Documentation

www.novell.com

How to Shield Your Linux Resources

September 2010

Novell®

Table of Contents

Introduction

In the Linux kernel, the cpuset facility provides a mechanism for creating logical entities called "cpusets" that encompass definitions of CPUs and NUMA Memory Nodes (if NUMA is available). Cpusets constrain the CPU and Memory placement of a task to only the resources defined within that cpuset. These cpusets can then be arranged into a nested hierarchy visible in the "cpuset" virtual file system. Sets of tasks can be assigned to these cpusets to constrain the resources that they use. The tasks can be moved from one cpuset to another to utilize other resources defined in those other cpusets.

The cset command is a Python application that provides a command line front end for the Linux cpusets functionality. Working with cpusets directly can be confusing and slightly complex. The cset tool hides that complexity behind an easy-to-use command line interface.

There are two distinct use cases for cset: the basic shielding use case and the "advanced" case of using raw `set` and `proc` subcommands. The basic shielding function is accessed with the `shield` subcommand and described in the next section. Using the raw `set` and `proc` subcommands allows one to set up arbitrarily complex cpusets and is described in the later sections.

Note that in general, one either uses the `shield` subcommand *or* a combination of the `set` and `proc` subcommands. One rarely, if ever, uses all of these subcommands together. Doing so will likely become too confusing. Additionally, the `shield` subcommand sets up its required cpusets with exclusively marked CPUs. This can interfere with your cpuset strategy. If you find that you need more functionality for your strategy than `shield` provides, go ahead and transition to using `set` and `proc` exclusively. It is straightforward to implement what `shield` does with a few extra `set` and `proc` subcommands.

Obtaining Online Help

For a full list of cset subcommands

```
# cset help
```

For in-depth help on individual subcommands

```
# cset help <subcommand>
```

For options of individual subcommands

```
# cset <subcommand> (-h | --help)
```

The Basic Shielding Model

Although any set up of cpusets can really be described as "shielding," there is one prevalent shielding model in use that is so common that cset has a subcommand that is dedicated to its use. This subcommand is called shield.

The concept behind this model is the use of three cpusets. The *root* cpuset which is always present in all configurations and contains all CPUs. The *system* cpuset which contains CPUs which are used for system tasks. These are the normal tasks that are not "important," but which need to run on the system. And finally, the *user* cpuset which contains CPUs which are used for "important" tasks. The *user* cpuset is the shield. Only those tasks that are somehow important, usually tasks whose performance determines the overall rating for the machine, are run in the *user* cpuset.

The shield subcommand manages all of these cpusets and lets you define the CPUs and Memory Nodes that are in the *shielded* and *unshielded* sets. The subcommand automatically moves all movable tasks on the system into the *unshielded* cpuset on shield activation, and back into the *root* cpuset on shield tear down. The subcommand then lets you move tasks into and out of the shield. Additionally, you can move special tasks (kernel threads) which normally run in the *root* cpuset into the *unshielded* set so that your shield will have even less disturbance.

The shield subcommand abstracts the management of these cpusets away from you and provides options that drive how the shield is set up, which tasks are to be shielded and which tasks are not, and status of the shield. In fact, you need not be bothered with the naming of the required cpusets or even where the cpuset files ystem is mounted. Cset and the shield subcommand takes care of all that.

If you find yourself needing to define more cpusets for your application, then it is likely that this simple shielding is not a rich enough model for you. In this case, you should transition to using the set and proc subcommands described in a later section.

A Simple Shielding Example

Assume that we have a 4-way machine that is not NUMA. This means there are 4 CPUs at our disposal and there is only one Memory Node available. On such machines, we do not need to specify any memory node parameters to cset, it sets up the only available memory node by default.

Usually, one wants to dedicate as many CPUs to the shield as possible and leave a minimal set of CPUs for normal system processing. The reasoning for this is because the performance of the important tasks will rule the performance of the installation as a whole and these important tasks need as many resources available to them as possible, exclusive of other, unimportant tasks that are running on the system.

Note: I use the word "task" to represent either a process or a thread that is running on the system.

