



Linux[®] Resource Administration Guide

007-4413-017

COPYRIGHT

© 2002–2007, 2010, 2011, 2013 SGI. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of SGI.

LIMITED RIGHTS LEGEND

The software described in this document is "commercial computer software" provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

SGI, the SGI logo, and Supportfolio are trademarks or registered trademarks of Silicon Graphics International Corp. or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds. UNIX and the X Window System are registered trademarks of The Open Group in the United States and other countries. All other trademarks mentioned herein are the property of their respective owners.

New Features in This Manual

The information about Array Services has been moved to the *Message Passing Toolkit (MPT) User Guide*.

Record of Revision

Version	Description
001	February 2003 Original publication.
002	June 2003 Updated to support the SGI ProPack for Linux v2.2 release
003	October 2003 Updated to support the SGI ProPack for Linux v2.3 release.
004	February 2004 Updated to support the SGI ProPack for Linux v2.4 release.
005	May 2004 Updated to support the SGI ProPack 3 for Linux release.
006	January 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 3 release.
007	February 2005 Updated to support the SGI ProPack 4 for Linux release.
008	April 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 5 release.
009	August 2005 Updated to support the SGI ProPack 4 for Linux Service Pack 2 release.
010	January 2006 Updated to support the SGI ProPack 4 for Linux Service Pack 3 release.
011	July 2006 Updated to support the SGI ProPack 5 for Linux release.

- 012 April 2007
Updated to support the SGI ProPack 5 for Linux Service Pack 1 release.
- 013 July 2008
Updated to support the SGI ProPack 6 for Linux release.
- 014 January 2009
Updated to support the SGI ProPack 6 for Linux Service Pack 2 release.
- 015 October 2010
Updated to support the SGI Performance Suite 1.0 release.
- 016 November 2011
Updated to support the SGI Performance Suite 1.3 release.
- 017 May 2013
Updated to support the SGI Performance Suite 1.6 release.

Contents

About This Guide	xiii
Related Publications	xiii
Obtaining Publications	xiii
Conventions	xiii
Reader Comments	xiv
1. Cpusets on Linux	1
An Overview of the Advantages Gained by Using Cpusets	1
Linux 2.6 Kernel Support for Cpusets	3
Cpuset Facility Capabilities	4
Initializing Cpusets	4
How to Determine if Cpusets are Installed	5
Fine-grained Control within Cpusets	6
Cpuset Interaction with Other Placement Mechanism	6
Cpusets and Thread Placement	8
Safe Job Migration and Cpusets	8
Cpuset File System Directories	10
Exclusive Cpusets	15
Notify on Release Flag	15
Memory Pressure of a Cpuset	16
Memory Spread	18
Memory Migration	19
Mask Format	20
List Format	20
007-4413-017	vii

Cpuset Permissions	21
CPU Scheduling and Memory Allocation for Cpusets	21
Linux Kernel CPU and Memory Placement Settings	22
Manipulating Cpusets	23
Using Cpusets at the Shell Prompt	23
Cpuset Command Line Utility	25
Boot Cpuset	30
Creating a Bootcpuset	30
bootcpuset.conf File	31
Configuring a User Cpuset for Interactive Sessions	32
Cpuset Text Format	35
Modifying the CPUs in a Cpuset and Kernel Processing	36
Using Cpusets with Hyper-Threads	37
Cpuset Programming Model	40
System Error Messages	41
2. NUMA Tools	43
Appendix A. Cpuset Library Functions	45
Extensible Application Programming Interface	45
Basic Cpuset Library Functions	46
cpuset_pin	47
cpuset_size	48
cpuset_where	48
cpuset_unpin	48
Advanced Cpuset Library Functions	49
cpuset_alloc	54
cpuset_free	55

cpuset_cpus_nbits	55
cpuset_mems_nbits	55
cpuset_setcpus	55
cpuset_setmems	56
cpuset_set_iopt	56
cpuset_set_sopt	57
cpuset_getcpus	57
cpuset_getmems	57
cpuset_cpus_weight	58
cpuset_mems_weight	58
cpuset_get_iopt	58
cpuset_get_sopt	59
cpuset_localcpus	59
cpuset_localmems	60
cpuset_cpumemdist	60
cpuset_cpu2node	60
cpuset_create	61
cpuset_delete	61
cpuset_query	61
cpuset_modify	62
cpuset_getcpusetpath	62
cpuset_cpusetofpid	63
cpuset_cpusetofpid	63
cpuset_mountpoint	64
cpuset_collides_exclusive	64
cpuset_init_pidlist	65
cpuset_pidlist_length	65
cpuset_get_pidlist	65

Contents

cpuset_free_pidlist	65
cpuset_move	66
cpuset_move_all	66
cpuset_move_cpuset_tasks	66
cpuset_migrate	67
cpuset_migrate_all	68
cpuset_reattach	68
cpuset_open_memory_pressure	68
cpuset_read_memory_pressure	69
cpuset_close_memory_pressure	69
cpuset_c_rel_to_sys_cpu	70
cpuset_c_sys_to_rel_cpu	70
cpuset_c_rel_to_sys_mem	70
cpuset_c_sys_to_rel_mem	70
cpuset_p_rel_to_sys_cpu	71
cpuset_p_sys_to_rel_cpu	71
cpuset_p_rel_to_sys_mem	71
cpuset_p_sys_to_rel_mem	71
cpuset_get_placement	71
cpuset_equal_placement	72
cpuset_free_placement	72
cpuset_cpupbind	72
cpuset_latestcpu	73
cpuset_membind	73
cpuset_nuke	74
cpuset_export	75
cpuset_import	75
cpuset_function	76

cpuset_version	76
Functions to Traverse a Cpuset Hierarchy	77
cpuset_fts_open	77
cpuset_fts_read	77
cpuset_fts_reverse	78
cpuset_fts_rewind	78
cpuset_fts_get_path	78
cpuset_fts_get_stat	79
cpuset_fts_get_cpuset	79
cpuset_fts_get_errno	80
cpuset_fts_close	80
Index	81

About This Guide

This guide is a reference document for people who administer SGI computer systems that run the SGI Performance Suite software. It contains information needed in the administration of various system resource management features.

This manual contains the following chapters:

- Chapter 1, "Cpusets on Linux" on page 1
- Chapter 2, "NUMA Tools" on page 43
- Appendix A, "Cpuset Library Functions" on page 45

Related Publications

The *SGI Performance Suite X.X Start Here* contains a list of manuals that support the SGI Performance Suite software.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- You can access user guides and reference manuals from the SGI Technical Publications Library at the following website:

<http://docs.sgi.com>.

Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man(1) pages, and other information.

- You can log into an SGI system and retrieve Linux man(1) pages for a particular topic title. Type `man title` on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:
techpubs@sgi.com
- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:
SGI
Technical Publications
46600 Landing Parkway
Fremont, CA 94538

SGI values your comments and will respond to them promptly.

Cpusets on Linux

This chapter describes the cpuset facility on systems running the SGI® Accelerate™ software and covers the following topics:

- "An Overview of the Advantages Gained by Using Cpusets" on page 1
- "Cpuset Permissions" on page 21
- "Cpuset File System Directories" on page 10
- "CPU Scheduling and Memory Allocation for Cpusets" on page 21
- "Using Cpusets at the Shell Prompt" on page 23
- "Cpuset Command Line Utility"
- "Boot Cpuset" on page 30
- "Configuring a User Cpuset for Interactive Sessions" on page 32
- "Cpuset Text Format" on page 35
- "Modifying the CPUs in a Cpuset and Kernel Processing" on page 36
- "Using Cpusets with Hyper-Threads" on page 37
- "Cpuset Programming Model" on page 40
- "System Error Messages" on page 41

An Overview of the Advantages Gained by Using Cpusets

The cpuset facility allows a system administrator or a workload manager, such as PBS Pro or MOAB/Torque, to restrict the number of processor and memory resources that a process or set of processes may use.

A *cpuset* defines a list of CPUs and memory nodes. A process contained in a cpuset may only execute on the CPUs in that cpuset and may only allocate memory on the memory nodes in that cpuset. Essentially, cpusets provide you with a CPU and memory containers or “soft partitions” within which you can run sets of related tasks. Using cpusets on an SGI UV system improves memory locality and memory access times and can substantially improve an application’s performance and runtime

repeatability. Restraining all other jobs from using any of the CPUs or memory resources assigned to a critical job minimizes interference from other jobs on the system. For example, Message Passing Interface (MPI) jobs frequently consist of a number of threads that communicate using message passing interfaces. All threads need to be executing at the same time. If a single thread loses a CPU, all threads stop making forward progress and spin at a synchronization or communication function.

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 a job. This results in both optimum and repeatable performance.

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

Cpusets are represented in a hierarchical virtual file system. Cpusets can be nested and they have file-like permissions.

The `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls allow you to specify 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 filesystem-like namespace and permissions, and support for guaranteed exclusive use of resources.

You can have a *boot* cpuset running the traditional daemon and server tasks and a second cpuset to hold interactive `telnet`, `rlogin` and/or secure shell (SSH) user sessions called the *user* cpuset.

Creating a user cpuset provides additional isolation between interactive user login sessions and essential system tasks. For example, a user process in the user cpuset consuming excessive CPU, system file buffer cache, or memory resources will not seriously impact essential system services in the boot cpuset. For more information, see "Configuring a User Cpuset for Interactive Sessions" on page 32.

This section covers the following topics:

- "Linux 2.6 Kernel Support for Cpusets" on page 3
- "Cpuset Facility Capabilities" on page 4
- "Initializing Cpusets" on page 4

- "How to Determine if Cpusets are Installed" on page 5
- "Fine-grained Control within Cpusets" on page 6
- "Cpuset Interaction with Other Placement Mechanism" on page 6
- "Cpusets and Thread Placement" on page 8
- "Safe Job Migration and Cpusets" on page 8

Linux 2.6 Kernel Support for Cpusets

The Linux 2.6 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.
- Hooks in the `sched_setaffinity` system call, used for CPU placement, and in the `mbind` system call, used for memory placement, ensure that any requested CPU or memory node is available in that task's cpuset.
- All tasks sharing the same placement constraints reference the same cpuset.
- 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 allocation mechanism is constrained to only allocate physical memory to a task from the memory nodes in that task's cpuset.
- The kernel memory allocation mechanism provides an economical, per-cpuset metric of the aggregate memory pressure of the tasks in a cpuset. *Memory pressure* is defined as the frequency of requests for a free memory page that is not easily satisfied by an available free page. For more information, see "Memory Pressure of a Cpuset" on page 16.
- The kernel memory allocation mechanism provides an option that allows you 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. Otherwise, they are preferentially allocated on whatever memory node that the task first accessed the memory page.

