe Job Migration

Jobs that make proper use of Cpuset Aware Thread Pinning, rather than the unsafe Other Placement Mechanisms, can be safely migrated to a different cpuset, or have their cpuset's CPUs or memory nodes safely changed, without destroying the per-thread placement done within the job.

A batch scheduler can safely migrate jobs while preserving per-thread placement of a job that is concurrently using Cpuset Aware Thread Pinning.

A batch scheduler can accomplish this with the following steps:

1. Suspend the tasks in the job, perhaps by sending their process group a SIGSTOP.
2. Use the cpuset_init_pidlist and related pidlist functions to determine the list of tasks in the job.
3. Use `sched_getaffinity(2)` to query the CPU affinity of each task in the job.
4. Create a new cpuset, under a temporary name, with the new desired CPU and memory placement.
5. Invoke cpuset_migrate_all to move the jobs tasks from the old cpuset to the new.

6. Use cpuset_delete to delete the old cpuset.
7. Use `rename(2)` on the `/dev/cpuset` based path of the new temporary cpuset to rename that cpuset to the to the old cpuset name.
8. Convert the results of the previous `sched_getaffinity(2)` calls to the new cpuset placement, preserving cpuset relative offset by using the cpuset_c_rel_to_sys_cpu and related functions.
9. Use `sched_setaffinity(2)` to reestablish the per-task CPU binding of each thread in the job.
10. Resume the tasks in the job, perhaps by sending their process group a SIGCONT.

The `sched_getaffinity(2)` and `sched_setaffinity(2)` 'C' library calls are limited by 'C' library internals to systems with 1024 CPUs or fewer. To write code that will work on larger systems, one should use the `syscall(2)` indirect system call wrapper to directly invoke the underlying system call, bypassing the 'C' library API for these calls. Perhaps in the future the **libcpuset** library will provide functions that make it easier for a batch scheduler to obtain, migrate, and set a tasks CPU affinity.

The suspend and resume are required in order to keep tasks in the job from changing their per thread CPU placement between step 3 and step 6. The kernel automatically migrates the per-thread memory node placement during step 4, which it has to as there is no way for one task to modify the NUMA memory placement policy of another task. But the kernel does not automatically migrate the per-thread CPU placement, as that can be handled by a user level process such as a batch scheduler doing the migration, as above.

Migrating a job from a larger cpuset (more CPUs or nodes) to a smaller cpuset will lose placement information, and subsequently moving that cpuset back to a larger cpuset will not recover that information. Such migrations lose track of information on a jobs placement. This loss of information of the jobs CPU affinity can be avoided as described above, using `sched_getaffinity(2)` and `sched_setaffinity(2)` to save and restore the placement (affinity) across such a pair of moves. This loss of information of the jobs NUMA memory placement cannot be avoided because one task (the one doing the migration) cannot save nor restore the NUMA memory placement policy of another. So if a batch scheduler wants to migrate jobs without causing them to lose their `mbind(2)` or `set_mempolicy(2)` placement, it should only migrate to cpusets with at least as many memory nodes as the original cpuset.

5 CPUs and Memory Nodes

As of this writing, the Linux kernel NUMA support presumes that there are some memory nodes and some CPUs, and that for each CPU, there is exactly one preferred or local memory

node. Frequently, multiple (two or four perhaps) CPUs will be local to the same memory node. Some memory nodes have no local CPUs - these are called headless nodes.

However, this is not the only possible architecture, and architectures are constantly changing, usually toward the more complex. Driven by nonstop increases in logic density for a half century now, the bus, cache, CPU, memory and storage hierarchy of large systems continues to evolve.

This cpuset interface should remain stable over a long period of time, and be usable over a variety of system architectures. So this interface *avoids* presuming that there is exactly one memory node local to each CPU. Rather it just presumes that there are some CPUs and memory nodes, that some zero or more memory nodes are local to each CPU, and that some zero or more CPUs are local to each memory node.

This interface provides the functions cpuset_localmems to identify, for a CPU, which memory node(s) are local to that CPU, and cpuset_localcpus to identify for a memory node, which CPU(s) are local. Applications using this interface can explicitly specify just one of CPU or memory placement, and then use these functions to determine the corresponding memory or CPU placement.

In some situations, applications will want to explicitly place both CPUs and memory nodes, not necessarily according to the default relation between a CPU and its local memory node. Such situations include the following.

- You can control specific CPU and memory node placement when precise placement control is required, such as when optimizing performance for a specific important application.
- A higher level system service such as a batch scheduler can control specific CPU and memory node placement.
- On systems having some memory on headless nodes (memory nodes with no associated local CPU), you can explicitly use a particular headless node with a particular CPU.
- On systems supporting Hyper-Threadthreading, one could affectively disable Hyper-Threading, by using just one out of the two or more execution engines (CPUs) available on a processor die.

6 Extensible API

In order to provide for the convenient and robust extensibility of this cpuset API over time, the following function enables dynamically obtaining pointers for optional functions by name, at run-time:

void *cpuset_function(const char * function_name) - returns function pointer, or NULL if function name unrecognized

For maximum portability, you should not reference any optional cpuset function by name by explicit name.

However, if you are willing to presume that an optional function will always be available on the target systems of interest, you might decide to explicitly reference it by name, in order to improve the clarity and simplicity of the software in question.

Also to support robust extensibility, flags and integer option values have names dynamically resolved at run-time, not via preprocessor macros.

All functions use only the primitive types of int, char *, pid_t (for process id of a task), size_t (for buffer sizes), pointers to opaque structures, functions whose signatures use these types, and pointers to such functions. They use no structure members, special types or magic define constants.

Some functions in Advanced Cpuset Library Functions are marked [**optional**]. They are not available in all implementations of libcpuset. Additional [**optional**] cpuset_* functions may also be added in the future. Functions that are *not* marked [**optional**] are available on all implementations of libcpuset.so, and can be called directly without using cpuset_function(). However, any of them *can* also be called indirectly via cpuset_function().

To safely invoke an optional function, such as for example `cpuset_migrate()`, use the following call sequence:

```
/* fp points to function of the type of cpuset_migrate() */
int (*fp)(struct cpuset *fromcp, struct cpuset *tocp, pid_t pid);
fp = cpuset_function("cpuset_migrate");
if (fp) {
        fp( ... );
} else {
        puts ("cpuset migration not supported");
}
```

If you invoke an [**optional**] function directly, then your resulting program will not be able to link with any version of libcpuset.so that does not define that particular function.

7 Cpuset Text Format

Cpuset settings may be exported to, and imported from, text files using a text format representation of cpusets.

The permissions of files holding these text representations have no special significance to the implementation of cpusets. Rather, the permissions of the special cpuset files in the cpuset file system, normally mounted at `/dev/cpuset`, control reading and writing of and attaching to cpusets.

The text representation of cpusets is not essential to the use of cpusets. One can directly manipulate the special files in the cpuset file system. This text representation provides an alternative that may be convenient for some uses.

The cpuset text format supports one directive per line. Comments begin with the '#' character and extend to the end of line.

After stripping comments, the first white space separated token on each remaining line selects from the following possible directives:

cpus Specify which CPUs are in this cpuset. The second token on the line must be a comma-separated list of CPU numbers and ranges of numbers, optionally modified by a stride operator (see below).

mems Specify which memory nodes are in this cpuset. The second token on the line must be a comma-separated list of memory node numbers and ranges of numbers, optionally modified by a stride operator (see below).

cpu_exclusive The cpu_exclusive flag is set.

mem_exclusive The mem_exclusive flag is set.

notify_on_release The notify_on_release flag is set

Additional unnecessary tokens on a line are quietly ignored. Lines containing only comments and white space are ignored.

The token "cpu" is allowed for "cpus", and "mem" for "mems". Matching is case insensitive.

The *stride operator* for "cpus" and "mems" values is used to designate every N-th CPU or memory node in a range of numbers. It is written as a colon **":"** followed by the number N, with no spaces on either side of the colon. For example, the range `0-31:2` designates the 16 even numbers 0, 2, 4, ... 30.

The **libcpuset** routines cpuset_import and cpuset_export to handle converting the internal 'struct cpuset' representation of cpusets to (export) and from (import) this text representation.

8 Basic Cpuset Library Functions

The basic cpuset API provides functions usable from 'C' for processor and memory placement within a cpuset.

These functions enable an application to place various threads of its execution on specific CPUs within its current cpuset, and perform related functions such as asking how large the current cpuset is, and on which CPU within the current cpuset a thread is currently executing.

These functions do not provide the full power of the advanced cpuset API, but they are easier to use, and provide some common needs of multi-threaded applications.