Setup and Teardown of the Shield. To set up a shield of 3 CPUs with 1 CPU left for low priority system processing, issue the following command.

```
[zuul:cpuset-trunk]# cset shield -c 1-3
cset: --> activating shielding:
cset: moving 176 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0) with 176 tasks running
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
```

This command does a number of things. First, a *user* cpuset is created with what’s called a CPUSPEC (CPU specification) from the `-c/--cpu` option. This CPUSPEC specifies to use CPUs 1 through 3 inclusively. Next, the command creates a *system* cpuset with a CPUSPEC that is the inverse of the `-c` option for the current machine. On this machine that cpuset will only contain the first CPU, CPU0. Next, all userspace processes running in the *root* cpuset are transfered to the *system* cpuset. This makes all those processes run only on CPU0. The effect of this is that the shield consists of CPUs 1 through 3 and they are now idling.

Note that the command did not move the kernel threads that are running in the *root* cpuset to the *system* cpuset. This is because you may want these kernel threads to use all available CPUs. If you do not, then you can use the -k/--kthread option as described below.

The shield setup command above outputs the information of which cpusets were created and how many tasks are running on each. If you want to see the current status of the shield again, issue this command:

```
[zuul:cpuset-trunk]# cset shield
cset: --> shielding system active with
cset: "system" cpuset of CPUSPEC(0) with 176 tasks running
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
```

Which shows us that the shield is set up and that 176 tasks are running in the *system* cpuset—the "unshielded" cpuset.

It is important to move all possible tasks from the *root* cpuset to the unshielded *system* cpuset because a task's cpuset property is inherited by its children. Since we've moved all running tasks (including init) to the unshielded *system* cpuset, that means that any new tasks that are spawned will also run in the unshielded *system* cpuset.

Some kernel threads can be moved into the unshielded *system* cpuset as well. These are the threads that are not bound to specific CPUs. If a kernel thread is bound to a specific CPU, then it is generally not a good idea to move that thread to the *system* set because at worst it may hang the system and at best it will slow the system down significantly. These threads are usually the IRQ threads on a real time Linux kernel, for example, and you may want to not move these kernel threads into *system*. If you leave them in the *root* cpuset, then they will have access to all CPUs.

However, if your application demands an even "quieter" shield, then you can move all movable kernel threads into the unshielded *system* set with the following command.

```
[zuul:cpuset-trunk]# cset shield -k on
cset: --> activating kthread shielding
cset: kthread shield activated, moving 70 tasks into system cpuset...
[==================================================]%
cset: done
```

You can see that this moved an additional 70 tasks to the unshielded *system* cpuset. Note that the -k/--kthread on parameter can be given at the shield creation time as well and you do not need to perform these two steps separately if you know that you will want kernel thread shielding as well. Executing cset shield again shows us the current state of the shield.

```
[zuul:cpuset-trunk]# cset shield
cset: --> shielding system active with
cset: "system" cpuset of CPUSPEC(0) with 246 tasks running
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
```

You can get a detailed listing of what is running in the shield by specifying either -s/--shield or -u/--unshield to the shield subcommand and using the verbose flag. You will get output similar to the following.

```
[zuul:cpuset-trunk]# cset shield --unshield -v
cset: "system" cpuset of CPUSPEC(0) with 251 tasks running
   USER       PID  PPID SPPr TASK NAME
   -------- ----- ----- ---- ---------
   root        1     0 Soth init [5]
   root        2     0 Soth [kthreadd]
   root       84     2 Sf50 [IRQ-9
]...
   alext    31796 31789 Soth less
   root     32653 25222 Roth python ./cset shield --unshield -v
```

Note that I abbreviated the listing; we do have 251 tasks running in the *system* set. The output is self-explanatory; however, the "SPPr" field may need a little explanation. "SPPr" stands for State, Policy and Priority. You can see that the initial two tasks are Stopped and running in timeshare priority, marked as "oth" (for "other"). The [IRQ-9] task is also stopped, but marked at real time FIFO policy with a priority of 50. The last task in the listing is the cset command itself and is marked as running. Also note that adding a second `-v/--verbose` option will not restrict the output to fit into an 80 character screen.