- You can control the memory migration facility in the kernel using per-cpuset files. When the memory nodes allowed to a task by cpusets changes, any memory pages no longer allowed on that node may be migrated to nodes now allowed. For more information, see "Safe Job Migration and Cpusets" on page 8.

Cpuset Facility Capabilities

A cpuset constrains the jobs (set of related tasks) running in it to a subset of the system's memory and CPUs. The cpuset facility allows you and your system service software to do the following:

- 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, thereby, 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 of system resources using hierarchical permissions and resource management.

Initializing Cpusets

The kernel, at system boot time, initializes one cpuset, the root cpuset, containing the entire system's CPUs and memory nodes. Subsequent user space operations can create additional cpusets.

Mounting the cpuset virtual file system (VFS) at `/dev/cpuset` exposes the kernel mechanism to user space. This VFS allows for nested resource allocations and the associated hierarchical permission model.

You can initialize and perform other cpuset operations, using any of the these three mechanisms, as follows:

- You can create, change, or query cpusets by using shell commands on `/dev/cpuset`, such as `echo(1)`, `cat(1)`, `mkdir(1)`, or `ls(1)` as described in "Using Cpusets at the Shell Prompt" on page 23.
- You can use the `cpuset(1)` command line utility to create or destroy cpusets or to retrieve information about existing cpusets and to attach processes to existing cpusets as described in "Cpuset Command Line Utility" on page 25.
- You can use the `libcputset` C programming application programming interface (API) functions to query or change them from within your application as described in Appendix A, "Cpuset Library Functions" on page 45. You can find information about `libcputset` at `/usr/share/doc/packages/libcputset/libcputset.html`.

How to Determine if Cpusets are Installed

You can issue several commands to determine whether cpusets are installed on your system, as follows:

1. Use the `grep(1)` command to search the `/proc/filesystems` for cpusets, as follows:

```
% grep cpuset /proc/filesystems
nodev cpuset
```

2. Determine if `cpuset tasks` file is present on your system by changing directory to `/dev/cpuset` and listing the content of the directory, as follows:

```
% cd /dev/cpuset
Directory: /dev/cpuset
```

```
% ls
cpu_exclusive cpus mem_exclusive mems notify_on_release
pagecache_list pagecache_local slabcache_local tasks
```

3. If the `/dev/cpuset/tasks` file is not present on your system, it means the `cpuset` file system is not mounted (usually, it is automatically mounted when the system was booted). As root, you can mount the `cpuset` file system, as follows:

```
% mount -t cpuset cpuset /dev/cpuset
```

Fine-grained Control within Cpusets

Within a single cpuset, use facilities such as `taskset(1)`, `dplace(1)`, `first-touch` memory placement, `pthread`s, `sched_setaffinity` and `mbind` to manage processor and memory placement to a more fine-grained level.

The user-level bitmask library supports convenient manipulation of multiword bitmasks useful for CPUs and memory nodes. This bitmask library is required by and designed to work with the cpuset library. You can find information on the bitmask library on your system at

`/usr/share/doc/packages/libbitmask/libbitmask.html`.

Cpuset Interaction with Other Placement Mechanism

The Linux 2.6 kernel supports additional processor and memory placement mechanisms, as follows:

Note: Use the `uname(1)` command to print out system information to make sure you are running the Linux 2.6.x `sn2` kernel, as follows:

```
% uname -r -s
Linux 2.6.16.14-6-default
```

-
- The `sched_setaffinity(2)` and `sched_getaffinity(2)` system calls set and get the CPU affinity mask of a process. This determines the set of CPUs on which the process is eligible to run. The `taskset(1)` command provides a command line utility for manipulating the CPU affinity mask of a process using these system calls. For more information, see the appropriate man page.
 - The `set_mempolicy` system call sets the NUMA memory policy of the current process to *policy*. A NUMA machine has different memory controllers with different distances to specific CPUs. The memory *policy* defines in which node memory is allocated for the process.

The `get_mempolicy(2)` system retrieves the NUMA policy of the calling process or of a memory address, depending on the setting of *flags*. The `numactl(8)` command provides a command line utility for manipulating the NUMA memory policy of a process using these system calls.

- The `mbind(2)` system call sets the NUMA memory policy for the pages in a specific range of a task's virtual address space.

Cpusets are designed to interact cleanly with other placement mechanisms. For example, a workload manager can use cpusets to control the CPU and memory placement of various jobs; while within each job, these other kernel mechanisms are used to manage placement in more detail. It is possible for a workload manager to change a job's cpuset placement while preserving the internal CPU affinity and NUMA memory placement policy, without requiring any special coding or awareness by the affected job.

Most jobs initialize their placement early in their time slot, and jobs are rarely migrated until they have been running for a while. As long as a workload manager does **not** try to migrate a job at the same time as it is adjusting its own CPU or memory placement, there is little risk of interaction between cpusets and other kernel placement mechanisms.

The CPU and memory node placement constraints imposed by cpusets always override those of these other mechanisms.

Calls to the `sched_setaffinity(2)` system call automatically mask off CPUs that are not allowed by the affected task's cpuset. If a request results in all the CPUs being masked off, the call fails with `errno` set to `EINVAL`. If some of the requested CPUs are allowed by the task's cpuset, the call proceeds as if only the allowed CPUs were requested. The disallowed CPUs are silently ignored. If a task is moved to a different cpuset, or if the CPUs of a cpuset are changed, the CPU affinity of the affected task or tasks is lost. If a workload manager needs to preserve the CPU affinity of the tasks in a job that is being moved, it should use the `sched_setaffinity(2)` and `sched_getaffinity(2)` calls to save and restore each affected task's 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 from the `libbitmask` bitmasks used by `libcpuset`, so some coding will be required to convert between the two, in order to calculate and preserve cpuset relative CPU affinity.

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

Cpusets and Thread Placement

If your job uses the placement mechanisms described in "Cpuset Interaction with Other Placement Mechanism" on page 6 and operates under the control of a workload manager, you **cannot** guarantee that a migration will preserve placement done using the mechanisms. These 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 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.

The `libcputset` library provides the following mechanisms to support cpuset relative thread placement that is robust even if the job is being migrated using a batch scheduler.

If your job needs to pin a thread to a single CPU, you can use the convenient `cpuset_pin` function. This is the most common case. For more information on `cpuset_pin`, see "Basic Cpuset Library Functions" on page 46.

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

- `cpuset_get_placement` (see "`cpuset_get_placement`" on page 71)
- `cpuset_equal_placement` (see "`cpuset_equal_placement`" on page 72)
- `cpuset_free_placement` (see "`cpuset_free_placement`" on page 72)

Safe Job Migration and Cpusets

Jobs that make use of cpuset aware thread pinning described in "Cpusets and Thread Placement" on page 8 can be safely migrated to a different cpuset or have the CPUs or memory nodes of the cpuset safely changed without destroying the per-thread placement done within the job.

Procedure 1-1 Safe Job Migration Between Cpusets

To safely migrate a job to a different cpuset, perform the following steps:

1. Suspend the tasks in the job by sending their process group a `SIGSTOP` signal.

2. Use the `cpuset_init_pidlist` function 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` function to move the job's tasks from the old cpuset to the new cpuset.
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 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 by sending their process group a `SIGCONT` signal.

The `sched_getaffinity(2)` and `sched_setaffinity(2)` C library calls are limited by C library internals to systems with 1024 CPUs or less. To write code that will work on larger systems, you should use the `syscall(2)` indirect system call wrapper to directly invoke the underlying system call, bypassing the C library API for these calls.

The suspend and resume operation are required in order to keep tasks in the job from changing their per thread CPU placement between steps three and six. The kernel automatically migrates the per-thread memory node placement during step four. This is necessary, because there is no way for one task to modify the NUMA memory placement policy of another task. The kernel does not automatically migrate the per-thread CPU placement, as this can be handled by the user level process doing the migration.

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. This loss of 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 NUMA memory placement information 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 workload manager 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.

Cpuset File System Directories

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`, as described earlier.

The state of each cpuset is represented by small text files in the directory for the cpuset. 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 programming languages, such as `open(2)`, `read(2)`, `write(2)` and `close(2)` from the C programming library.

To view the files in a cpuset that can be either read or written, perform the following commands:

```
% cd /dev/cpuset
% ls
cpu_exclusive  memory_migrate          memory_spread_page  notify_on_release
cpus           memory_pressure        memory_spread_slab  tasks
mem_exclusive  memory_pressure_enabled mems
```

Descriptions of the files in the cpuset directory are, as follows:

Cpuset Directory File

tasks

Description

List of process IDs (PIDs) of tasks in the 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 cpuset's tasks file (with or without a trailing newline.)