Unlike the rest of this document, the functions described in this section use cpuset relative numbering. In a cpuset of N CPUs, the relative cpu numbers range from zero to N - 1.

Unlike the underlying system calls `sched_setaffinity`, `mbind` or `set_mempolicy`, these basic cpuset API functions are robust in the presence of cpuset migration. If you pin a thread in your job to one of the CPUs in your jobs cpuset, it will stay properly pinned even if your jobs cpuset is migrated later to another set of CPUs and memory nodes, or even if the migration is occurring at the same time as your calls.

memory placement is done automatically, preferring the node local to the requested CPU. Threads may only be placed on a single CPU. This avoids the need to allocate and free the bitmasks required to specify a set of several CPUs. These functions do not support creating or removing cpusets, only the placement of threads within an existing cpuset. This avoids the need to explicitly allocate and free cpuset structures. Operations only apply to the current thread, avoiding the need to pass the process id of the thread to be affected.

If more powerful capabilities are needed, use the Advanced Cpuset Library Functions. These basic functions do not provide any essential new capability. They are implemented using the advanced functions, and are fully interoperable with them.

On error, these routines return -1 and set `errno`. If invoked on an operating system kernel that does not support cpusets, `errno` is set to `ENOSYS`. If invoked on a system that supports cpusets, but when the cpuset file system is not currently mounted at `/dev/cpuset`, then `errno` is set to `ENODEV`.

The following inclusion and linkage provides access to the cpuset API from 'C' code:

```
#include <cpuset.h>
/* link with -lcpuset */
```

The following functions are supported in the basic cpuset 'C' API:

- cpuset_pin - Pin the current thread to a CPU, preferring local memory
- cpuset_size - Return the number of CPUs are in the current tasks cpuset
- cpuset_where - On which CPU in current tasks cpuset did the task most recently execute
- cpuset_unpin - Remove affect of cpuset_pin, let task have run of its entire cpuset

8.1 cpuset_pin

```
int cpuset_pin(int relcpu);
```

Pin the current task to execute only on the CPU `relcpu`, which is a relative CPU number within the current cpuset of that task. Also, automatically pin the memory allowed to be used by the current task to prefer the memory on that same node (as determined by the cpuset_cpu2node function), but to allow any memory in the cpuset if no free memory is readily available on the same node.

Return 0 on success, -1 on error. Errors include `relcpu` being too large (greater than cpuset_size() - 1). They also include running on a system that doesn't support cpusets (`ENOSYS`) and running when the cpuset file system is not mounted at `/dev/cpuset` (`ENODEV`).

8.2 cpuset_size

```
int cpuset_size();
```

Return the number of CPUs in the current tasks cpuset. The relative CPU numbers that are passed to the cpuset_pin function and that are returned by the cpuset_where function, must be between 0 and N - 1 inclusive, where N is the value returned by cpuset_size.

Returns -1 and sets `errno` on error. Errors include running on a system that doesn't support cpusets (`ENOSYS`) and running when the cpuset file system is not mounted at `/dev/cpuset` (`ENODEV`).

8.3 cpuset_where

```
int cpuset_where();
```

Return the CPU number, relative to the current tasks cpuset, of the CPU on which the current task most recently executed. If a task is allowed to execute on more than one CPU, then there is no guarantee that the task is still executing on the CPU returned by cpuset_where, by the time that the user code obtains the return value.

Returns -1 and sets `errno` on error. Errors include running on a system that doesn't support cpusets (`ENOSYS`) and running when the cpuset file system is not mounted at `/dev/cpuset` (`ENODEV`).

8.4 cpuset_unpin

```
int cpuset_unpin();
```

Remove the CPU and memory pinning affects of any previous cpuset_pin call, allowing the current task to execute on any CPU in its current cpuset and to allocate memory on any memory node in its current cpuset. Return -1 on error, 0 on success.

Returns -1 and sets `errno` on error. Errors include running on a system that doesn't support cpusets (`ENOSYS`) and running when the cpuset file system is not mounted at `/dev/cpuset` (`ENODEV`).

9 Using Cpusets with Hyper-Threads

Threading in a software application splits instructions into multiple streams so that multiple processors can act on them.

Hyper-Threading (HT) Technology, developed by Intel Corporation, provides thread-level parallelism on each processor, resulting in more efficient use of processor resources, higher processing throughput, and improved performance. One physical CPU can appear as two logical CPUs by having additional registers to overlap two instruction streams or a single processor can have dual-cores executing instructions in parallel.

In addition to their traditional use to control the placement of jobs on the CPUs and memory nodes of a system, cpusets also provides a convenient mechanism to control the use of Hyper-Threading.

Some jobs achieve better performance using both of the Hyper-Thread sides, A and B, of a processor core, and some run better using just one of the sides, allowing the other side to idle.

Since each logical (Hyper-Threaded) processor in a core has a distinct CPU number, one can easily specify a cpuset that contains both sides, or contains just one side from each of the processor cores in the cpuset.

Cpusets can be configured to include any combination of the logical CPUs in a system.

For example, the following cpuset configuration file called `cpuset.cfg` includes the A sides of an HT enabled system, along with all the memory, on the first 32 nodes (assuming 2 cores per node). The colon ':' prefixes the stride. The stride of '2' in this example means use every other logical CPU.

```
cpus 0-127:2    # even numbered CPUs 0, 2, 4, ... 126
```

```
mems 0-31       # memory nodes 0, 1, 2, ... 31
```

To create a cpuset called `foo` and run a command called `bar` in that cpuset, defined by the cpuset configuration file `cpuset.cfg` shown above, use the following commands:

```
cpuset -c /foo < cpuset.cfg
cpuset -i /foo -I bar
```

To specify both sides of the first 64 cores, use the following entry in your cpuset configuration file:

```
cpus 0-127
```

To specify just the B sides, use the following entry in your cpuset configuration file:

```
cpus 1-127:2
```

The examples above assume that CPUs are uniformly numbered, with the even numbers for the A side and odd numbers for the B side. This is usually the case, but not guaranteed. One could still place a job on a system that was not uniformly numbered, but currently it would involve a longer argument list to the `cpus` option, explicitly listing the desired CPUs.

Typically, the logical numbering of CPUs puts the even numbered CPUs on the A sides, and the odd numbered CPUs on the B side. The stride operator (":2", above) makes it easy to specify that only every other side will be used. If the CPU number range starts with an even number, this will be the A sides, and if the range starts with an odd number, this will be the B sides.

A `ps(1)` or `top(1)` invocation will show a handful of threads on unused CPUs, but these are kernel threads assigned to every CPU in support of user applications running on those CPUs, to handle tasks such as asynchronous file system writes and task migration between CPUs. If no application is actually using a CPU, then the kernel threads on that CPU will be almost always idle and will consume very little memory.

10 Advanced Cpuset Library Functions

The advanced cpuset API provides functions usable from 'C' for managing cpusets on a system-wide basis.

These functions primarily deal with the three entities (1) `struct cpuset *`, (2) system cpusets and (3) tasks.

The `struct cpuset *` provides a transient in-memory structure used to build up a description of an existing or desired cpuset. These structs can be allocated, freed, queried and modified.

Actual kernel cpusets are created under `/dev/cpuset`, which is the usual mount point of the kernels virtual cpuset filesystem. These cpusets are visible to all tasks (with sufficient authority) in the system, and persist until the system is rebooted or until the cpuset is explicitly deleted. These cpusets can be created, deleted, queried, modified, listed and examined.

Every task (also known as a process) is bound to exactly one cpuset at a time. You can list which tasks are bound to a given cpuset, and to which cpuset a given task is bound. You can change to which cpuset a task is bound.

The primary attributes of a cpuset are its lists of CPUs and memory nodes. The scheduling affinity for each task, whether set by default or explicitly by the `sched_setaffinity()` system call, is constrained to those CPUs that are available in that tasks cpuset. The NUMA memory placement for each task, whether set by default or explicitly by the `mbind()` system call, is constrained to those memory nodes that are available in that tasks cpuset. This provides the essential purpose of cpusets - to constrain the CPU and memory usage of tasks to specified subsets of the system.

The other essential attribute of a cpuset is its pathname beneath `/dev/cpuset`. All tasks bound to the same cpuset pathname can be managed as a unit, and this hierarchical name space describes the nested resource management and hierarchical permission space supported by cpusets. Also, this hierarchy is used to enforce strict exclusion, using the following rules:

- A cpuset may only be marked strictly exclusive for CPU or memory if its parent is also.
- A cpuset may not make any CPUs or memory nodes available that are not also available in its parent.
- If a cpuset is exclusive for CPU or memory, then it may not overlap CPUs or memory with any of its siblings.