Tear down of the shield, stopping the shield in other words, is done with the `-r/--reset` option to the `shield` subcommand. When this command is issued, both the *system* and *user* cpusets are deleted and any tasks that are running in both of those cpusets are moved to the *root* cpuset. Once so moved, all tasks will have access to all resources on the system. For example:

```
[zuul:cpuset-trunk]# cset shield --reset
cset: --> deactivating/reseting shielding
cset: moving 0 tasks from "/user" user set to root set...
cset: moving 250 tasks from "/system" system set to root set...
[==================================================]%
cset: deleting "/user" and "/system" sets
cset: done
```

Moving Interesting Tasks Into and Out of the Shield. Now that we have a shield running, the objective is to run our "important" processes in that shield. These processes can be anything, but usually they are directly related to the purpose of the machine. There are two ways to run tasks in the shield:

- Exec a process into the shield
- Move an already running task into the shield

Execing a Process into the Shield

Running a new process in the shield can be done with the -e/--exec option to the shield subcommand. This is the simplest way to get a task to run in the shield. For this example, let’s exec a new bash shell into the shield with the following commands.

```
[zuul:cpuset-trunk]# cset shield -s
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
cset: done

[zuul:cpuset-trunk]# cset shield -e bash
cset: --> last message, executed args into cpuset "/user", new pid is: 13300

[zuul:cpuset-trunk]# cset shield -s -v
cset: "user" cpuset of CPUSPEC(1-3) with 2 tasks running

   USER       PID  PPID SPPr TASK NAME
   -------- ----- ----- ---- ---------
   root     13300  8583 Soth bash
   root     13329 13300 Roth python ./cset shield -s -v

[zuul:cpuset-trunk]# exit

[zuul:cpuset-trunk]# cset shield -s
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
cset: done
```

The first command above lists the status of the shield. We see that the shield is defined as CPUs 1 through 3 inclusive and currently there are no tasks running in it.

The second command execs the bash shell into the shield with the -e option. The last message of cset lists the PID of the new process.

NOTE: cset follows the tradition of separating the tool options from the command to be execed options with a double dash (--). This is not shown in this simple example, but if the command you want to exec also takes options, separate them with the double dash like so: # cset shield -e mycommand -- -v The -v will be passed to mycommand, and not to cset.

The next command lists the status of the shield again. You will note that there are actually two tasks running shielded: our new shell and the cset status command itself. Remember that the cpuset property of a task is inherited by its children. Since we ran the new shell in the shield, its child, which is the status command, also ran in the shield.

TIP: Execing a shell into the shield is a useful way to experiment with running tasks in the shield since all children of the shell will also run in the shield.

The last command exits the shell after which we request a shield status again and see that once again, it does not contain any tasks.

You may have noticed in the output above that both the new shell and the status command are running as the root user. This is because cset needs to run as root and so all it's children will also run as root. If you need to run a process under a different user and or group, you may use the --user and --group options for exec as follows.

```
[zuul:cpuset-trunk]# cset shield --user=alext --group=users -e bash
cset: --> last message, executed args into cpuset "/user", new pid is: 14212

alext@zuul> cset shield -s -v
cset: "user" cpuset of CPUSPEC(1-3) with 2 tasks running
   USER       PID  PPID SPPr TASK NAME
   -------- ----- ----- ---- ---------
   alext    14212  8583 Soth bash
   alext    14241 14212 Roth python ./cset shield -s -v
```

Moving a Running Task into and out of the Shield. While execing a process into the shield is undoubtably useful, most of the time, you'll want to move already running tasks into and out of the shield. The cset shield subcommand includes two options for doing this: -s/--shield and -u/--unshield. These options require what's called a PIDSPEC (process specification) to also be specified with the -p/--pid option. The PIDSPEC defines which tasks get operated on. The PIDSPEC can be a single process ID, a list of process IDs separated by commas, and a list of process ID ranges separated by dashes, groups of which are separated by commas. For example:

--shield --pid 1234

This PIDSPEC argument specifies that PID 1234 be shielded.

--shield --pid 1234,42,1934,15000,15001,15002

This PIDSPEC argument specifies that this list of PIDs only be moved into the shield.

--unshield -p 5000,5100,6010-7000,9232

This PIDSPEC argument specifies that PIDs 5000,5100 and 9232 be unshielded (moved out of the shield) along with any existing PID that is in the range 6010 through 7000 inclusive.

Note: A range in a PIDSPEC does not have to have tasks running for every number in that range. In fact, it is not even an error if there are no tasks running in that range; none will be moved in that case. The range simply specifies to act on any tasks that have a PID or TID that is within that range.

Use of the appropriate PIDSPEC can thus be handy to move tasks and groups of tasks into and out of the shield. Additionally, there is one more option that can help with multi-threaded processes, and that is the --threads flag. If this flag is present in a shield or unshield command with a PIDSPEC and if any of the task IDs in the PIDSPEC belong to a thread in a process container, then all the sibling threads in that process container will get shielded or unshielded as well. This flag provides an easy mechanism to shield/unshield all threads of a process by simply specifying one thread in that process.