Note that only one PID may be written to the tasks file at a time. If a string is written that contains

<code>notify_on_release</code>	<p>more than one PID, all but the first are ignored.</p> <p>Flag (0 or 1) - If set (1), the <code>/sbin/cpuset_release_agent</code> binary is invoked, with the name (<code>/dev/cpuset</code> relative path) of that cpuset in <code>argv[1]</code>, when the last user of it (task or child cpuset) goes away. This supports automatic cleanup of abandoned cpusets.</p>
<code>cpus</code>	<p>List of CPUs that tasks in the cpuset are allowed to use. For a description of the format of the <code>cpus</code> file, see "List Format" on page 20. The CPUs allowed to a cpuset may be changed by writing a new list to its <code>cpus</code> file. Note, however, such a change does not take affect until the PIDs of the tasks in the cpuset are rewritten to the cpuset's <code>tasks</code> file.</p>
<code>cpu_exclusive</code>	<p>Flag (0 or 1) - The <code>cpu_exclusive</code> flag, when set, automatically defines scheduler domains. The kernel performs automatic load balancing of active threads on available CPUs more rapidly within a scheduler domain than it does across scheduler domains. By default, this flag is off (0). Newly created cpusets initially default this flag to off (0).</p>
<code>mems</code>	<p>List of memory nodes that tasks in the cpuset are allowed to use. For a description of the format of the <code>mems</code> file, see "List Format" on page 20.</p>
<code>mem_exclusive</code>	<p>Flag (0 or 1) - The <code>mem_exclusive</code> flag, when set, automatically defines</p>

	constraints for kernel internal memory allocations. Allocations of user space memory pages are strictly confined by the allocating task's cpuset. Allocations of kernel internal pages are only confined by the nearest enclosing cpuset that is marked <code>mem_exclusive</code> . By default, this flag is off (0). Newly created cpusets also initially default this flag to off (0).
<code>memory_migrate</code>	Flag (0 or 1). If set (1), memory migration is enabled. For more information, see "Memory Migration" on page 19.
<code>memory_pressure</code>	A measure of how much memory pressure the tasks in this cpuset are causing. Always has value zero (0) unless <code>memory_pressure_enabled</code> is enabled in the top cpuset. This is a read-only file. The <code>memory_pressure</code> mechanism makes it easy to detect when the job in a cpuset is running short of memory and needing to page memory out to swap. For more information, see "Memory Pressure of a Cpuset" on page 16.
<code>memory_pressure_enabled</code>	Flag (0 or 1). This file is only present in the root cpuset, normally at <code>/dev/cpuset</code> . If set (1), <code>memory_pressure</code> calculations are enabled for all cpusets in the system. For more information, see "Memory Pressure of a Cpuset" on page 16.
<code>memory_spread_page</code>	Flag (0 or 1). If set (1), the kernel page cache (file system buffers) are

uniformly spread across the cpuset. For more information, see "Memory Spread" on page 18.

`memory_spread_slab` Flag (0 or 1). If set (1), the kernel slab caches for file I/O (directory and inode structures) are uniformly spread across the cpuset. For more information, see "Memory Spread" on page 18.

A new file has been added to `/proc` file system, as follows:

`/proc/pid/cpuset`

For each task (PID), list its cpuset path, relative to the root of the cpuset file system. This is a read-only file.

There are two control fields used by the kernel scheduler and memory allocation mechanism to constrain scheduling and memory allocation to the allowed CPUs. These are two fields in the `status` file of each task, as follows:

`/proc/pid/status`

<code>Cpus_allowed</code>	A bit vector of CPUs on which this task may be scheduled
<code>Mems_allowed</code>	A bit vector of memory nodes on which this task may obtain memory

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

- A task might use the `sched_setaffinity`, `mbind`, or `set_mempolicy` functions to restrain its placement to less than its cpuset.
- Various temporary changes to `cpus_allowed` values are done by kernel internal code.
- Attaching a task to a cpuset does not change its `mems_allowed` value until the next time that task needs kernel memory.

- Changing a cpuset's `cpus` value does not 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).

User space action is required to update a task's `Cpus_allowed` values after changing its cpuset. Use the `cpuset_reattach` routine to perform this update after a changing the CPUs allowed to a cpuset.

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

The confines of a cpuset can be violated after a hotplug removal that empties a cpuset, until, and unless, the system's cpuset configuration is updated to accurately reflect the new hardware configuration, and in particular, to not define a cpuset that has no CPUs still online, or no memory nodes still online. 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 task's cpuset, the kernel relaxes the cpuset, and looks for memory anywhere it can find it. It is better to violate the cpuset than stress the kernel operation.

New cpusets are created using the `mkdir` command at the shell (see "Using Cpusets at the Shell Prompt" on page 23) or via the C programming language (see Appendix A, "Cpuset Library Functions" on page 45). Old cpusets are removed using the `rmdir(1)` command. 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 task's 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 task's `mems_allowed` vector.

Exclusive Cpusets

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

A cuset that is `cpu_exclusive` has a scheduler (`sched`) domain associated with it. The `sched` domain consists of all CPUs in the current cuset 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 described in "Cuset File System Directories" on page 10 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` cuset only to be prevented by the `Cpus_allowed` mask of the task.

A cuset 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 isolating the user allocation of each job to its own cuset. To do this, construct a large `mem_exclusive` cuset 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` cuset.

Notify on Release Flag

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

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

The usual contents of the `/sbin/cpuset_release_agent` command is a simple shell script, as follows:

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

By default, the `notify_on_release` flag is off (0). Newly created cpusets inherit their `notify_on_release` flag setting from their parent cpuset. As with other flag values, 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.

Memory Pressure of a Cpuset

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 workload managers, monitoring jobs running in dedicated cpusets, to efficiently detect what level of memory pressure that job is causing.

This is useful in the following situations:

- Tightly managed systems running a wide mix of submitted jobs that may choose to terminate or re-prioritize jobs trying to use more memory than allowed on the nodes to which they are assigned.
- 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.

This mechanism provides a very economical way for the workload manager to monitor a cpuset for signs of memory pressure. It is up to the workload manager or other user code to decide when to take action to alleviate memory pressures.

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

If the `memory_pressure_enabled` flag in the top cpuset is set (1), 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. This mechanism allows batch schedulers to poll these files and detect jobs that are causing memory stress. They can then take action to avoid impacting the rest of the system with a job that is trying to aggressively exceed its allowed memory.

Note: 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.

A running average per cpuset has the following advantages:

- The system load imposed by a batch scheduler monitoring this metric is sharply reduced on large systems because this meter is per-cpuset, rather than per-task or memory region and this avoids a scan of the system-wide task list on each set of queries.
- 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 a running average, rather than an accumulating counter.
- A 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 because this meter is per-cpuset rather than per-task or memory region.

A simple, per-cpuset 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 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 (approximate) seconds. At constant rates faster than one per msec, it reaches maximum at values just under 1,000,000. At constant rates between one per msec and one per second, it stabilizes 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 is 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.

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, `memory_spread_page` flag is set, the kernel spreads 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, `memory_spread_slab` flag is set, the kernel spreads 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 the 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 memory policy or cpuset configuration. This is true 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 task's NUMA memory policy and be spread instead. Tasks using `mbind()` or `set_mempolicy()` calls to set NUMA memory policies 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 memory policy once again applies to memory page allocations.

Both `memory_spread_page` and `memory_spread_slab` are Boolean flag files. By default, they contain 0. This means the feature is off for the cpuset. If a 1 is written to this file, the named feature is turned on for the cpuset.

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 job's 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.

Memory Migration

Normally, under the default setting 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, even if the cpuset's memory placement `mems` policy subsequently changes. The default setting has the `memory_migrate` flag disabled.

When memory migration is enabled in a cpuset, if the `mems` setting of the cpuset is changed, any memory page in use by any task in the cpuset that is on a memory node no longer allowed is 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, are 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 then 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 cpuset's `mems` file is written, changing it from say memory nodes "4-7", to memory nodes "5-8", the following page migrations are done, in order, for all pages in the address space of tasks in that cpuset:

1. Migrate pages on node 7 to node 8.
2. Migrate pages on node 6 to node 7.
3. Migrate pages on node 5 to node 6.
4. Migrate pages on node 4 to node 5.

In this example, pages on any memory node other than "4 through 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.

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 hexadecimal 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. An example of the mask format is, as follows:

```
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.

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. An example of list format is, as follows:

```
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
```

Cpuset Permissions

The permissions of a cpuset are determined by the permissions of the special files and directories in the cpuset file system, normally mounted at `/dev/cpuset`.

For example, a task can put itself in some other cpuset (than its current one) if it can write the `tasks` file (see "Cpuset File System Directories" on page 10) for that cpuset (requires execute permission on the encompassing directories and write permission on that `tasks` file).

An additional constraint is applied to requests to place some other task in a cpuset. One task may not attach another task to a cpuset unless it has 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 cpuset's directory (execute permissions on the encompassing directories) and write the corresponding `cpus` or `mems` file (see "Cpuset File System Directories" on page 10).

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).

There is one minor difference between the manner in which cpuset path permissions are evaluated by `libcputset` 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 task's current directory. Therefore, such calls can successfully operate on files in or below a task's current directory, even if the task lacks search permission on some ancestor directory. Calls in `libcputset` that operate on cpuset pathnames, such as the `cpuset_query()` call, rely on `libcputset` internal conversion of all cpuset pathnames to full, root-based paths. They 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`).

CPU Scheduling and Memory Allocation for Cpusets

This section describes CPU scheduling and memory allocation for cpusets and covers these topics:

- "Linux Kernel CPU and Memory Placement Settings" on page 22
- "Manipulating Cpusets" on page 23

Linux Kernel CPU and Memory Placement Settings

The Linux kernel exposes to user space three important attributes of each task that the kernel uses to control that tasks processor and memory placement, as follows:

- The cpuset path of each task, relative to the root of the cpuset file system, is available in the file `/proc/pid/cpuset`. For each task (PID), the file lists its cpuset path relative to the root of the cpuset file system.
- The actual CPU bitmask used by the kernel scheduler to determine on which CPUs a task may be scheduled is displayed in the `Cpus_allowed` field of the file `/proc/pid/status` for that task *pid*.
- The actual memory node bitmask used by the kernel memory allocator to determine on which memory nodes a task may obtain memory is displayed in the `Mems_allowed` field of the file of the file `/proc/pid/status` for that task *pid*.

Each of the above files is read-only. You can ask the kernel to make changes to these settings by using the various cpuset interfaces and the `sched_setaffinity(2)`, `mbind(2)`, and `set_mempolicy(2)` system calls.

The `cpus_allowed` and `mems_allowed` status file values for a task may differ from the `cpus` and `mems` values defined in the cpuset directory for the task for the following reasons:

- A task might call the `sched_setaffinity`, `mbind`, or `set_mempolicy` system calls to restrain its placement to less than its cpuset.
- Various temporary changes to `cpus_allowed` status file values are done by kernel internal code
- Attaching a task to a cpuset does not change its `mems_allowed` status file value until the next time that task needs kernel memory.
- Changing the CPUs in a cpuset does not change the `cpus_allowed` status file value of the tasks attached to the cpuset until those tasks are reattached to it (to avoid a hook in the hotpath scheduler code in the kernel).

Use the `cpuset_reattach` routine to perform this update after a changing the CPUs allowed to a cpuset.

- If hotplug is used to remove all the CPUs or all the memory nodes in a cuset, the tasks attached to that cuset will have their `cpus_allowed` status file values or `mems_allowed` status file values altered to the CPUs or memory nodes when the closest ancestor to that cuset is not empty.

Manipulating Cpusets

New cpusets are created using the `mkdir(1)` command (at the shell (see Procedure 1-2 on page 23) or in C programs (see Appendix A, "Cpuset Library Functions" on page 45)). Old cpusets are removed using the `rmdir(1)` commands. The `Cpus_allowed` and `Mems_allowed` status file files are accessed using `read(2)` and `write(2)` system calls or shell commands such as `cat` and `echo`.

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

Using Cpusets at the Shell Prompt

This section describes the use of cpusets using shell commands. For information on the `cpuset(1)` command line utility, see "Cpuset Command Line Utility" on page 25. For information on using the `cpuset` library functions, see Appendix A, "Cpuset Library Functions" on page 45.

When modifying the CPUs in a cuset from the from the shell prompt, you must write the process ID (PID) of each task attached to that cuset back into the cuset's `tasks` file. When using the `libcpuset` API, use the `cpuset_reattach()` routine to perform this step. The reasons for performing this step are described in "Modifying the CPUs in a Cpuset and Kernel Processing" on page 36.

Procedure 1-2 Starting a New Job within a Cpuset

In this procedure, you will create a new cuset called `green`, assign CPUs 2 and 3 and memory node 1 to the new cuset, and start a subshell running in the cuset.

To start a new job and contain it within a cuset, perform the following steps:

1. The cpuset system is created and initialized by the kernel at system boot. You allow user space access to the cpuset system by mounting the cpuset virtual file system (VFS) at `/dev/cpuset`, as follows:

```
% mkdir /dev/cpuset
% mount -t cpuset cpuset /dev/cpuset
```

Note: If the `mkdir(1)` and/or the `mount(8)` command fail, it is because they have already been performed.

2. Create the new cpuset called `green` within the `/dev/cpuset` virtual file system using the `mkdir` command, as follows:

```
% cd /dev/cpuset
% mkdir green
% cd green
```

3. Use the `echo` command to assign CPUs 2 and 3 and memory node 1 to the `green` cpuset, as follows:

```
% /bin/echo 2-3 > cpus
% /bin/echo 1 > mems
```

4. Start a task that will be the “parent process” of the new job and attach the task to the new cpuset by writing its PID to the `/dev/cpuset/tasks` file for that cpuset.

```
/bin/echo $$ > tasks
sh
```

5. The subshell `sh` is now running in the `green` cpuset.

The file `/proc/self/cpuset` shows your current cpuset, as follows:

```
% cat /proc/self/cpuset
/green
```

6. From this shell, you can `fork`, `exec` or `clone(2)` the job tasks. By default, any child task of this shell will also be in `cpuset green`. You can list the PIDs of the tasks currently in `cpuset green` by performing the following:

```
% cat /dev/cpuset/green/tasks
4965
5043
```

In this example, PID 4965 is your shell, and PID 5043 is the `cat` command itself.

Procedure 1-3 Removing a Cpuset from the /dev/cpuset Directory

To remove the `cpuset green` from the `/dev/cpuset` directory, perform the following:

1. Use the `rmdir` command to remove a directory from the `/dev/cpuset` directory, as follows:

```
%cd /dev/cpuset
%rmdir green
```

2. To determine if you can remove the `cpuset`, you can perform the `cat` command on the `cpuset` directory `tasks` files to ensure no PIDs are listed or within an application using `libcpuset 'C'` API. You can also perform an `ls` command on the `cpuset` directory to ensure it has no subdirectories.

The `green` `cpuset` must be empty in order for you to remove it, if not a message similar to the following appears:

```
%rmdir green
rmdir: `green': Device or resource busy
```

Cpuset Command Line Utility

The `cpuset(1)` command is used to create and destroy `cpusets`, to retrieve information about existing `cpusets`, and to attach processes to `cpusets`. The `cpuset(1)` command line utility is not essential to the use of `cpusets`. This utility provides an alternative that may be convenient for some uses. Users of earlier versions of `cpusets` may find this utility familiar, though the details of the options have changed in order to reflect the current implementation of `cpusets`.

A `cpuset` is defined by a `cpuset` configuration file and a name. For a definition of the `cpuset` configuration file format, see "Cpuset Text Format" on page 35. The `cpuset` configuration file is used to list the CPUs and memory nodes that are members of the `cpuset`. It also contains any additional parameters required to define the `cpuset`. For

more information on the cpuset configuration file, see "bootcpuset.conf File" on page 31.

This command automatically handles reattaching tasks to their cpuset whenever necessary, as described in the `cpuset_reattach` routine in Appendix A, "Cpuset Library Functions" on page 45.

The `cpuset` command accepts the following options:

Action Options (choose exactly one):

- | | |
|---|--|
| <code>-c <i>csname</i>, --create=<i>csname</i></code> | Creates cpuset named <i>csname</i> using the cpuset text format (see "Cpuset Text Format" on page 35) representation read from the commands input stream. |
| <code>-m <i>csname</i>, --modify=<i>csname</i></code> | Modifies the existing cpuset <i>csname</i> to have the properties in the cpuset text format (see "Cpuset Text Format" on page 35) representation read from the commands input stream. |
| <code>-x <i>csname</i>, --remove=<i>csname</i></code> | Removes the cpuset named <i>csname</i> . A cpuset may only be removed if there are no processes currently attached to it and the cpuset has no descendant cpusets. |
| <code>-d <i>csname</i>, --dump=<i>csname</i></code> | Writes a cpuset text format representation (see "Cpuset Text Format" on page 35) of the cpuset named <i>csname</i> to the commands output stream. |
| <code>-p <i>csname</i>, --procs=<i>csname</i></code> | Lists to the commands output stream the processes (by <code>pid</code>) attached to the cpuset named <i>csname</i> . If the <code>-r</code> option is also specified, lists the <code>pid</code> of each process attached to any descendant of cpuset <i>csname</i> . |

<code>-a <i>csname</i>, --attach=<i>csname</i></code>	Attaches to the cpuset named <i>csname</i> the processes whose <code>pids</code> are read from the commands input stream, one <code>pid</code> per line.
<code>-i <i>csname</i>, --invoke=<i>csname</i></code>	Invokes a command in the cpuset named <i>csname</i> . If <code>-I</code> option is set, use that command and arguments, otherwise if the environment variable <code>\$SHELL</code> is set, use that command, otherwise, use <code>/bin/sh</code> .
<code>-w <i>pid</i>, --which=<i>pid</i></code>	Lists the name of the cpuset to which process <code>pid</code> is attached, to the commands output stream. If <code>pid</code> is zero (0), then the full cpuset path of the current task is displayed.
<code>-s <i>csname</i>, --show=<i>csname</i></code>	Prints to the commands output stream the names of the cpusets below cpuset <i>csname</i> . If the <code>-r</code> option is also specified, this recursively includes <i>csname</i> and all its descendants, otherwise it just includes the immediate child cpusets of <i>csname</i> . The cpuset names are printed one per line.
<code>-R <i>csname</i>, --reattach=<i>r</i></code>	Reattaches each task in cpuset <i>csname</i> . This is required after changing the <code>cpus</code> value of a cpuset, in order to get the tasks already attached to that cpuset to rebind to the changed CPU placement.
<code>-z <i>csname</i>, --size=<i>csname</i></code>	Prints the size of (number of CPUs in) a cpuset to the commands output stream, as an ASCII decimal newline terminated string.

`-F flist, --family=flist`

Creates a family of non-overlapping child cpusets, given an *flist* of cpuset names and sizes (number of CPUs). Fails if the total sizes exceeds the size of the current cpuset. Enter cpuset names relative to the current cpuset, and their requested size, as alternating command line arguments. For example:

```
cpuset -F foo 2 bar 6 baz 4
```

This creates three child cpusets named `foo`, `bar`, and `baz`, having 2, 6, and 4 CPUs, respectively.

This example will fail with an error message and a nonzero exit status if the current cpuset lacks at least 12 CPUs.

These cpuset names are relative to the current cpuset and will not collide with the cpuset names descendent from other cpusets. Hence two commands, running in different cpusets, can both create a child cpuset named `foo` without a problem.

Modifier Options (may be used in any combination):

`-r, --recursive`

When used with `-p` or `-s` option, applies to all descendants recursively of the named cpuset *csname*.

`-I cmd,
--invokecmd=cmd`

When used with the `-i` option, the command *cmd* is invoked, with any optional unused arguments. The following example invokes an interactive subshell in cpuset `foo`:

```
cpuset -i foo -I sh -- -i
```

The next example invokes a second `cpuset` command in `cpuset foo`, which then displays the full `cpuset` path of `foo`:

```
cpuset -i foo -I cpuset -- -w 0
```

Note: The double minus `--` is needed to end option parsing by the initial `cpuset` command.

<p><code>-f <i>fname</i></code>, <code>--file=<i>fname</i></code></p> <p><code>--</code> <code>move_tasks_from=<i>csname1</i></code> <code>--</code> <code>move_tasks_to=<i>csname2</i></code></p>	<p>Uses file named <i>fname</i> for command input or output stream, instead of <code>stdin</code> or <code>stdout</code>.</p> <p>Move all tasks from <code>cpuset <i>csname1</i></code> to <code>cpuset <i>csname2</i></code>. Retries up to ten times to move all tasks, in case it is racing against parallel attempts to fork or add tasks into <code>cpuset <i>csname1</i></code>. Fails with nonzero exit status and an error message to <code>stderr</code> if unable to move all tasks out of <i>csname1</i>.</p>
--	--

Help Option (overrides all other options):

<code>-h, --help</code>	Displays command usage
-------------------------	------------------------

Notes

The *csname* of `"/` (slash) refers to the top `cpuset`, which encompasses all CPUs and memory nodes in the system. The *csname* of `."` (dot) refers to the `cpuset` of the current task. If a *csname* begins with the `"/` (slash) character, it is resolved relative to the top `cpuset`, otherwise it is resolved relative to the `cpuset` of the current task.

The 'command input stream' and 'command output stream' refer to the `stdin` (file descriptor 0) and `stdout` (file descriptor 1) of the command, unless the `-f` option is specified, in which case they refer to the file specified to `-f` option. Specifying the file name `-` to the `-f` option, as in `-f -`, is equivalent to not specifying the `-f` option at all.

Exactly **one** of the action options must be specified. They are, as follows:

```
-c, -m, -x, -d, -p, -a, -i, -w, -s, -R
```

The additional modifier options may be specified in any order. All modifier options are evaluated first, before the action option. If the help option is present, no action option is evaluated. The modifier options are, as follows:

```
-r, -I, -f
```

Boot Cpuset

You can use the `bootcpuset(8)` command to create a “boot”cpuset during the system boot that you can use to restrict the default placement of almost all UNIX processes on your system. You can use the `bootcpuset` to reduce the interference of system processes with applications running on dedicated cpusets.

The default cpuset for the `init` process, classic UNIX daemons, and user login shells is the root cpuset that contains the entire system. For systems dedicated to running particular applications, it is better to restrict `init`, the kernel daemons, and login shells to a particular set of CPUs and memory nodes called the `bootcpuset`.

This section covers the following topics:

- "Creating a Bootcpuset" on page 30
- "bootcpuset.conf File" on page 31

Creating a Bootcpuset

This section describes how to create a bootcpuset.

Procedure 1-4 Creating a Bootcpuset

To create a bootcpuset, perform the following steps:

1. Create `/etc/bootcpuset.conf` file with values to restrict system processes to the CPUs and memory nodes appropriate for your system, similar to the following:

```
cpus 0-7  
mems 0
```

2. In the `/boot/efi/efi/SuSE/elilo.conf` file (or a similar path to the `elilo.conf` file), add the following string using the instructions that follow to the `append` argument for the kernel you are booting:

```
append="init=/sbin/bootcpuset"
```

For RHEL, you need to add the following string to the kernel line of the `grub.conf` file:

```
init=/sbin/bootcpuset
```

You should not directly edit the `elilo.conf` file because YaST and the install kernel tools may overwrite your changes when kernels are updated. Instead, edit the `/etc/elilo.conf` file and run the `elilo` command. This will place an updated `elilo.conf` in `/boot/efi/efi/SuSE` and the system will know about the change for new kernels or YaST runs.

3. Reboot your system.

Subsequent system reboots will restrict most processes to the `bootcpuset` defined in `/etc/bootcpuset.conf`.

bootcpuset.conf File

The `/etc/bootcpuset.conf` file describes what CPUs and memory nodes are to be in the `bootcpuset`. The kernel boot command line option `init` is used to invoke the `/sbin/bootcpuset` binary ahead of the `/sbin/init` binary, using the `elilo` syntax: `append="init=/sbin/bootcpuset"`.

When invoked with `pid=1`, the `/sbin/bootcpuset` binary does the following:

- Sets up a `bootcpuset` (configuration defined in the `/etc/bootcpuset.conf` file).
- Attaches itself to this `bootcpuset`.
- Attaches any unpinned kernel threads to it.
- Invokes an `exec` call to execute `/sbin/init`, `/etc/init`, `/bin/init` or `/bin/sh`.

A kernel thread is deemed to be unpinned (third bullet in the list above) if its `Cpus_allowed` value (as listed in that thread's `/proc/pid/status` file for the `Cpus_allowed` field) allows running on all online CPUs. Kernel threads that are restricted to some proper subset of CPUs are left untouched, under the assumption

that they have a good reason to be running on those restricted CPUs. Such kernel threads as migration (to handle moving threads between CPUs) and `ksoftirqd` (to handle per-CPU work off interrupts) must be pinned to each CPU or each memory node.

Comments in the `/etc/bootcpuset.conf` configuration file begin with the pound (#) character and extend to the end of the line. After stripping comments, the `bootcpuset` command examines the first white space separated token on each line.

If the first token on the line matches `mems` or `mem` (case insensitive match) then the second token on the line is written to the `/dev/cpuset/boot/mems` file.

If the first token on the line matches `cpus` or `cpu` (case insensitive match), then the second token is written to the `/dev/cpuset/boot/cpus` file.

If the first token in its entirety matches (case insensitive match) "verbose", the `bootcpuset` command prints a trace of its actions to the console. A typical such trace has 20 or 30 lines, detailing the steps taken by `/sbin/bootcpuset` and is useful in understanding its behavior and analyzing problems. The `bootcpuset` command ignores all other lines in the `/etc/bootcpuset.conf` configuration file.

Configuring a User Cpuset for Interactive Sessions

You can have a *boot* cpuset running the traditional daemon and server tasks and a second cpuset to hold interactive `telnet`, `rlogin` and/or secure shell (SSH) user sessions called the *user* cpuset.

Creating a user cpuset provides additional isolation between interactive user login sessions and essential system tasks. For example, a user process in the user cpuset consuming excessive CPU, system file buffer cache, or memory resources will not seriously impact essential system services in the boot cpuset.

The following `init` script provides an example of how you can set up user cpuset. It runs as one of the last `init` scripts when the system is being booted.

If the system has a boot cpuset configured, the following script creates a second cpuset called *user* and place the `sshd` and `xinetd` daemons that create interactive login sessions in this new user cpuset. The user cpuset configuration is defined in the `/etc/usercpuset.conf` file.

To isolate the boot cpuset from the user cpuset, the set of cpus and mems values in the `/etc/bootcpuset.conf` file should not overlap the cpus and mems values in the `/etc/usercpuset.conf` file.

Procedure 1-5 Configuring a User Cpuset for Interactive Sessions

To implement a `usercpuset` for interactive sessions, perform the following:

1. Add the `/etc/usercpuset.conf` file.
2. Add the `/etc/init.d/usercpuset` script (see below).
3. Perform the following command:

```
% chkconfig --add usercpuset
```

Instructions for using this script are included in comments within the script, as follows:

```
#!/bin/sh
# /etc/init.d/usercpuset
#
# ----- RedHat -----
# chkconfig: 2345 016 99
# description: usercpuset: Put login sessions in user cpuset
# ----- SuSe -----
### BEGIN INIT INFO
# Provides: usercpuset
# Required-Start: sshd xinetd
# Required-Stop:
# Default-Start: 3 5
# Default-Stop: 0 1 2 6
# Description: Put login sessions in user cpuset
### END INIT INFO
# -----
#
# This init script creates a 'user' cpuset and places the login servers
# (sshd and xinetd) in that cpuset,
#
# This script presumes you also have a 'boot' cpuset configured,
# and does nothing if you don't.
#
# By using this init script, one can isolate essential daemon and
# server tasks from interactive user login sessions in separate
```

1: Cpusets on Linux

```
# cpusets.
#
# To use this usercpuset init script:
#
# 1) Using your editor, create a file /etc/init.d/usercpuset
#    containing this script.
# 2) Run the command the following command to insert this script
#    into the sequence of init scripts executed during system boot.
# SLES --> "insserv usercpuset"
# RHEL --> "chkconfig --add usercpuset"
# 3) Create a /etc/usercpuset.conf, in the "Cpuset Text Format"
#    described in the libcpuset(3) man page, describing what CPUs
#    ("cpus") and memory nodes ("mems") are to be used by the
#    user cpuset.
# 4) Also configure and enable a boot cpuset, as documented in
#    the bootcpuset(8) man page.
# 5) Beginning with the next system reboot, login sessions under
#    either the SSH daemon (sshd) or xinetd (telnet, rlogin) will
#    be started in the 'user' cpuset, while other daemons and
#    system services, including the consolelogin, will be in the
#    'boot' cpuset.
# 6) If you did not do both steps (3) and (4) above, then this
#    usercpuset script will do nothing, quietly, with no harm.

echo "### Creating user cpuset ###"

CPUSET_CMD=/usr/bin/cpuset

# Define the 'mems' and 'cpus' for user cpuset in this configuration file:
CONF=/etc/usercpuset.conf

USERCPUSET=/user

test -x $CPUSET_CMD || exit 5
test -r $CONF || exit 6

# Skip this if we didn't have a boot cpuset

test -d /dev/cpuset/boot || exit 7
if [ -f $CONF ]; then
    $CPUSET_CMD -c $USERCPUSET -f $CONF
```



```
fi

if [ -r /etc/rc.status ]; then
# SuSE
SSHD_PIDFILE=/var/run/sshd.init.pid
status=$?
else
# not SuSE
SSHD_PIDFILE=/var/run/sshd.pid
status=$?
fi