The combination of these rules enables checking for strict exclusion just by making various checks on the parent, siblings and existing child cpusets of the cpuset being changed, without having to check all cpusets in the system.

On error, some of these routines return -1 or NULL and set `errno`. If one of the routines below that requires cpuset kernel support is invoked on an operating system kernel that does not support cpusets, then that routine returns failure and `errno` is set to `ENOSYS`. If invoked on a system that supports cpusets, but when the cpuset file system is not currently mounted at `/dev/cpuset`, then it returns failure and `errno` is set to `ENODEV`.

The following inclusion and linkage provides access to the cpuset API from 'C' code:

```
#include <bitmask.h>
#include <cpuset.h>
/* link with -lcpuset */
```

The following functions are supported in the advanced cpuset 'C' API:

- Cpuset library (libcpuset) version
  - cpuset_version - [**optional**] Version (simple integer) of the library
- Allocate and free `struct cpuset *`
  - cpuset_alloc - Return handle to newly allocated `struct cpuset *`
  - cpuset_free - Discard no longer needed `struct cpuset *`
- Lengths of CPUs and memory nodes bitmasks - needed to allocate bitmasks
  - cpuset_cpus_nbits - Number of bits needed for a CPU bitmask on current system
  - cpuset_mems_nbits - Number of bits needed for a memory bitmask on current system
- Set various attributes of a `struct cpuset *`
  - cpuset_setcpus - Specify CPUs in cpuset
  - cpuset_setmems - Specify memory nodes in cpuset
  - cpuset_set_iopt - Specify an integer value option of cpuset
  - cpuset_set_sopt - [**optional**] Specify a string value option of cpuset
- Query various attributes of a `struct cpuset *`
  - cpuset_getcpus - Query CPUs in cpuset
  - cpuset_getmems - Query memory nodes in cpuset
  - cpuset_cpus_weight - Number of CPUs in a cpuset
  - cpuset_mems_weight - Number of memory nodes in a cpuset
  - cpuset_get_iopt - Query an integer value option of cpuset
  - cpuset_get_sopt - [**optional**] Query a string value option of cpuset
- Local CPUs and memory nodes
  - cpuset_localcpus - Query the CPUs local to specified memory nodes
  - cpuset_localmems - Query the memory nodes local to specified CPUs
  - cpuset_cpumemdist - [**optional**] Hardware distance from CPU to memory node
  - cpuset_cpu2node - Return system number of memory node closest to specified CPU.
  - cpuset_addr2node - Return sytem number of memory node holding page at specified address.

- Create, delete, query, modify, list and examine cpusets.
  - cpuset_create - Create named cpuset as specified by struct cpuset *
  - cpuset_delete - Delete the specified cpuset (if empty)
  - cpuset_query - Set struct cpuset to settings of specified cpuset
  - cpuset_modify - Modify a cpuset's settings to those specified in a struct cpuset
  - cpuset_getcpusetpath - Get path of a tasks (0 for current) cpuset.
  - cpuset_cpusetofpid - Set struct cpuset to settings of cpuset of specified task
  - cpuset_mountpoint - Return path at which cpuset filesystem is mounted
  - cpuset_collides_exclusive - [**optional**] True if would collide exclusive
  - cpuset_nuke - [**optional**] Remove cpuset anyway possible
- List tasks currently attached to a cpuset
  - cpuset_init_pidlist - Initialize a list of tasks attached to a cpuset
  - cpuset_pidlist_length - Return the number of tasks in such a list
  - cpuset_get_pidlist - Return a specific task from such a list
  - cpuset_freepidlist - Deallocate such a list
- Attach tasks to cpusets.
  - cpuset_move - Move task (0 for current) to a cpuset
  - cpuset_move_all - Move all tasks in a list of pids to a cpuset
  - cpuset_move_cpuset_tasks - [**optional**] Move all tasks in a cpuset to another cpuset
  - cpuset_migrate - [**optional**] Move a task and its memory to a cpuset
  - cpuset_migrate_all - [**optional**] Move all tasks with memory in a list of pids to a cpuset
  - cpuset_reattach - Rebind `cpus_allowed` of each task in a cpuset after changing its cpus
- Determine memory pressure
  - cpuset_open_memory_pressure - [**optional**] Open handle to read memory_pressure
  - cpuset_read_memory_pressure - [**optional**] Read cpuset current memory_pressure
  - cpuset_close_memory_pressure - [**optional**] Close handle to read memory pressure
- Map between cpuset relative and system-wide CPU and memory node numbers
  - cpuset_c_rel_to_sys_cpu - Map cpuset relative CPU number to system wide number
  - cpuset_c_sys_to_rel_cpu - Map system wide CPU number to cpuset relative number
  - cpuset_c_rel_to_sys_mem - Map cpuset relative memory node number to system wide number

  - cpuset_c_sys_to_rel_mem - Map system wide memory node number to cpuset relative number
  - cpuset_p_rel_to_sys_cpu - Map task cpuset relative CPU number to system wide number
  - cpuset_p_sys_to_rel_cpu - Map system wide CPU number to task cpuset relative number
  - cpuset_p_rel_to_sys_mem - Map task cpuset relative memory node number to system wide number
  - cpuset_p_sys_to_rel_mem - Map system wide memory node number to task cpuset relative number

- Placement operations - for detecting cpuset migration
  - cpuset_get_placement - [**optional**] Return current placement of task pid
  - cpuset_equal_placement - [**optional**] True if two placements equal
  - cpuset_free_placement - [**optional**] Free placement

- Traverse a cpuset hierarchy.
  - cpuset_fts_open - [**optional**] Open cpuset hierarchy
  - cpuset_fts_read - [**optional**] Next entry in hierarchy
  - cpuset_fts_reverse - [**optional**] Reverse order of cpusets
  - cpuset_fts_rewind - [**optional**] Rewind to first cpuset in list
  - cpuset_fts_get_path - [**optional**] Get entry's cpuset path
  - cpuset_fts_get_stat - [**optional**] Get entry's stat(2) pointer
  - cpuset_fts_get_cpuset - [**optional**] Get entry's cpuset pointer
  - cpuset_fts_get_errno - [**optional**] Get entry's errno
  - cpuset_fts_get_info - [**optional**] Get operation causing error
  - cpuset_fts_close - [**optional**] Close cpuset hierarchy

- Bind to a CPU or memory node within the current cpuset
  - cpuset_cpubind - Bind to a single CPU within a cpuset (uses sched_setaffinity(2))
  - cpuset_latestcpu - Most recent CPU on which a task has executed
  - cpuset_membind - Bind to a single memory node within a cpuset (uses set_mempolicy(2))

- Export cpuset settings to a regular file, and import them from a regular file
  - cpuset_export - Export cpuset settings to a text file
  - cpuset_import - Import cpuset settings from a text file

- Support calls to [**optional**] cpuset_* API routines

– cpuset_function - Return pointer to a libcpuset.so function, or `NULL`

A typical calling sequence would use the above functions in the following order to create a new cpuset named "xyz" and attach itself to it.

```
struct cpuset *cp = cpuset_alloc();
various cpuset_set*(cp, ...) calls
cpuset_create(cp, "xyz");
cpuset_free(cp);
cpuset_move(0, "xyz");
```

Some functions above are marked [**optional**]. For more information, see the *Extensible API* section, above, for an explanation of this marking and how to invoke such functions in a portable manner.

10.1 cpuset_version

```
int cpuset_version();
```

Version (simple integer) of the cpuset library (`libcpuset`). The version number returned by cpuset_version() is incremented anytime that any changes or additions are made to its API or behaviour. Other mechanims are provided to maintain full upward compatibility with this libraries API. This cpuset_version() call is intended to provide a fallback mechanism in case an application needs to distinguish between two previous versions of this library.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.2 cpuset_alloc

```
struct cpuset *cpuset_alloc();
```

Creates, initializes and returns a handle to a struct cpuset, which is an opaque data structure used to describe a cpuset.

After obtaining a struct cpuset handle with this call, one can use the various `cpuset_set()` methods to specify which CPUs and memory nodes are in the cpuset and other attributes. Then one can create such a cpuset with the `cpuset_create()` call and free cpuset handles with the `cpuset_free()` call.

The cpuset_alloc function returns a zero pointer (NULL) and sets *errno* in the event that `malloc(3)` fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.3 cpuset_free

```
void cpuset_free(struct cpuset *cp);
```

Frees the memory associated with a struct cpuset handle, which must have been returned by a previous `cpuset_alloc()` call. If cp is NULL, no operation is performed.