In the following example, we move the current shell into the shield with a range PIDSPEC and back out with the bash variable for the current PID.

```
[zuul:cpuset-trunk]# echo $$
22018

[zuul:cpuset-trunk]# cset shield -s -p 22010-22020
cset: --> shielding following pidspec: 22010-22020
cset: done

[zuul:cpuset-trunk]# cset shield -s -v
cset: "user" cpuset of CPUSPEC(1-3) with 2 tasks running
   USER       PID  PPID SPPr TASK NAME
   -------- ----- ----- ---- ---------
   root      3770 22018 Roth python ./cset shield -s -v
   root     22018  5034 Soth bash
cset: done

[zuul:cpuset-trunk]# cset shield -u -p $$
cset: --> unshielding following pidspec: 22018
cset: done

[zuul:cpuset-trunk]# cset shield -s
cset: "user" cpuset of CPUSPEC(1-3) with 0 tasks running
cset: done
```

NOTE: Ordinarily, the shield option will shield a PIDSPEC only if it is currently running in the system set—the unshielded set. The unshield option will unshield a PIDSPEC only if it is currently running in the user set—the shielded set. If you want to shield/unshield a process that happens to be running in the root set (not common), then use the --force option for these commands.

Full Featured Cpuset Manipulation Commands

While basic shielding as described above is useful and a common use model for cset, there comes a time when more functionality will be desired to implement your strategy. To implement this, cset provides two subcommands: set, which allows you to manipulate cpusets; and proc, which allows you to manipulate processes within those cpusets.

The Set Subcommand

In order to do anything with cpusets, you must be able to create, adjust, rename, move and destroy them. The set subcommand allows the management of cpusets in such a manner.

Creating and Destroying Cpusets with Set. The basic syntax of set for cpuset creation is:

```
[zuul:cpuset-trunk]# cset set -c 1-3 -s my_cpuset1
cset: --> created cpuset "my_cpuset1"
```

This creates a cpuset named "my_cpuset1" with a CPUSPEC of CPU1, CPU2 and CPU3. The CPUSPEC is the same concept as described in the "Setup and Teardown of the Shield" section above. The set subcommand also takes a -m/--mem option that lets you specify the memory nodes the set will use as well as flags to make the CPUs and MEMs exclusive to the cpuset. If you are on a non-NUMA machine, just leave the -m option out and the default memory node 0 will be used.

Just like with shield, you can adjust the CPUs and MEMs with subsequent calls to set. If, for example, you wish to adjust the "my_cpuset1" cpuset to only use CPUs 1 and 3 (and omit CPU2), then issue the following command.

```
[zuul:cpuset-trunk]# cset set -c 1,3 -s my_cpuset1
cset: --> modified cpuset "my_cpuset
```

cset will then adjust the CPUs that are assigned to the "my_cpuset1" set to only use CPU1 and CPU3.

To rename a cpuset, use the -n/--newname option. For example:

```
[zuul:cpuset-trunk]# cset set -s my_cpuset1 -n super_set
cset: --> renaming "/cpusets/my_cpuset1" to "super_set"
```

Renames the cpuset called "my_cpuset1" to "super_set".

To destroy a cpuset, use the -d/--destroy option as follows.

```
[zuul:cpuset-trunk]# cset set -d super_set
cset: --> processing cpuset "super_set", moving 0 tasks to parent "/"...
cset: --> deleting cpuset "/super_set"
cset: done
```

This command destroys the newly created cpuset called "super_set". When a cpuset is destroyed, all the tasks running in it are moved to the parent cpuset. The root cpuset, which always exists and always contains all CPUs, cannot be destroyed. You may also give the --destroy option a list of cpusets to destroy.

NOTE: The cset subcommand creates the cpusets based on a mounted cpuset file system. You do not need to know where that file system is mounted, although it is easy to figure out (by default it's on /cpusets). When you give the set subcommand a name for a new cpuset, it is created wherever the cpuset file system is mounted.