# sshd
$CPUSET_CMD -a $USERCPUSET < $SSHD_PIDFILE

# xinetd
echo $(pidof xinetd) | $CPUSET_CMD -a $USERCPUSET
```

Cpuset Text Format

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

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 and a form for representing cpusets that users of earlier versions of cpusets will find familiar.

The exported cpuset text format has fewer directives than earlier Linux versions. Additional directives may be added in the future.

The cpuset text format supports one directive per line. Comments begin with the pound 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:

<code>cpus</code>	Specifies 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.
<code>mems</code>	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.
<code>cpu_exclusive</code>	The <code>cpu_exclusive</code> flag is set.
<code>mem_exclusive</code>	The <code>mem_exclusive</code> flag is set.
<code>notify_on_release</code>	The <code>notify_on_release</code> 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.

See the `libcputset` routines `cpuset_import` and `cpuset_export` to handle converting the internal `struct cpuset` representation of cpusets to (export) and from (import) this text representation.

For information on manipulating cpuset text files at the shell prompt or in shell scripts using the `cpuset(1)` command, see "Cpuset Command Line Utility" on page 25.

Modifying the CPUs in a Cpuset and Kernel Processing

In order to minimize the impact of cpusets on critical kernel code, such as the scheduler, and due to the fact that the Linux 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 and is described in the following paragraphs.

When a cpuset has its memory nodes modified, for each task attached to that cpuset, the next time that the kernel attempts to allocate a page of memory for a particular task, the kernel notices the change in the task's cpuset, and updates its per-task memory placement to remain within the new cpusets memory placement. If the task was using memory policy `MPOL_BIND` and the nodes to which it was bound overlaps with its new cpuset, the task continues to use whatever subset of `MPOL_BIND` nodes that 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 cgroup, the task is essentially treated as if it was `MPOL_BIND` bound to the new cgroup (even though its NUMA placement, as queried by the `get_mempolicy()` routine, does not change). If a task is moved from one cgroup to another, the kernel adjusts the task's memory placement, as above, the next time that the kernel attempts to allocate a page of memory for that task.

When a cgroup has its CPUs modified, each task using that cgroup **does not change** its behavior automatically. In order to minimize the impact on the critical kernel scheduling code, tasks continue to use their prior CPU placement until they are rebound to their cgroup by rewriting their PID to the `tasks` file of their cgroup. If a task is moved from one cgroup to another, its CPU placement is updated in the same way as if the task's PID is rewritten to the `tasks` file of its current cgroup.

In summary, the memory placement of a task whose cgroup 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 task's PID is rewritten to the `tasks` file of its cgroup. The delay in rebinding a task's memory placement is necessary because the kernel does not support one task changing memory placement of another task. The added user level step in rebinding a task's CPU placement is necessary to avoid impacting the scheduler code in the kernel with a check for changes in a task's processor placement.

Using Cgroups 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, cgroups also provide a convenient mechanism to control the use of Hyper-Threading Technology.

Some jobs achieve better performance by using both of the Hyper-Threaded sides, A and B, of a processor core, and some run better by 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, you can specify a cpuset that contains both sides of a processor core or a cpuset that contains just one side from a processor core.

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    # the even numbered CPUs 0, 2, 4, ... 126
mems 0-63      # all memory nodes 0, 1, 2, ... 63
```

To create a cpuset called `foo` and run a job 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 of the processor cores of an HT enabled system, use the following entry in your cpuset configuration file:

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

The examples above assume that the CPUs are uniformly numbered with the even numbers for the A side and odd numbers for the B side of the processors cores. This is usually the case, but not guaranteed. You can still place a job on a system that is not uniformly numbered. Currently, it involves a longer argument list to the `cpus` option, that is, you must explicitly list the desired CPUs.

If you are using a `bootcpuset` to keep other tasks confined, you do not need to create a separate cpuset with just the B side CPUs to avoid having some tasks running on the B sides of the processor cores. If there is no cpuset for the B sides of the processor cores, except the all encompassing root cpuset, and if only root can put tasks in the root cpuset, then no one other tasks can run on the B sides.

You can use the `dplace(1)` command to manage more detailed placement of job tasks within a cpuset. Since the `dplace` command numbering of CPUs is relative to the cpuset, it does not affect the `dplace` configuration. This is true in the case where the cpuset includes both sides of Hyper-Threaded cores, just one side of the Hyper-Threaded cores, or even is on a system that does not support Hyper-threading.

Typically, the logical numbering of CPUs puts the even numbered CPUs on the A sides of processor cores and the odd numbered CPUs on the B sides. You can easily specify that only every other side is used using the stride suffix `":2"`, described above. If the CPU number range starts with an even number, the A sides of the processor cores are used. If the CPU range starts with an odd number, the B sides of the processor cores are used.

Procedure 1-6 Configuring a System with Hyper-Threaded Cores

To setup a job to run only on the A sides of the system's Hyper-Threaded cores and to ensure that no other tasks run on the B sides (they remain idle), perform the following steps:

1. Define a bootcpuset to restrain the kernel, system daemon, and user login session threads to a designated set of CPUs.
2. Create a cpuset that includes on the A sides of the processors to be used for this job. (Either a system administrator or batch scheduler with root permission).
3. Make sure no cpuset is created using the B side CPUs in these processors to prevent disruptive tasks from running on the corresponding B side CPUs. (Either a system administrator or batch scheduler with root permission).

If you use a bootcpuset to confine the traditional UNIX load processes, nothing will run on the other CPUs in the system, except when those CPUs are included in a cpuset to which a job has been assigned. These CPUs are of course in the root cpuset, however, this cpuset is normally only usable by a system administrator or batch scheduler with root permissions. This prevents any user without root permission from running a task on those CPUs, unless an administrator or service with root permission allows it. For more information, see "Boot Cpuset" on page 30.

A `ps(1)` or `top(1)` invocation will show a handful of threads on unused CPUs. 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, the kernel threads on that CPU will be almost always idle.

Cpuset Programming Model

The programming model for this version of cpusets is an extension of the cpuset model provided on IRIX and earlier versions of SGI Linux.

The flat name space of earlier cpuset versions on SGI systems is extended to a hierarchical name space. This will become more important as systems become larger. The 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 therefore the usual file system path to this root cpuset, is `/dev/cpuset`.

Optionally, a "boot" cpuset may be created, at `/dev/cpuset/boot`, to include typically just a one or a few CPUs and memory nodes. A typical use for a "boot" cpuset is to contain the general purpose UNIX daemons and login sessions, while reserving the rest of the system for running specific major applications on dedicated cpusets. For more information, see "Boot Cpuset" on page 30.

Moved tasks do not have the memory they might have allocated on their old nodes moved to the new nodes. On kernels that support such memory migration, use the [optional] `cpuset_migrate` to move allocated memory as well.

Cpusets have a permission structure which determines which users have rights to query, modify, and attach to any given cpuset. Rights are based on the hierarchical model provided by the underlying Linux 2.6 kernel cpuset file system.

To create a cpuset from within a C language application, your program obtains a handle to a new `struct cpuset`, sets the desired attributes via that handle, and issues a `cpuset_create()` call to create the desired cpuset and bind it to the

specified name. Your program can also issue calls to list by name what cpusets exist, query their properties, move tasks between cpusets, list what tasks are currently attached to a cpuset, and delete cpusets.

The names of cpusets in this C library are always relative to the root cpuset mount point, typically `/dev/cpuset`. For more information on the `libcputset` C language application programming interface (API) functions, see Appendix A, "Cpuset Library Functions" on page 45.

System Error Messages

The Linux kernel implementation of cpusets sets `errno` to specify the reason for a failed system call that affects cpusets. These `errno` values are available when a cpuset library call fails. They can be displayed by shell commands used to directly manipulate files below the `/dev/cpuset` directory and can be displayed by the `cpuset(1)` command.

The possible `errno` settings and their meaning when set on a failed cpuset call are, as follows:

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 <code>/dev/cpuset</code> .
ENOMEM	Insufficient memory is available.
EBUSY	Attempted <code>cpuset_delete()</code> on a cpuset with attached tasks.
EBUSY	Attempted <code>cpuset_delete()</code> on a cpuset with child cpusets.
ENOENT	Attempted <code>cpuset_create()</code> in a parent cpuset that does not exist.
EEXIST	Attempted <code>cpuset_create()</code> for a cpuset that already exists.
E2BIG	Attempted a <code>write(2)</code> 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 <code>cpuset_move()</code> a nonexistent task.
EACCES	Attempted to <code>cpuset_move()</code> a task that the process lacks permission to move.
ENOSPC	Attempted to <code>cpuset_move()</code> a task to an empty cpuset.
EINVAL	The <code>relcpu</code> argument to <code>cpuset_pin()</code> function is out of range (not between "zero" and " <code>cpuset_size() - 1</code> ").
EINVAL	Attempted to change a cpuset in a way that would violate a <code>cpu_exclusive</code> or <code>mem_exclusive</code> attribute of that cpuset or any of its siblings.
EINVAL	Attempted to write an empty <code>cpus</code> or <code>mems</code> bitmask to the kernel. The kernel creates new cpusets (using the <code>mkdir</code> function) with empty <code>cpus</code> and <code>mems</code> files 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 <code>cpus</code> or <code>mems</code> files of a cpuset.
EIO	Attempted to <code>write(2)</code> a string to a cpuset tasks file that does not begin with an ASCII decimal integer.
ENOSPC	Attempted to <code>write(2)</code> a list to a <code>cpus</code> file that did not include any online CPUs.
ENOSPC	Attempted to <code>write(2)</code> a list to a <code>mems</code> file that did not include any online memory nodes.
EACCES	Attempted to add a CPUS or memory resource to a cpuset that is not already in its parent.
EACCES	Attempted to set the <code>cpu_exclusive</code> or <code>mem_exclusive</code> flag on a cpuset whose parent lacks the same setting.
EBUSY	Attempted to remove a CPU or memory resource 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 <code>/proc/pid/cpuset</code> file for a cpuset path that was longer than the kernel page size.

NUMA Tools

You can use the `dlook(1)` and `dplace(1)` tools to improve the performance of processes running on your SGI nonuniform memory access (NUMA) machine. You can use `dlook(1)` to find out where in memory the operating system is placing your application's pages and how much system and user CPU time it is consuming. You can use the `dplace(1)` command to bind a related set of processes to specific CPUs or nodes to prevent process migration. This can improve the performance of your application since it increases the percentage of memory accesses that are local.

The `taskset(1)` command is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity; the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

Note: Information about these commands and memory locality and application performance, in general, can be found in the *Linux Application Tuning Guide*.

Cpuset Library Functions

This appendix describes the `libcpuset` C programming application programming interface (API) functions and covers the following topics:

- "Extensible Application Programming Interface" on page 45
- "Basic Cpuset Library Functions" on page 46
- "Advanced Cpuset Library Functions" on page 49

Extensible Application Programming Interface

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 runtime:

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

It returns a function pointer or NULL if function name is not recognized.

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

However, if you 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 runtime, not via preprocessor macros.

Some functions in Advanced Cpuset Library Functions are marked `[optional]`. (see "Advanced Cpuset Library Functions" on page 49). 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 has function signature of pointer to 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, your resulting program will not be able to link with any version of `libcputset.so` that does not define that particular function.