10.4 cpuset_cpus_nbits

```
int cpuset_cpus_nbits();
```

Return the number of bits in a CPU bitmask on current system. Useful when using `bitmask_alloc()` to allocate a CPU mask. Some other routines below return `cpuset_cpus_nbits()` as an out-of-bounds indicator.

10.5 cpuset_mems_nbits

```
int cpuset_mems_nbits();
```

Return the number of bits in a memory node bitmask on current system. Useful when using `bitmask_alloc()` to allocate a memory node mask. Some other routines below return `cpuset_mems_nbits()` as an out-of-bounds indicator.

10.6 cpuset_setcpus

```
int cpuset_setcpus(struct cpuset *cp, const struct bitmask *cpus);
```

Given a bitmask of CPUs, the cpuset_setcpus() call sets the specified cpuset `cp` to include exactly those CPUs.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if malloc(3) fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.7 cpuset_setmems

```
void cpuset_setmems(struct cpuset *cp, const struct bitmask *mems);
```

Given a bitmask of memory nodes, the `cpuset_setmems()` call sets the specified cpuset `cp` to include exactly those memory nodes.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if malloc(3) fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.8 cpuset_set_iopt

```
int cpuset_set_iopt(struct cpuset *cp, const char *optionname, int value);
```

Sets cpuset integer valued option `optionname` to specified integer value. Returns 0 if `optionname` is recognized and value is an allowed value for that option. Returns -1 if `optionname` is recognized, but value is not allowed. Returns -2 if `optionname` is not recognized. Boolean options accept any non-zero value as equivalent to a value of one (1).

The following `optionname`'s are recognized:

- `cpu_exclusive` - sibling cpusets not allowed to overlap cpus (see section Exclusive Cpusets, above)
- `mem_exclusive` - sibling cpusets not allowed to overlap mems (see section Exclusive Cpusets, above)
- `notify_on_release` - invoke `/sbin/cpuset_release_agent` when cpuset released (see section Notify On Release, above)
- `memory_migrate` - causes memory pages to migrate to new mems (see section Memory Migration, above)
- `memory_spread_page` - causes kernel buffer (page) cache to spread over cpuset (see section Memory Spread, above)
- `memory_spread_slab` - causes kernel file i/o data (directory and inode slab caches) to spread over cpuset(see section Memory Spread, above)

10.9 cpuset_set_sopt

```
int cpuset_set_sopt(struct cpuset *cp, const char *optionname, const char *value);
```

Sets cpuset string valued option `optionname` to specified string value. Returns 0 if `optionname` is recognized and value is an allowed value for that option. Returns -1

if `optionname` is recognized, but value is not allowed. Returns -2 if `optionname` is not recognized.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.10 cpuset_getcpus

```
int cpuset_getcpus(const struct cpuset *cp, struct bitmask *cpus);
```

Query CPUs in cpuset `cp`, by writing them to the bitmask `cpus`. Pass `cp` == NULL to query the current tasks cpuset

If the CPUs have not been set in cpuset `cp`, then no operation is performed, -1 is returned, and `errno` is set to `EINVAL`.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if malloc(3) fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.11 cpuset_getmems

```
int cpuset_getmems(const struct cpuset *cp, struct bitmask *mems);
```

Query memory nodes in cpuset `cp`, by writing them to the bitmask `mems`. Pass `cp` == NULL to query the current tasks cpuset.

If the memory nodes have not been set in cpuset `cp`, then no operation is performed, -1 is returned, and `errno` is set to `EINVAL`.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if malloc(3) fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.12 cpuset_cpus_weight

```
int cpuset_cpus_weight(const struct cpuset *cp);
```

Query number of CPUs in cpuset `cp`. Pass `cp` == NULL to query the current tasks cpuset

If the CPUs have not been set in cpuset `cp`, then zero(0) is returned.

10.13 cpuset_mems_weight

```
int cpuset_mems_weight(const struct cpuset *cp);
```

Query number of memory nodes in cpuset `cp`. Pass `cp` == NULL to query the current tasks cpuset

If the memory nodes have not been set in cpuset `cp`, then zero (0) is returned.

10.14 cpuset_get_iopt

```
int cpuset_get_iopt(const struct cpuset *cp, const char *optionname);
```

Query value of integer option `optionname` in cpuset `cp`. Returns value of `optionname` if it is recognized, else returns -1. Integer values in an uninitialized cpuset have value 0.

The following `optionname`'s are recognized:

- `cpu_exclusive` - sibling cpusets not allowed to overlap cpus (see section Exclusive Cpusets, above)
- `mem_exclusive` - sibling cpusets not allowed to overlap mems (see section Exclusive Cpusets, above)
- `notify_on_release` - invoke `/sbin/cpuset_release_agent` when cpuset released (see section Notify On Release, above)
- `memory_migrate` - causes memory pages to migrate to new mems (see section Memory Migration, above)
- `memory_spread_page` - causes kernel buffer (page) cache to spread over cpuset (see section Memory Spread, above)
- `memory_spread_slab` - causes kernel file i/o data (directory and inode slab caches) to spread over cpuset(see section Memory Spread, above)

10.15 cpuset_get_sopt

```
const char *cpuset_get_sopt(const struct cpuset *cp, const char *optionname);
```

Query value of string option `optionname` in cpuset `cp`. Returns pointer to value of `optionname` if it is recognized, else returns `NULL`. String values in an uninitialized cpuset have value `NULL`.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.16 cpuset_localcpus

```
int cpuset_localcpus(const struct bitmask *mems, struct bitmask *cpus);
```

Query the CPUs local to specified memory nodes `mems`, by writing them to the bitmask `cpus`. Return 0 on success, -1 on error, setting `errno`.

10.17 cpuset_localmems

```
int cpuset_localmems(const struct bitmask *cpus, struct bitmask *mems);
```

Query the memory nodes local to specified CPUs `cpus`, by writing them to the bitmask `mems`. Return 0 on success, -1 on error, setting `errno`.

10.18 cpuset_cpumemdist

```
unsigned int cpuset_cpumemdist(int cpu, int mem);
```

Distance between hardware CPU `cpu` and memory node `mem`, on a scale which has the closest distance of a CPU to its local memory valued at ten (10), and other distances more or less proportional. Distance is a rough metric of the bandwidth and delay combined, where a higher distance means lower bandwidth and longer delays.

If either `cpu` or `mem` is not known to the current system, or if any internal error occurs while evaluating this distance, or if a node has no CPUs nor memory (I/O only), then the distance returned is UCHAR_MAX (from limits.h).

These distances are obtained from the systems ACPI SLIT table, and should conform to:

> System Locality Information Table Interface Specification
> Version 1.0, July 25, 2003

This is an [**optional**] function. Use cpuset_function to invoke it.

10.19 cpuset_cpu2node

```
int cpuset_cpu2node(int cpu);
```

Return system wide number of memory node closest to CPU `cpu`. For NUMA architectures common as of this writing, this would be the number of the node on

which `cpu` is located. If an architecture did not have memory on the same node as a CPU, then it would be the node number of the memory node closest to or preferred by that `cpu`.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.20 cpuset_addr2node

```
int cpuset_addr2node(void *addr);
```

Return system wide number of memory node on which is located the physical page of memory at virtual address `addr` of the current tasks address space. Returns -1 if `addr` is not a valid address in the address space of the current process, with `errno` set to EFAULT. If the referenced physical page was not allocated (faulted in) by the kernel prior to this call, it will be during the call.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.21 cpuset_create

```
int cpuset_create(const char *cpusetpath, const struct *cp);
```

Create a cpuset at the specified `cpusetpath`, as described in the provided `struct cpuset *cp`. The parent cpuset of that pathname must already exist.

The parameter `cp` refers to a handle obtained from a `cpuset_alloc()` call. If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if malloc(3) fails. See the `malloc(3)` man page for possible values of *errno* (`ENOMEM` being the most likely).

10.22 cpuset_delete

```
int cpuset_delete(const char *cpusetpath);
```

Delete a cpuset at the specified `cpusetpath`. The cpuset of that pathname must already exist, be empty (no child cpusets) and be unused (no using tasks).

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

10.23 cpuset_query

```
int cpuset_query(struct cpuset *cp, const char *cpusetpath);
```

Set struct cpuset to settings of cpuset at specified path `cpusetpath`. `Struct cpuset *cp` must have been returned by a previous `cpuset_alloc()` call. Any previous settings of `cp` are lost.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, or -1 on error, setting `errno`. Errors include `cpusetpath` not referencing a valid cpuset path relative to `/dev/cpuset`, or the current task lacking permission to query that cpuset.