If you want to create a cpuset hierarchy, then you must give a path to the cset set subcommand. This path will always begin with the root cpuset, for which the path is /. For example.

```
[zuul:cpuset-trunk]# cset set -c 1,3 -s top_set
cset: --> created cpuset "top_set"

[zuul:cpuset-trunk]# cset set -c 3 -s /top_set/sub_set
cset: --> created cpuset "/top_set/sub_set"
```

These commands created two cpusets: top_set and sub_set. The top_set uses CPU1 and CPU3. It has a subset of sub_set which only uses CPU3. Once you have created a subset with a path, then if the name is unique, you do not have to specify the path in order to affect it. If the name is not unique, then cset will complain and ask you to use the path. For example:

```
[zuul:cpuset-trunk]# cset set -c 1,3 -s sub_set
cset: --> modified cpuset "sub_set
```

This command adds CPU1 to the sub_set cpuset for its use. Note that using the path in this case is optional.

If you attempt to destroy a cpuset which has sub-cpusets, cset will complain and not do it unless you use the -r/--recurse and the --force options. If you do use --force, then all the tasks running in all subsets of the deletion target cpuset will be moved to the target's parent cpuset and all cpusets.

Moving a cpuset from under a certain cpuset to a different location is currently not implemented and is slated for a later release of cset.

Listing Cpusets with Set. To list cpusets, use the set subcommand with the -l/--list option. For example:

```
[zuul:cpuset-trunk]# cset set -l
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y   320    1 /
         one          3 n       0 n     0    1 /one
```

This shows that there is currently one cpuset present called one. (Of course there is also the root set, which is always present.) The output shows that the one cpuset has no tasks running in it. The root cpuset has 320 tasks running. The "-X" for "CPUs" and "MEMs" fields denotes whether the CPUs and MEMs in the cpusets are marked exclusive to those cpusets. Note that the one cpuset has subsets as indicated by a 1 in the Subs field. You can specify a cpuset to list with the set subcommand as follows.

```
[zuul:cpuset-trunk]# cset set -l -s one
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         one          3 n       0 n     0    1 /one
         two          3 n       0 n     0    1 /one/two
```

This output shows that there is a cpuset called two in cpuset one and it also has subset. You can also ask for a recursive listing as follows.

```
[zuul:cpuset-trunk]# cset set -l -r
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y   320    1 /
         one          3 n       0 n     0    1 /one
         two          3 n       0 n     0    1 /one/two
       three          3 n       0 n     0    0 /one/two/three
```

This command lists all cpusets existing on the system since it asks for a recursive listing beginning at the root cpuset. Incidentally, should you need to specify the root cpuset you can use either root or / to specify it explicitely—just remember that the root cpuset cannot be deleted or modified.

The Proc Subcommand

Now that you know how to create, rename and destroy cpusets with the set subcommand, the next step is to manage threads and processes in those cpusets. The subcommand to do this is called proc and it allows you to exec processes into a cpuset, move existing tasks around existing cpusets, and list tasks running in specified cpusets. For the following examples, let us assume a cpuset setup of two sets as follows:

```
[zuul:cpuset-trunk]# cset set -l
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y   309    2 /
         two          2 n       0 n     3    0 /two
       three          3 n       0 n    10    0 /three
```

Listing Tasks with Proc. Operation of the proc subcommand follows the same model as the set subcommand. For example, to list tasks in a cpuset, you need to use the -l/--list option and specify the cpuset by name or, if the name exists multiple times in the cpuset hierarchy, by path. For example:

```
[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 3 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root     16141  4300 Soth bash
 root     16171 16141 Soth bash
 root     16703 16171 Roth python ./cset proc -l two
```

This output shows us that the cpuset called two has CPU2 only attached to it and is running three tasks: two shells and the python command to list it. Note that cpusets are inherited so that if a process is contained in a cpuset, then any children it spawns also run within that set. In this case, the python command to list set two was run from a shell already running in set two. This can be seen by the PPID (parent process ID) of the python command matching the PID of the shell.

Additionally, the "SPPr" field needs explanation. "SPPr" stands for State, Policy and Priority. You can see that the initial two tasks are stopped and running in timeshare priority, marked as "oth" (for "other"). The last task is marked as running, "R" and also at timeshare priority, "oth." If any of these tasks would have been at real time priority, then the policy would be shown as "f" for FIFO or "r" for round robin, and the priority would be a number from 1 to 99. See below for an example.

```
[zuul:cpuset-trunk]# cset proc -l -s root | head -7
cset: "root" cpuset of CPUSPEC(0-3) with 309 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root         1     0 Soth init [5]
 root         2     0 Soth [kthreadd]
 root         3     2 Sf99 [migration/0]
 root         4     2 Sf99 [posix_cpu_timer]
```

This output shows the first few tasks in the root cpuset. Note that both init and [kthread] are running at timeshare; however, the [migration/0] and [posix_cpu_timer] kernel threads are running at real-time policy of FIFO and priority of 99. Incidentally, this output is from a system running the real-time Linux kernel which runs some kernel threads at real-time priorities. And finally, note that you can use cset as any other Linux tool and include it in pipelines as in the example above.