Basic Cpuset Library Functions

The basic cpuset API provides functions usable from a C program for the 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 multithreaded 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.

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 (see "Advanced Cpuset Library Functions" on page 49). These basic functions do not provide any essential new capability. They are implemented using the advanced function 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`, the `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:

<code>cpuset_pin</code>	Pins the current thread to a CPU, preferring local memory
<code>cpuset_size</code>	Returns the number of CPUs that are in the current tasks cpuset
<code>cpuset_where</code>	Returns on which CPU in current tasks cpuset did the task most recently execute
<code>cpuset_unpin</code>	Removes the affect of <code>cpuset_pin</code> , lets the task have run of its entire cpuset

`cpuset_pin`

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

Pins 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 pins 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 does not support cpusets (`ENOSYS`) and running when the cpuset file system is not mounted at `/dev/cpuset` (`ENODEV`).

cpuset_size

```
int cpuset_size();
```

Returns 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 does not support cpusets (ENOSYS) and running when the cpuset file system is not mounted at `/dev/cpuset` (ENODEV).

cpuset_where

```
int cpuset_where();
```

Returns 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 does not support cpusets (ENOSYS) and running when the cpuset file system is not mounted at `/dev/cpuset` (ENODEV).

cpuset_unpin

```
int cpuset_unpin();
```

Removes the CPU and memory pinning effects 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. Returns -1 on error, 0 on success.

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

Advanced Cpuset Library Functions

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

These functions primarily deal with the following three entities:

- `struct cpuset *` structure
- system cpusets
- tasks

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

Actual kernel cpusets are created under the `/dev/cpuset` directory, which is the usual mount point of the kernel's virtual cpuset filesystem. These cpusets are visible to all tasks 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 task's 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 task's 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, 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 or the cpuset file system mounted 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`, 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:

Allocate and free struct cpuset * structure

- `cpuset_alloc` - Returns handle to newly allocated `struct cpuset *` structure
- `cpuset_free` - Discards no longer needed `struct cpuset *` structure

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 * Structure

- `cpuset_setcpus` - Specifies CPUs in cpuset
- `cpuset_setmems` - Specifies memory nodes in cpuset
- `cpuset_set_iopt` - Specifies an integer value option of cpuset

- `cpuset_set_sopt` - [optional] Specifies a string value option of `cpuset`

Query various attributes of a `struct cpuset` * Structure

- `cpuset_getcpus` - Queries CPUs in `cpuset`
- `cpuset_getmems` - Queries 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_set_sopt` - [optional] Species a string value option of `cpuset`

Local CPUs and memory nodes

- `cpuset_localcpus` - Queries the CPUs local to specified memory nodes
- `cpuset_localmems` - Queries the memory nodes local to specified CPUs
- `cpuset_cpumemdist` - [optional] Hardware distance from CPU to memory node
- `cpuset_cpu2node` - Returns number of memory node closed to specified CPU
- `cpuset_addr2node` - Return number of memory node holding page at specified address.

Create, delete, query, modify, list, and examine cpusets

- `cpuset_create` - Creates a named `cpuset` as specified by `struct cpuset` * structure
- `cpuset_delete` - Deletes the specified `cpuset` (if empty)
- `cpuset_query` - Sets the `struct cpuset` structure to settings of specified `cpuset`
- `cpuset_modify` - Modifies the settings of a `cpuset` to those specified in a `struct cpuset` structure
- `cpuset_getcpusetpath` - Gets path of a tasks (0 for current) `cpuset`
- `cpuset_cpusetofpid` - Sets the `struct cpuset` structure to settings of `cpuset` of specified task
- `cpuset_mountpoint` - Returns path at which `cpuset` filesystem is mounted

- `cpuset_collides_exclusive` - [optional] True, if it would collide an exclusive

List tasks (pids) currently attached to a cpuset

- `cpuset_init_pidlist` - Initializes a list of tasks (pids) attached to a cpuset
- `cpuset_pidlist_length` - Returns number of elements in a list of `pid`
- `cpuset_get_pidlist` - Returns i'th element of a list of `pids`
- `cpuset_free_pidlist` - Deallocates a list of `pids`

Attach tasks to cpusets

- `cpuset_move` - Moves task (0 for current) to a cpuset
- `cpuset_move_all` - Moves all tasks in a list of `pids` to a cpuset
- `cpuset_migrate` - [optional] Moves a task and its memory to a cpuset
- `cpuset_migrate_all` - [optional] Moves all tasks with memory in a list of `pids` to a cpuset
- `cpuset_reattach` - Rebinds `cpus_allowed` of each task in a cpuset after changing its `cpus`

Determine memory pressure

- `cpuset_open_memory_pressure` - [optional] Opens handle to read `memory_pressure`
- `cpuset_read_memory_pressure` - [optional] Reads cpuset current `memory_pressure`
- `cpuset_close_memory_pressure` - [optional] Closes handle to read `memory_pressure`

Map between cpuset relative and system-wide CPU and memory node numbers

- `cpuset_c_rel_to_sys_cpu` - Maps cpuset relative CPU number to system wide number
- `cpuset_c_sys_to_rel_cpu` - Maps system-wide CPU number to cpuset relative number
- `cpuset_c_rel_to_sys_mem` - Maps cpuset relative memory node number to system wide number

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

Placement operations for detecting cpuset migration

- `cpuset_get_placement` - [optional] Returns the current placement of task `pid`
- `cpuset_equal_placement` - [optional] True, if two placements are equal
- `cpuset_free_placement` - [optional] Free placement

Bind to a CPU or memory node within the current cpuset

- `cpuset_cpupbind` - Binds to a single CPU within a cpuset (uses `sched_setaffinity(2)`)
- `cpuset_latestcpu` - Most recent CPU on which a task has executed
- `cpuset_membind` - Binds 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` - Exports cpuset settings to a text file
- `cpuset_import` - Imports cpuset settings from a text file

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

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

Cpuset Library Functions Calling Sequence

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, as follows:

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

Note: Some functions are marked [optional]. For an explanation, see "Extensible Application Programming Interface" on page 45.

`cpuset_alloc`

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

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

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

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 noticeable 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`.

cpuset_free

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

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

cpuset_cpus_nbits

```
int cpuset_cpus_nbits();
```

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

cpuset_mems_nbits

```
int cpuset_mems_nbits();
```

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

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).

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).

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 nonzero value as equivalent to a value of one (1).

The following `optionname` values are recognized:

- `cpu_exclusive` - Sibling cpusets not allowed to overlap cpus (see "Exclusive Cpusets" on page 15)
- `mem_exclusive` - Sibling cpusets not allowed to overlap mems (see "Exclusive Cpusets" on page 15)
- `notify_on_release` - Invokes `/sbin/cpuset_release_agent` when cpuset released (see "Notify on Release Flag" on page 15)
- `memory_migrate` - Causes memory pages to migrate to new mems (see "Memory Migration" on page 19)
- `memory_spread_page` - Causes kernel buffer (page) cache to spread over cpuset (see "Memory Spread" on page 18)
- `memory_spread_slab` - Causes kernel file I/O data (directory and inode slab caches) to spread over cpuset (see "Memory Spread" on page 18)

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 the `cpuset_function()` to invoke it, as follows:

```
/* fp has function signature of pointer to cpuset_set_sopt() */
int (*fp)(struct cpuset *cp, const char *optionname, const char *value);
fp = cpuset_function("cpuset_set_sopt");
if (fp) {
    fp( ... );
} else {
    puts ("cpuset_set_sopt not supported");
}
```

cpuset_getcpus

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

Queries CPUs in cpuset `cp`, by writing them to the bitmask `cpus`. 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).

cpuset_getmems

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

Queries 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).

cpuset_cpus_weight

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

Queries the 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.

cpuset_mems_weight

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

Queries the 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.

cpuset_get_iopt

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

Queries the value of integer option `optionname` in cpuset `cp`.

Returns value of `optionname` is recognized, else returns -1. Integer values in an uninitialized cpuset have value 0. The following `optionname` values are recognized:

- `cpu_exclusive` - Sibling cpusets not allowed to overlap cpus (see "Exclusive Cpusets" on page 15)

- `mem_exclusive` - Sibling cpusets not allowed to overlap mems (see "Exclusive Cpusets" on page 15)
- `notify_on_release` - Invokes `/sbin/cpuset_release_agent` when cpuset released (see "Notify on Release Flag" on page 15)
- `memory_migrate` - Causes memory pages to migrate to new mems (see "Memory Migration" on page 19)
- `memory_spread_page` - Causes kernel buffer (page) cache to spread over cpuset (see "Memory Spread" on page 18)
- `memory_spread_slab` - Causes kernel file I/O data (directory and inode slab caches) to spread over cpuset (see "Memory Spread" on page 18)

`cpuset_get_sopt`

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

Queries the value of string option `optionname` in cpuset `cp`.

Returns pointer to value of `optionname` 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, as follows:

```
/* fp has function signature of pointer to cpuset_get_sopt() */
int (*fp)(struct cpuset *cp, const char *optionname);
fp = cpuset_function("cpuset_get_sopt");
if (fp) {
    fp( ... );
} else {
    puts ("cpuset_get_sopt not supported");
}
```

`cpuset_localcpus`

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

Queries the CPUs local to specified memory nodes `mems`, by writing them to the bitmask `cpus`.

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

cpuset_localmems

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

Queries the memory nodes local to specified CPUs `cpus`, by writing them to the bitmask `mems`.

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

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 26, 2003

This is an [optional] function. Use `cpuset_function()` to invoke it.

cpuset_cpu2node

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

Returns number of memory node closest to CPU `cpu`. For NUMA architectures (as of this writing), this commonly 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, it would be the node number of the memory node closest to or preferred by that `cpu`.

cpuset_create

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

Creates a cpuset at the specified `cpusetpath`, as described in the provided `struct cpuset *cp` structure. 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, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

The default behavior of the `libcpuset` routine `cpuset_create()` has changed in the following way. In previous versions of SGI ProPack, the cpuset attributes `memory_spread_page`, `memory_spread_slab`, and `notify_on_release` in the newly created cpuset would default to the value zero (0) for off, regardless of the setting of these attributes in the parent cpuset. In this version of SGI ProPack, these attributes are inherited from the parent cpuset, unless explicitly set otherwise in the cpuset creation code.