10.24 cpuset_modify

```
int cpuset_modify(const char *cpusetpath, const struct *cp);
```

Modify the cpuset at the specified `cpusetpath`, as described in the provided `struct cpuset *cp`. The cpuset at that pathname must already exist.

The parameter `cp` refers to a handle obtained from a `cpuset_alloc()` call.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

10.25 cpuset_getcpusetpath

```
char *cpuset_getcpusetpath(pid_t pid, char *buf, size_t size);
```

The `cpuset_getcpusetpath()` function copies an absolute pathname of the cpuset to which task of process id `pid` is attached, to the array pointed to by `buf`, which is of length `size`. Use pid == 0 for the current process.

The provided path is relative to the cpuset file system mount point.

If the cpuset path name would require a buffer longer than size elements, `NULL` is returned, and `errno` is set to `ERANGE` an application should check for this error, and allocate a larger buffer if necessary.

Returns `NULL` on failure with `errno` set accordingly, and buf on success. The contents of buf are undefined on error.

ERRORS

EACCES Permission to read or search a component of the file name was denied.

EFAULT buf points to a bad address.

ESRCH The pid does not exist.

E2BIG Larger buffer needed.

ENOSYS Kernel does not support cpusets.

10.26 cpuset_cpusetofpid

```
int cpuset_cpusetofpid(struct cpuset *cp, pid_t pid);
```

Set struct cpuset to settings of cpuset to which specified task `pid` is attached. `Struct cpuset *cp` must have been returned by a previous `cpuset_alloc()` call. Any previous settings of `cp` are lost.

Returns 0 on success, or -1 on error, setting `errno`.

ERRORS

EACCES Permission to read or search a component of the file name was denied.

EFAULT buf points to a bad address.

ESRCH The pid does not exist.

ERANGE Larger buffer needed.

ENOSYS Kernel does not support cpusets.

10.27 cpuset_mountpoint

```
const char *cpuset_mountpoint();
```

Return the filesystem path at which the cpuset file system is mounted. The current implementation of this routine returns `/dev/cpuset`, or the string `[cpuset filesystem not mounted]` if the cpuset file system is not mounted, or the string `[cpuset filesystem not supported]` if the system does not support cpusets.

In general, if the first character of the return string is a slash (`/`) then the result is the mount point of the cpuset file system, otherwise the result is an error message string.

This is an **[optional]** function. Use cpuset_function to invoke it.

10.28 cpuset_collides_exclusive

```
int cpuset_collides_exclusive(const char *cpusetpath, const struct *cp);
```

Return true (1) if cpuset `cp` would collide with any sibling of the cpuset at `cpusetpath` due to overlap of `cpu_exclusive` `cpus` or `mem_exclusive` `mems`. Return false (0) if no collision, or for any error.

cpuset_create fails with `errno == EINVAL` if the requested cpuset would overlap with any sibling, where either one is `cpu_exclusive` or `mem_exclusive`. This is a common, and not obvious error. `cpuset_collides_exclusive()` checks for this particular case, so that code creating cpusets can better identify the situation, perhaps to issue a more informative error message.

Can also be used to diagnose cpuset_modify failures. This routine ignores any existing cpuset with the same path as the given `cpusetpath`, and only looks for exclusive collisions with sibling cpusets of that path.

In case of any error, returns (0) -- does not collide. Presumably any actual attempt to create or modify a cpuset will encounter the same error, and report it usefully.

This routine is not particularly efficient; most likely code creating or modifying a cpuset will want to try the operation first, and then if that fails with `errno EINVAL`, perhaps call this routine to determine if an exclusive cpuset collision caused the error.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.29 cpuset_nuke

```
int cpuset_nuke(const char *cpusetpath, unsigned int seconds);
```

Remove a cpuset, including killing tasks in it, and removing any descendent cpusets and killing their tasks.

Tasks can take a long time (minutes on some configurations) to exit. Loop up to `seconds` seconds, trying to kill them.

The following steps are taken to remove a cpuset:

1. First, kill all the pids, looping until there are no more pids in this cpuset or below, or until the 'seconds' timeout limit is exceeded.
2. Then depth first recursively rmdir the cpuset directories.
3. If by this point the original cpuset is gone, return success.

If the timeout is exceeded, and tasks still exist, fail with errno == ETIME.

This routine sleeps a variable amount of time. After the first attempt to kill all the tasks in the cpuset or its descendents, it sleeps one second, the next time two

seconds, increasing one second each loop up to a max of ten seconds. If more loops past ten are required to kill all the tasks, it sleeps ten seconds each subsequent loop. In any case, before the last loop, it sleeps however many seconds remain of the original timeout `seconds` requested. The total time of all sleeps will be no more than the requested `seconds`.

If the cpuset started out empty of any tasks, or if the passed in `seconds` was zero, then this routine will return quickly, having not slept at all. Otherwise, this routine will at a minimum send a `SIGKILL` to all the tasks in this cpuset subtree, then sleep one second, before looking to see if any tasks remain. If tasks remain in the cpuset subtree, and a longer `seconds` timeout was requested (more than one), it will continue to kill remaining tasks and sleep, in a loop, for as long as time and tasks remain.

The signal sent for the kill is hardcoded to `SIGKILL`. If some other signal should be sent first, use a separate code loop, perhaps based on cpuset_init_pidlist and cpuset_get_pidlist, to scan the task pids in a cpuset. If `SIGKILL` should -not- be sent, this `cpuset_nuke()` routine can still be called to recursively remove a cpuset subtree, by specifying a timeout of zero `seconds`.

On success, returns 0 with `errno` == 0.

On failure, returns -1, setting `errno`.

ERRORS

`EACCES` search permission denied on intervening directory
`ETIME` timed out - tasks remain after 'seconds' timeout
`EMFILE` too many open files
`ENODEV` /dev/cpuset not mounted
`ENOENT` component of cpuset path doesn't exist
`ENOMEM` out of memory
`ENOSYS` kernel doesn't support cpusets
`ENOTDIR` component of cpuset path is not a directory
`EPERM` lacked permission to kill a task
`EPERM` lacked permission to read cpusets or files therein

This is an [**optional**] function. Use cpuset_function to invoke it.

10.30 cpuset_init_pidlist

```
struct cpuset_pidlist *cpuset_init_pidlist(const char *cpusetpath, int recursive-
flag);
```

Initialize and return a list of tasks (PIDs) attached to cpuset `cpusetpath`. If `recursiveflag` is zero, include only the tasks directly in that cpuset, otherwise include all tasks in that cpuset or any descendant thereof.

Beware that tasks may come and go from a cpuset, after this call is made.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

On error, return `NULL` and set `errno`.

10.31 cpuset_pidlist_length

```
int cpuset_pidlist_length(const struct cpuset_pidlist *pl);
```

Return the number of elements (PIDs) in cpuset_pidlist `pl`.

10.32 cpuset_get_pidlist

```
pid_t cpuset_get_pidlist(const struct cpuset_pidlist *pl, int i);
```

Return the i'th element of a `cpuset_pidlist`. The elements of a `cpuset_pidlist` of length `N` are numbered `0` through `N-1`. Return `(pid_t)-1` for any other index `i`.

10.33 cpuset_freepidlist

```
void cpuset_freepidlist(struct cpuset_pidlist *pl);
```

Deallocate a list of attached pids.

10.34 cpuset_move

```
int cpuset_move(pid_t p, const char *cpusetpath);
```

Move task whose process id is `p` to cpuset `cpusetpath`. If `pid` is zero, then the current task is moved.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

10.35 cpuset_move_all

```
int cpuset_move_all(struct cpuset_pid_list *pl, const char *cpusetpath);
```

Move all tasks in list `pl` to cpuset `cpusetpath`.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

10.36 cpuset_move_cpuset_tasks

```
int cpuset_move_cpuset_tasks(const char *fromrelpath, const char *torelpath);
```

Move all tasks in cpuset `fromrelpath` to cpuset `torelpath`. This may race with tasks being added to or forking into `fromrelpath`. Loop repeatedly, reading the tasks file of cpuset `fromrelpath` and writing any task pid's found there to the tasks file of cpuset `torelpath`, up to ten attempts, or until the `tasks` file of cpuset `fromrelpath` is empty, or until the cpuset `fromrelpath` is no longer present.

Returns 0 with errno == 0 if able to empty the tasks file of cpuset `fromrelpath`. Of course it is still possible that some independent task could add another task to cpuset `fromrelpath` at the same time that such a successful result is being returned, so there can be no guarantee that a successful return means that `fromrelpath` is still empty of tasks.