Taking a peek into the third cpuset called three, we see:

```
[zuul:cpuset-trunk]# cset proc -l -s three
cset: "three" cpuset of CPUSPEC(3) with 10 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 alext    16165     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16169     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16170     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16237     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16491     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16492     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16493     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17243     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17244     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17265     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe –bg -...
```

This output shows that a lot of beagled tasks are running in this cpuset and it also shows an ellipsis (…) at the end of their listings. If you see this ellipsis, that means that the command was too long to fit onto an 80 character screen. To see the entire commandline, use the -v/--verbose flag, as per following.

```
[zuul:cpuset-trunk]# cset proc -l -s three -v | head -4
cset: "three" cpuset of CPUSPEC(3) with 10 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 alext    16165     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg --autostarted --indexing-delay 300
```

Execing Tasks with Proc. To exec a task into a cpuset, the proc subcommand needs to be employed with the -e/--exec option. Let’s exec a shell into the cpuset named two in our set. First we check to see what is running that set:

```
[zuul:cpuset-trunk]# cset proc -l -s two
```

```
cset: "two" cpuset of CPUSPEC(2) with 0 tasks running

[zuul:cpuset-trunk]# cset proc -s two -e bash
cset: --> last message, executed args into cpuset "/two", new pid is: 20955

[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 2 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root     20955 19253 Soth bash
 root     20981 20955 Roth python ./cset proc -l two
```

You can see that initially, two had nothing running in it. After the completion of the second command, we list two again and see that there are two tasks running: the shell which we execed and the python cset command that is listing the cpuset. The reason for the second task is that the cpuset property of a running task is inherited by all its children. Since we executed the listing command from the new shell which was bound to cpuset two, the resulting process for the listing is also bound to cpuset two. Let's test that by just running a new shell with no prefixed cset command.

```
[zuul:cpuset-trunk]# bash

[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 3 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root     20955 19253 Soth bash
 root     21118 20955 Soth bash
 root     21147 21118 Roth python ./cset proc -l two
```

Here again we see that the second shell, PID 21118, has a parent PID of 20955 which is the first shell. Both shells, as well as the listing command, are running in the two cpuset.

NOTE: cset follows the tradition of separating the tool options from the command to be execed options with a double dash (--). This is not shown in this simple example, but if the command you want to exec also takes options, separate them with the double dash like so: # cset proc -s myset -e mycommand -- -v The -v will be passed to mycommand, and not to cset.

TIP: Execing a shell into a cpuset is a useful way to experiment with running tasks in that cpuset since all children of the shell will also run in the same cpuset. Finally, if you misspell the command to be execed, the result may be puzzling. For example:

```
[zuul:cpuset-trunk]# cset proc -s two -e blah-blah
cset: --> last message, executed args into cpuset "/two", new pid is: 21655
cset: **> [Errno 2] No such file or directory
```

The result is no new process even though a new PID is output. The reason for the message is of course that the cset process forked in preparation for exec, but the command blah-blah was not found in order to exec it.

Moving Tasks with Proc. Although the ability to exec a task into a cpuset is fundamental, you will most likely be moving tasks between cpusets more often. Moving tasks is accomplished with the -m/--move and -p/--pid options to the proc subcommand of cset. The move option tells the proc subcommand that a task move is requested. The -p/--pid option takes an argument called a PIDSPEC (PID Specification). The PIDSPEC defines which tasks get operated on.

The PIDSPEC can be a single process ID, a list of process IDs separated by commas, and a list of process ID ranges also separated by commas. For example:

--move --pid 1234

> *This PIDSPEC argument specifies that task 1234 be moved.*

--move --pid 1234,42,1934,15000,15001,15002

> *This PIDSPEC argument specifies that this list of tasks only be moved.*

--move --pid 5000,5100,6010-7000,9232

> *This PIDSPEC argument specifies that tasks 5000, 5100 and 9232 be moved along with any existing task that is in the range 6010 through 7000 inclusive.*

NOTE: A range in a PIDSPEC does not have to have running tasks for every number in that range. In fact, it is not even an error if there are no tasks running in that range; none will be moved in that case. The range simply specifies to act on any tasks that have a PID or TID that is within that range.