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).

cpuset_delete

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

Deletes 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, 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`.

cpuset_query

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

Set `struct cpuset` structure 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, 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`.

`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, 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`.

`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 are, as follows:

<code>EACCES</code>	Permission to read or search a component of the file name was denied.
<code>EFAULT</code>	<code>buf</code> points to a bad address.

ESRCH	The pid does not exist.
E2BIG	Larger buffer needed.
ENOSYS	Kernel does not support cpusets.

cpuset_cpusetofpid

```
int cpuset_cpusetofpid(struct cpuset *cp, int 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 are, as follows:

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.

cpuset_cpusetofpid

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

Sets the struct cpuset structure 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 are, as follows:

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.

cpuset_mountpoint

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

Returns 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 (`/`), 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.

cpuset_collides_exclusive

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

Returns 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.

The `cpuset_create` function 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.

cpuset_init_pidlist

```
struct cpuset_pidlist *cpuset_init_pidlist(const char
*cpusetpath, int recursiveflag);
```

Initializes and returns 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 can 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`.

cpuset_pidlist_length

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

Returns the number of elements in `cpuset_pidlist pl`.

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*.

cpuset_free_pidlist

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

Deallocates a list of attached pids

cpuset_move

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

Moves the 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, 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`.

cpuset_move_all

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

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

If the parameter `cpusetpath` starts with a slash (/) character, 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`.

The `cpuset_move_all()` routine now returns an error if it was unable to move all the tasks requested.

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 task's file of cpuset `fromrelpath` and writing any task PIDs found there to the task's file of cpuset `torelpath`, up to ten attempts, or until the task's 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 task's 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. Therefore, 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` flag 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 this 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 are, as follows:

- `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 does not exist
- `ENOMEM` out of memory
- `ENOSYS` kernel does not support `cpusets`
- `ENOTDIR` component of `cpuset` path is not a directory
- `EPERM` lacked permission to read `cpusets` or files therein

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

`cpuset_migrate`

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

Migrates the 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, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks `cpuset`. For more information, see "Memory Migration" on page 19.

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

This is an [optional] function. Use `cpuset_function()` to invoke it.

`cpuset_migrate_all`

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

Moves 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, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks `cpuset`.

This is an [optional] function. Use `cpuset_function()` to invoke it.

`cpuset_reattach`

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

Reattaches 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, 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`.

`cpuset_open_memory_pressure`

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

Opens a file descriptor from which to read the `memory_pressure` of the cuset `cpusetpath`.

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

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

For more information, see "Memory Pressure of a Cuset" on page 16.

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

`cpuset_read_memory_pressure`

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

Reads and return the current `memory_pressure` of the cuset for which file descriptor `fd` was opened using the `cpuset_open_memory_pressure` function.

Uses the system call `pread(2)`. On success, returns a non-negative number, as described in "Memory Pressure of a Cuset" on page 16. 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 cuset to enable it.

For more information, see "Memory Pressure of a Cuset" on page 16.

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

`cpuset_close_memory_pressure`

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

Closes the file descriptor `fd` which was opened using the `cpuset_open_memory_pressure` function.

If `fd` is not a valid open file descriptor, 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 cuset to enable it.

For more information, see "Memory Pressure of a Cpuset" on page 16.

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

cpuset_c_rel_to_sys_cpu

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

Returns 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))]`.

cpuset_c_sys_to_rel_cpu

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

Returns 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.

cpuset_c_rel_to_sys_mem

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

Returns 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))]`. Note that this is a left closed, right open interval. The set of points in the interval `[a, b)` is the set of all points `x` such that `a <= x < b`.

cpuset_c_sys_to_rel_mem

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

Returns 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.

cpuset_p_rel_to_sys_cpu

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

Returns 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 does not encompass that relative cpu number.

cpuset_p_sys_to_rel_cpu

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

Returns 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 does not encompass that system-wide cpu number.

cpuset_p_rel_to_sys_mem

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

Returns 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 does not encompass that relative memory node number.

cpuset_p_sys_to_rel_mem

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

Returns 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 does not encompass that system-wide memory node.

cpuset_get_placement

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

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 using `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`.

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

cpuset_equal_placement

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

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 the `cpuset_free_placement` function.

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.

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

cpuset_free_placement

```
cpuset_free_placement(plc) - [optional] Free placement
```

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

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

cpuset_cpupbind

```
int cpuset_cpupbind(int cpu);
```

Binds 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. If in a cpuset of N CPUs,
 * valid values for n are 0 .. N-1.
 */
cpuset_cpupbind(cpuset_p_rel_to_sys_cpu(0, n));
```

`cpuset_latestcpu`

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

Returns 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, 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 it is the 3rd to the last field.

`cpuset_membind`

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

Binds 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.  If 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));
```

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:

- First, kills all the PIDs, looping until there are no more PIDs in this cpuset or below or until the `seconds` time-out limit is exceeded.
- Second, remove that cpuset, and any child cpusets it has, starting from the lowest level leaf node cpusets and working back upwards.
- Third, 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 it sleeps two seconds, increasing one second each loop up to a maximum of ten seconds. If more loops past ten seconds 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 time-out 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, this routine will return quickly, having not slept at all. Otherwise, this routine will at a minimum send a `SIGKILL` signal 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 time out 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.

cpuset_export

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

Writes 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, 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.

For details of the format required for exported cpuset file, see "Cpuset Text Format" on page 35.

cpuset_import

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

Reads 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, 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, zero is written to `*errlinenum_ptr`.

If parsing errors are encountered reading the file and if `errmsg_bufptr` is not `NULL`, it is presumed to point to a character buffer of at least `errmsg_buflen` characters and a null-terminated error message is written to `*errmsg_bufptr`, providing a human readable error message explaining the error message in more detail. Currently, the possible error messages are, as follows:

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

For details of the format required for imported cpuset file, see "Cpuset Text Format" on page 35.

`cpuset_function`

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

Returns 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 runtime environments that may or may not support that function.

`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 behavior. Other mechanisms 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.

Functions to Traverse a Cpuset Hierarchy

The functions described in this section are used to transverse a cpuset hierarchy.

`cpuset_fts_open`

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

Opens 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 information field.

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

`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.

`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.

`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.

`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 null-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.

`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 information 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.
- The information 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.

`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_CPUSSET` for the information field of a `cs_entry`. If the info field contained `CPUSET_FTS_ERR_CPUSSET`, then `cpuset_fts_get_cpuset` returns a pointer to a `struct cpuset` that is all zeros. If the information 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.

cpuset_fts_get_errno

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

Returns the error field of a `cs_entry` obtained from a `cpuset_fts_read()` call.

If this information field has one of the following `CPUSET_FTS_ERR_*` values, it indicates which operation failed, the error 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 information field, then the error field will have the value 0, and the other fields will be contain valid information about that cpuset.

Information field values are, as follows:

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

The above information 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.

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.

Index

C

Cpuset Facility

- advantages, 1
- boot cpuset, 30
 - creating, 30
 - /etc/bootcpuset.conf file, 31
- command line utility, 25
- cpuset
 - definition, 2
 - determine if cpusets are installed, 5
 - errno settings, 41
 - modifying CPUs and kernel processing, 36
 - system error messages, 41
- cpuset permissions, 21
- cpuset text format, 35
- directories, 10
- overview, 1
- programming model, 40
- scheduling and memory allocation, 21
- systems calls
 - mbind, 2
 - sched_setaffinity, 2
 - set_mempolicy, 2
- using cpusets at shell prompt, 23
 - create a cpuset, 23
 - remove a cpuset, 25

Cpuset Facility on SGI Performance Suite

Cpuset library functions

- cpuset_alloc, 54
- cpuset_c_rel_to_sys_cpu, 70
- cpuset_c_rel_to_sys_mem, 70
- cpuset_c_sys_to_rel_cpu, 70
- cpuset_c_sys_to_rel_mem, 70
- cpuset_close_memory_pressure, 69
- cpuset_collides_exclusive, 64
- cpuset_cpu2node, 60

- cpuset_cpupbind, 73
- cpuset_cpumemdist, 60
- cpuset_cpus_nbits, 55
- cpuset_cpus_weight, 58
- cpuset_cpusetofpid, 63
- cpuset_create, 61
- cpuset_delete, 61
- cpuset_equal_placement, 72
- cpuset_export, 75
- cpuset_free, 55
- cpuset_free_pidlist, 66
- cpuset_free_placement, 72
- cpuset_function, 76
- cpuset_get_iopt, 58
- cpuset_get_pidlist, 65
- cpuset_get_placement, 72
- cpuset_get_sopt, 59
- cpuset_getcpu, 57
- cpuset_getcpusetpath, 62
- cpuset_getmems, 58
- cpuset_import, 75
- cpuset_init_pidlist, 65
- cpuset_latestcpu, 73
- cpuset_localcpu, 60
- cpuset_localmems, 60
- cpuset_membind, 73
- cpuset_mems_nbits, 55
- cpuset_mems_weight, 58
- cpuset_migrate, 67
- cpuset_migrate_all, 68
- cpuset_modify, 62
- cpuset_mountpoint, 64
- cpuset_move, 66
- cpuset_move_all, 66
- cpuset_open_memory_pressure, 69
- cpuset_p_rel_to_sys_cpu, 71
- cpuset_p_rel_to_sys_mem, 71

cpuset_p_sys_to_rel_cpu, 71
cpuset_p_sys_to_rel_mem, 71
cpuset_pidlist_length, 65
cpuset_pin, 47
cpuset_query, 61
cpuset_read_memory_pressure, 69
cpuset_reattach, 68
cpuset_set_iopt, 56
cpuset_set_sopt, 57
cpuset_setcpus, 55
cpuset_setmems, 56
cpuset_size, 48
cpuset_unpin, 48
cpuset_where, 48

Creating a cpuset, 23

N

NUMA Tools
Command
dlook, 43

R

Removing a cpuset, 25