The cpuset `fromrelpath` might disappear during this operation, perhaps because it has `notify_on_release` set and was automatically removed as soon as its last task was detached from it. Consider a missing `fromrelpath` to be a successful move.

If called with `fromrelpath` and `torelpath` pathnames that evaluate to the same cpuset, then treat that as if cpuset_reattach() was called, rebinding each task in this cpuset one time, and return success or failure depending on the return of that cpuset_reattach() call.

On failure, returns -1, setting `errno`.

ERRORS

`EACCES` search permission denied on intervening directory
`ENOTEMPTY` tasks remain after multiple attempts to move them
`EMFILE` too many open files
`ENODEV` /dev/cpuset not mounted
`ENOENT` component of cpuset path doesn't exist
`ENOMEM` out of memory

`ENOSYS` kernel doesn't support cpusets
`ENOTDIR` component of cpuset path is not a directory
`EPERM` lacked permission to read cpusets or files therein
`EACCES‘ lacked permission to write a cpuset ‘‘tasks` file

This is an [**optional**] function. Use cpuset_function to invoke it.

10.37 cpuset_migrate

```
int cpuset_migrate(pid_t pid, const char *cpusetpath);
```

Migrate task whose process id is `p` to cpuset `cpusetpath`, moving its currently allocated memory to nodes in that cpuset, if not already there. If `pid` is zero, then the current task is migrated.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`

For more information, see the *Memory Migration* section, above.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.38 cpuset_migrate_all

```
int cpuset_migrate_all(struct cpuset_pid_list *pl, const char *cpusetpath);
```

Move all tasks in list `pl` to cpuset `cpusetpath`, moving their currently allocated memory to nodes in that cpuset, if not already there.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

For more information, see the *Memory Migration* section, above.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.39 cpuset_reattach

```
int cpuset_reattach(const char *cpusetpath);
```

Reattach all tasks in cpuset `cpusetpath` to itself. This additional step is necessary anytime that the `cpus` of a cpuset have been changed, in order to rebind the

`cpus_allowed` of each task in the cpuset to the new value. This routine writes the pid of each task currently attached to the named cpuset to the `tasks` file of that cpuset. If additional tasks are being spawned too rapidly into the cpuset at the same time, there is an unavoidable race condition, and some tasks may be missed.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

10.40 cpuset_open_memory_pressure

```
int cpuset_open_memory_pressure(const char *cpusetpath);
```

Open a file descriptor from which to read the `memory_pressure` of the cpuset `cpusetpath`.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this a path relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

By default, computation by the kernel of `memory_pressure` is disabled. Set the `memory_pressure_enabled` flag in the top cpuset to enable it.

On error, return `-1` and set `errno`.

For more information, see the *Memory Pressure* section, above.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.41 cpuset_read_memory_pressure

```
int cpuset_read_memory_pressure(int fd);
```

Read and return the current memory_pressure of the cpuset for which file descriptor `fd` was opened using `cpuset_open_memory_pressure`.

Uses the system call `pread(2)`. On success, returns a non-negative number, as described in section Memory Pressure. On failure, returns `-1` and sets `errno`.

By default, computation by the kernel of `memory_pressure` is disabled. Set the `memory_pressure_enabled` flag in the top cpuset to enable it.

For more information, see the *Memory Pressure* section, above.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.42 cpuset_close_memory_pressure

```
void cpuset_close_memory_pressure(int fd);
```

Close the file descriptor `fd` which was opened using `cpuset_open_memory_pressure`.

If `fd` is not a valid open file descriptor, then this call does nothing. No error is returned in any case.

By default, computation by the kernel of `memory_pressure` is disabled. Set the `memory_pressure_enabled` flag in the top cpuset to enable it.

For more information, see the *Memory Pressure* section, above.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.43 cpuset_c_rel_to_sys_cpu

```
int cpuset_c_rel_to_sys_cpu(const struct cpuset *cp, int cpu);
```

Return the system-wide CPU number that is used by the `cpu`-th CPU of the specified cpuset `cp`. Returns result of `cpuset_cpus_nbits()` if `cpu` is not in the range [0, `bitmask_weight(cpuset_getcpus(cp))`).

10.44 cpuset_c_sys_to_rel_cpu

```
int cpuset_c_sys_to_rel_cpu(const struct cpuset *cp, int cpu);
```

Return the `cpu`-th CPU of the specified cpuset `cp` that is used by the system-wide CPU number. Returns result of `cpuset_cpus_nbits()` if `bitmask_isbitset(cpuset_getcpus(cp), cpu)` is false.

10.45 cpuset_c_rel_to_sys_mem

```
int cpuset_c_rel_to_sys_mem(const struct cpuset *cp, int mem);
```

Return the system-wide memory node number that is used by the `mem`-th memory node of the specified cpuset `cp`. Returns result of `cpuset_mems_nbits()` if `mem` is not in the range [0, `bitmask_weight(cpuset_getmems(cp))`).

10.46 cpuset_c_sys_to_rel_mem

```
int cpuset_c_sys_to_rel_mem(const struct cpuset *cp, int mem);
```

Return the `mem`-th memory node of the specified cpuset `cp` that is used by the system-wide memory node number. Returns result of `cpuset_mems_nbits()` if `bitmask_isbitset(cpuset_getmems(cp), mem)` is false.

10.47 cpuset_p_rel_to_sys_cpu

```
int cpuset_p_rel_to_sys_cpu(pid_t pid, int cpu);
```

Return the system-wide CPU number that is used by the `cpu`-th CPU of the cpuset to which task `pid` is attached. Returns result of `cpuset_cpus_nbits()` if that cpuset doesn't encompass that relative cpu number.

10.48 cpuset_p_sys_to_rel_cpu

```
int cpuset_p_sys_to_rel_cpu(pid_t pid, int cpu);
```

Return the `cpu`-th CPU of the cpuset to which task `pid` is attached that is used by the system-wide CPU number. Returns result of `cpuset_cpus_nbits()` if that cpuset doesn't encompass that system-wide cpu number.

10.49 cpuset_p_rel_to_sys_mem

```
int cpuset_p_rel_to_sys_mem(pid_t pid, int mem);
```

Return the system-wide memory node number that is used by the `mem`-th memory node of the cpuset to which task `pid` is attached. Returns result of `cpuset_mems_nbits()` if that cpuset doesn't encompass that relative memory node number.

10.50 cpuset_p_sys_to_rel_mem

```
int cpuset_p_sys_to_rel_mem(pid_t pid, int mem);
```

Return the `mem`-th memory node of the cpuset to which task `pid` is attached that is used by the system-wide memory node number. Returns result of `cpuset_mems_nbits()`

if that cpuset doesn't encompass that system-wide memory node.

10.51 cpuset_get_placement

cpuset_get_placement(pid) - [optional Return current placement of task pid

This is an **[optional]** function. Use cpuset_function to invoke it.

This function returns an opaque `struct placement *` pointer. The results of calling cpuset_get_placement() twice at different points in a program can be compared by calling cpuset_equal_placement() to determine if the specified task has had its cpuset CPU and memory placement modified between those two cpuset_get_placement() calls.

When finished with a `struct placement *` pointer, free it by calling cpuset_free_placement().

10.52 cpuset_equal_placement

cpuset_equal_placement(plc1, plc2) - [optional] True if two placements equal

This is an **[optional]** function. Use cpuset_function to invoke it.

This function compares two `struct placement *` pointers, returned by two separate calls to cpuset_get_placement(). This is done to determine if the specified task has had its cpuset CPU and memory placement modified between those two cpuset_get_placement() calls.

When finished with a `struct placement *` pointer, free it by calling cpuset_free_placement().

Two `struct placement *` pointers will compare equal if they have the same CPU placement `cpus`, the same memory placement `mems`, and the same cpuset path.

10.53 cpuset_free_placement

cpuset_free_placement(plc) - [optional] Free placement

This is an **[optional]** function. Use cpuset_function to invoke it.

Use this routine to free a `struct placement *` pointer returned by a previous call to cpuset_get_placement().

10.54 cpuset_fts_open

```
struct cpuset_fts_tree *cpuset_fts_open(const char *cpusetpath);
```

Open a cpuset hierarchy. Returns a pointer to a `cpuset_fts_tree` structure, which can be used to traverse all cpusets below the specified cpuset `cpusetpath`.

If the parameter `cpusetpath` starts with a slash (`/`) character, then this path is relative to `/dev/cpuset`, otherwise it is relative to the current tasks cpuset.

The cpuset_fts_open routine is implemented internally using the `fts(3)` library routines for traversing a file hierarchy. The entire cpuset subtree below `cpusetpath` is traversed as part of the cpuset_fts_open() call, and all cpuset state and directory stat information is captured at that time. The other `cpuset_fts_*` routines just access this captured state. Any changes to the traversed cpusets made after the return of the cpuset_fts_open() call will not be visible via the returned `cpuset_fts_tree` structure.

Internally, the `fts(3)` options `FTS_NOCHDIR` and `FTS_XDEV` are used, to avoid changing the invoking tasks current directory, and to avoid descending into any other file systems mounted below `/dev/cpuset`. The order in which cpusets will be returned by the cpuset_fts_read routine corresponds to the fts `pre-order` (FTS_D) visitation order. The internal fts scan by cpuset_fts_open ignores the `post-order` (FTS_DP) results.

Because the cpuset_fts_open() call collects all the information at once from an entire cpuset subtree, a simple error return would not provide sufficient information as to what failed, and on what cpuset in the subtree. So, except for `malloc(3)` failures, errors are captured in the list of entries.

See cpuset_fts_get_info for details of the `info` field.

This is an **[optional]** function. Use cpuset_function to invoke it.

10.55 cpuset_fts_read

```
const struct cpuset_fts_entry *cpuset_fts_read(struct cpuset_fts_tree *cs_tree);
```

Returns next `cs_entry` in `cpuset_fts_tree` `cs_tree` obtained from an cpuset_fts_open() call. One `cs_entry` is returned for each cpuset directory that was found in the subtree scanned by the cpuset_fts_open() call. Use the `info` field obtained from a cpuset_fts_get_info() call to determine which fields of a particular `cs_entry` are valid, and which fields contain error information or are not valid.

This is an **[optional]** function. Use cpuset_function to invoke it.

10.56 cpuset_fts_reverse

```
void cpuset_fts_reverse(struct cpuset_fts_tree *cs_tree);
```

Reverse order of `cs_entry`'s in the `cpuset_fts_tree cs_tree` obtained from a cpuset_fts_open() call.

An open `cpuset_fts_tree` stores a list of `cs_entry` cpuset entries, in pre-order, meaning that a series of cpuset_fts_read() calls will always return a parent cpuset before any of its child cpusets. Following a cpuset_fts_reverse() call, the order of cpuset entries is reversed, putting it in post-order, so that a series of cpuset_fts_read() calls will always return any children cpusets before their parent cpuset. A second cpuset_fts_reverse() call would put the list back in pre-order again.

To avoid exposing confusing inner details of the implementation across the API, a cpuset_fts_rewind() call is always automatically performed on a `cpuset_fts_tree` whenever cpuset_fts_reverse() is called on it.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.57 cpuset_fts_rewind

```
void cpuset_fts_rewind(struct cpuset_fts_tree *cs_tree);
```

Rewind a cpuset tree `cs_tree` obtained from a cpuset_fts_open() call, so that subsequent cpuset_fts_read() calls start from the beginning again.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.58 cpuset_fts_get_path

```
const char *cpuset_fts_get_path(const struct cpuset_fts_entry *cs_entry);
```

Return the cpuset path, relative to `/dev/cpuset`, as nul-terminated string, of a `cs_entry` obtained from a cpuset_fts_read() call.

The results of this call are valid for all `cs_entry`'s returned from cpuset_fts_read() calls, regardless of the value returned by cpuset_fts_get_info() for that `cs_entry`.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.59 cpuset_fts_get_stat

```
const struct stat *cpuset_fts_get_stat(const struct cpuset_fts_entry *cs_entry);
```

Return pointer to `stat(2)` information about the cpuset directory of a `cs_entry` obtained from a cpuset_fts_read() call.

The results of this call are valid for all `cs_entry`'s returned from cpuset_fts_read() calls, regardless of the value returned by cpuset_fts_get_info() for that `cs_entry`, **except** in the cases that:

- the `info` field returned by cpuset_fts_get_info contains `CPUSET_FTS_ERR_DNR`, in which case, a directory in the path to the cpuset could not be read and this call will return a `NULL` pointer, or
- the `info` field returned by cpuset_fts_get_info contains `CPUSET_FTS_ERR_STAT`, in which case a `stat(2)` failed on this cpuset directory and this call will return a pointer to a `struct stat` containing all zeros.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.60 cpuset_fts_get_cpuset

```
const struct cpuset *cpuset_fts_get_cpuset(const struct cpuset_fts_entry *cs_entry);
```

Return the `struct cpuset` pointer of a `cs_entry` obtained from a cpuset_fts_read() call. The `struct cpuset` so referenced describes the cpuset represented by one directory in the cpuset hierarchy, and can be used with various other calls in this library.

The results of this call are only valid for a `cs_entry` if the cpuset_fts_get_info() call returns `CPUSET_FTS_CPUSET` for the `info` field of a `cs_entry`. If the `info` field contained `CPUSET_FTS_ERR_CPUSET`, then cpuset_fts_get_cpuset returns a pointer to a `struct cpuset` that is all zeros. If the `info` field contains any other `CPUSET_FTS_ERR_*` value, then cpuset_fts_get_cpuset returns a NULL pointer.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.61 cpuset_fts_get_errno

```
int cpuset_fts_get_errno(const struct cpuset_fts_entry *cs_entry);
```

Return the `err` field of a `cs_entry` obtained from a cpuset_fts_read() call.

If an entry (obtained from cpuset_fts_read) has one of the CPUSET_FTS_ERR_* values in the info field (as described in cpuset_fts_get_info), then this err field captures the failing errno value for that operation. If an entry has the value CPUSET_FTS_CPUSET in its info field, then this err field will have the value 0.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.62 cpuset_fts_get_info

```
int cpuset_fts_get_info(const struct cpuset_fts_entry *cs_entry);
```

Return the info field of a cs_entry obtained from a cpuset_fts_read() call.

If this info field has one of the following CPUSET_FTS_ERR_* values, then it indicates which operation failed, the err field (returned by cpuset_fts_get_errno) captures the failing errno value for that operation, the path field (returned by cpuset_fts_get_path) indicates which cpuset failed, and some of the other entry fields may not be valid, depending on the value. If an entry has the value CPUSET_FTS_CPUSET for its info field, then the err field will have the value 0, and the other fields will be contain valid information about that cpuset.

info field values:

CPUSET_FTS_CPUSET = 0: Valid cpuset
CPUSET_FTS_ERR_DNR = 1: Error - couldn't read directory
CPUSET_FTS_ERR_STAT = 2: Error - couldn't stat directory
CPUSET_FTS_ERR_CPUSET = 3: Error - cpuset_query failed

The above info field values are defined using an anonymous *enum* in the cpuset.h header file. If it necessary to maintain source code compatibility with earlier versions of the cpuset.h header file lacking the above CPUSET_FTS_* values, one can conditionally check that the C preprocessor symbol CPUSET_FTS_INFO_VALUES_DEFINED is not defined and provide alternative coding for that case.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.63 cpuset_fts_close

```
void cpuset_fts_close(struct cpuset_fts_tree *cs_tree);
```

Close a cs_tree obtained from a cpuset_fts_open() call, freeing any internally allocated memory for that cs_tree.

This is an [**optional**] function. Use cpuset_function to invoke it.

10.64 cpuset_cpubind

```
int cpuset_cpubind(int cpu);
```

Bind the scheduling of the current task to CPU `cpu`, using the sched_setaffinity(2) system call.

Fails with a return of -1, and `errno` set to `EINVAL`, if `cpu` is not allowed in the current tasks cpuset.

The following code will bind the scheduling of a thread to the n-th CPU of the current cpuset:

```
/*
 * Confine current task to only run on the n-th CPU
 * of its current cpuset. In a cpuset of N CPUs,
 * valid values for n are 0 .. N-1.
 */
cpuset_cpubind(cpuset_p_rel_to_sys_cpu(0, n));
```

10.65 cpuset_latestcpu

```
int cpuset_latestcpu(pid_t pid);
```

Return the most recent CPU on which the specified task `pid` executed. If pid is 0, examine current task.

The `cpuset_latestcpu()` call returns the number of the CPU on which the specified task `pid` most recently executed. If a process can be scheduled on two or more CPUs, then the results of `cpuset_lastcpu()` may become invalid even before the query returns to the invoking user code.

The last used CPU is visible for a given pid as field #39 (starting with #1) in the file /proc/pid/stat. Currently this file has 41 fields, so its the 3rd to the last field.

10.66 cpuset_membind

```
int cpuset_membind(int mem);
```

Bind the memory allocation of the current task to memory node `mem`, using the set_mempolicy(2) system call with a policy of MPOL_BIND.

Fails with a return of -1, and `errno` set to `EINVAL`, if `mem` is not allowed in the current tasks cpuset.

The following code will bind the memory allocation of a thread to the n-th memory node of the current cpuset:

```
/*
 * Confine current task to only allocate memory on
 * n-th node of its current cpuset.  In a cpuset of
 * N memory nodes, valid values for n are 0 .. N-1.
 */
cpuset_membind(cpuset_p_rel_to_sys_mem(0, n));
```

10.67 cpuset_export

```
int cpuset_export(const struct cpuset *cp, char *buf, int buflen);
```

Write the settings of cpuset `cp` to `file`. If no file exists at the path specified by `file`, create one. If a file already exists there, overwrite it.

Returns -1 and sets `errno` on error. Upon successful return, returns the number of characters printed (not including the trailing '0' used to end output to strings). The function cpuset_export does not write more than size bytes (including the trailing '0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '0') which would have been written to the final string if enough space had been available. Thus, a return value of size or more means that the output was truncated.

See Cpuset Text Format for details of the format of an exported cpuset file.

10.68 cpuset_import

```
int cpuset_import(struct cpuset *cp, const char *file, int *errlinenum_ptr, char
*errmsg_bufptr, int errmsg_buflen);
```

Read the settings of cpuset `cp` from `file`. If no file exists at the path specified by `file`, create one. If a file already exists there, overwrite it.

`Struct cpuset *cp` must have been returned by a previous `cpuset_alloc()` call. Any previous settings of `cp` are lost.

Returns 0 on success, or -1 on error, setting `errno`. Errors include `file` not referencing a readable file.

If parsing errors are encountered reading the file, and if errlinenum_ptr is not NULL, then the number of the first line (numbers start with one) with an error is written to

*errlinenum_ptr. If an error occurs on the open, and errlinenum_ptr is not NULL, then zero is written to *errlinenum_ptr.

If parsing errors are encountered reading the file and if errmsg_bufptr is not NULL, then it is presumed to point to a character buffer of at least errmsg_buflen characters, and a nul terminated error message is written to *errmsg_bufptr providing a human readable error message explaining the error message in more detail. As of this writing, the possible error messages are:

- "Token 'CPU' requires list"
- "Token 'MEM' requires list"
- "Invalid list format: %s"
- "Unrecognized token: %s"
- "Insufficient memory"

See Cpuset Text Format for details of the format required for imported cpuset file.

10.69 cpuset_function

```
cpuset_function(const char *function_name);
```

Return pointer to the named `libcpuset.so` function, or `NULL`. For base functions that are in all implementations of libcpuset, there is no particular value in using `cpuset_function()` to obtain a pointer to the function dynamically. But for [**optional**] cpuset functions, the use of `cpuset_function()` enables dynamically adapting to run-time environments that may or may not support that function.

For more information, see the *Extensible API* section, above.

11 System Error Numbers

The Linux kernel implementation of cpusets sets `errno` to specify the reason for a failed system call affecting cpusets. These `errno` values are available when a cpuset library call fails. Most of these values can also be displayed by shell commands used to directly manipulate files below `/dev/cpuset`.

The possible `errno` settings and their meaning when set on a failed cpuset call are as listed below.

ENOSYS Invoked on an operating system kernel that does not support cpusets.

ENODEV Invoked on a system that supports cpusets, but when the cpuset file system is not currently mounted at `/dev/cpuset`.

ENOMEM Insufficient memory is available.

EBUSY Attempted `cpuset_delete()` on a cpuset with attached tasks.

EBUSY Attempted `cpuset_delete()` on a cpuset with child cpusets.

ENOENT Attempted `cpuset_create()` in a parent cpuset that doesn't exist.

EEXIST Attempted `cpuset_create()` for a cpuset that already exists.

EEXIST Attempted to `rename(2)` a cpuset to a name that already exists.

ENOTDIR Attempted to `rename(2)` a non-existent cpuset.

E2BIG Attempted a `write(2)` system call on a special cpuset file with a length larger than some kernel determined upper limit on the length of such writes.

ESRCH Attempted to `cpuset_move()` a non-existent task.

EACCES Attempted to `cpuset_move()` a task which one lacks permission to move.

EACCES Attempted to `write(2)` a `memory_pressure` file.

ENOSPC Attempted to `cpuset_move()` a task to an empty cpuset.

EINVAL The `relcpu` argument to `cpuset_pin()` is out of range (not between "zero" and "cpuset_size() - 1").

EINVAL Attempted to change a cpuset in a way that would violate a cpu_exclusive or mem_exclusive attribute of that cpuset or any of its siblings.

EINVAL Attempted to write an empty cpus or mems bitmask to the kernel. The kernel creates new cpusets (via mkdir) with empty cpus and mems, and the user level cpuset and bitmask code works with empty masks. But the kernel will not allow an empty bitmask (no bits set) to be written to the special cpus or mems files of a cpuset.

EIO Attempted to `write(2)` a string to a cpuset tasks file that does not begin with an ASCII decimal integer.

EIO Attempted to `rename(2)` a cpuset outside of its current directory.

ENOSPC Attempted to `write(2)` a list to a cpus file that did not include any online cpus.

ENOSPC Attempted to `write(2)` a list to a mems file that did not include any online memory nodes.

EACCES Attempted to add a cpu or mem to a cpuset that is not already in its parent.

EACCES Attempted to set cpu_exclusive or mem_exclusive on a cpuset whose parent lacks the same setting.

ENODEV The cpuset was removed by another task at the same time as a `write(2)` was attempted on one of the special files in the cpuset directory.

EBUSY Attempted to remove a cpu or mem from a cpuset that is also in a child of that cpuset.

EFAULT Attempted to read or write a cpuset file using a buffer that was outside your accessible address space.

ENAMETOOLONG Attempted to read a `/proc/<pid>/cpuset` file for a cpuset path that was longer than the kernel page size.

ENAMETOOLONG Attempted to create a cpuset whose base directory name was longer than 255 characters.

ENAMETOOLONG Attempted to create a cpuset whose full pathname including the "/dev/cpuset" is longer than 4095 characters.

ENXIO Attempted to create a `cpu_exclusive` cpuset whose `cpus` covered just part of one or more physical processor packages, such as including just one of the two Cores on a package. For Linux kernel version 2.6.16 on i386 and x86_64, this operation is rejected with this error to avoid a fatal kernel bug. Otherwise, this is a normal and supported operation.

EINVAL Specified a cpus or mems list to the kernel which included a range with the second number smaller than the first number.

EINVAL Specified a cpus or mems list to the kernel which included an invalid character in the string.

ERANGE Specified a cpus or mems list to the kernel which included a number too large for the kernel to set in its bitmasks.

ETIME Time limit for cpuset_nuke operation reached without successful completion of operation.

ENOTEMPTY Tasks remain after multiple attempts by cpuset_move_cpuset_tasks to move them to a different cpuset.

EPERM Lacked permission to kill (send a signal to) a task.

EPERM Lacked permission to read a cpuset or its files.

EPERM Attempted to unlink a per-cpuset file. Such files can not be unlinked. They can only be removed by removing (rmdir) the directory representing the cpuset that contains these files.

12 Change History

Here is the history of changes to this document and associated software.

- 2006-04-26 Paul Jackson <pj@sgi.com>
  - First published.

- 2006-11-14 Paul Jackson <pj@sgi.com> Version 1 (cpuset_version())
  - Added cpuset_nuke routine.
  - Added various `cpuset_fts_*` routines.
  - Added various `cpuset_*_affinity` routines.
  - Added cpuset_move_cpuset_tasks routine.
  - Converted from using `ftw(3)` to `fts(3)`.
  - Various minor documentation errors fixed.
  - Added cpuset_version routine.
  - Improved error checking by cpuset_move_all.
  - Correct system call numbering for sched_setaffinity on i386 arch.
  - Correct cpuset_latestcpu result if command basename has space character.
  - Remove 256 byte limit on cpuset_export output.
- 2007-01-14 Paul Jackson <pj@sgi.com> Version 2 (cpuset_version())
  - Fix cpuset_create, cpuset_modify to not zero undefined attributes such as `memory_spread_page`, `memory_spread_slab` and `notify_on_release`. See further the explanation of an "*undefined* mark", in Cpuset Programming Model. cpuset_create now respects default kernel cpuset creation settings, and cpuset_modify now respects the existing settings of the cpuset, unless explicitly set.
- 2007-04-01 Paul Jackson <pj@sgi.com> Version 3 (cpuset_version())
  - Fix cpuset_setcpus() and cpuset_setmems() to mark the `cpus` and `mems` fields as defined, so that setting them before doing a cpuset_create() has affect.