In the following example, we move the current shell into the cpuset named two with a range PIDSPEC and back out to the root cpuset with the bash variable for the current PID.

```
[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 0 tasks running

[zuul:cpuset-trunk]# echo $$
19253

[zuul:cpuset-trunk]# cset proc -m -p 19250-19260 -t two
cset: moving following pidspec: 19253
cset: moving 1 userspace tasks to /two
cset: done

[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 2 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root     19253 16447 Roth bash
 root     29456 19253 Roth python ./cset proc -l -s two

[zuul:cpuset-trunk]# cset proc -m -p $$ -t root
cset: moving following pidspec: 19253
cset: moving 1 userspace tasks to /
cset: done

[zuul:cpuset-trunk]# cset proc -l -s two
cset: "two" cpuset of CPUSPEC(2) with 0 tasks running
```

Use of the appropriate PIDSPEC can thus be handy to move tasks and groups of tasks. Additionally, there is one more option that can help with multi-threaded processes, and that is the --threads flag. If this flag is present in a proc move command with a PIDSPEC and if any of the task IDs in the PIDSPEC belongs to a thread in a process container, then all the sibling threads in that process container will also get moved. This flag provides an easy mechanism to move all threads of a process by simply specifying one thread in that process. In the following example, we move all the threads running in cpuset three to cpuset two by using the --threads flag.

```
[zuul:cpuset-trunk]# cset set two three
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         two          2 n       0 n     0    0 /two
       three          3 n       0 n    10    0 /three

[zuul:cpuset-trunk]# cset proc -l -s three
cset: "three" cpuset of CPUSPEC(3) with 10 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 alext    16165     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16169     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16170     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16237     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16491     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16492     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16493     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17243     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17244     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    27133     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...

[zuul:cpuset-trunk]# cset proc -m -p 16165 --threads -t two
cset: moving following pidspec: 16491,16493,16492,16170,16165,16169,27133,17244,17243,16237
cset: moving 10 userspace tasks to /two
[==================================================]%
cset: done

[zuul:cpuset-trunk]# cset set two three
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         two          2 n       0 n    10    0 /two
       three          3 n       0 n     0    0 /three
```

Moving All Tasks from One Cpuset to Another. There is a special case for moving all tasks currently running in one cpuset to another. This can be a common use case, and when you need to do it, specifying a PIDSPEC with -p is not necessary so long as you use the -f/--fromset and the -t/--toset options.

In the following example, we move all 10 beagled threads back to cpuset three with this method.

```
[zuul:cpuset-trunk]# cset proc -l two three
cset: "two" cpuset of CPUSPEC(2) with 10 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 alext    16165     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -…
 alext    16169     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16170     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16237     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16491     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16492     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    16493     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17243     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    17244     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
 alext    27133     1 Soth beagled /usr/lib64/beagle/BeagleDaemon.exe --bg -...
cset: "three" cpuset of CPUSPEC(3) with 0 tasks running

[zuul:cpuset-trunk]# cset proc -m -f two -t three
cset: moving all tasks from two to /three
cset: moving 10 userspace tasks to /three
[==================================================]%
cset: done

[zuul:cpuset-trunk]# cset set two three
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         two          2 n       0 n     0    0 /two
       three          3 n       0 n    10    0 /three
```

Moving Kernel Threads with Proc. Kernel threads are special and cset detects tasks that are kernel threads and will refuse to move them unless you also add a -k/--kthread option to your proc move command. Even if you include -k, cset will still refuse to move the kernel thread if they are bound to specific CPUs. The reason for this is system protection.

A number of kernel threads, especially on the real-time Linux kernel, are bound to specific CPUs and depend on per-CPU kernel variables. If you move these threads to a different CPU than what they are bound to, you risk at best that the system will become horribly slow, and at worst a system hang. If you must move those threads (after all, cset needs to give the knowledgeable user access to the keys), then you also need to use the --force option.

WARNING: Overriding a task move command with --force can have dire consequences for the system. Please be sure of the command before you force it.

In the following example, we move all unbound kernel threads running in the root cpuset to the cpuset named two by using the -k option.

```
[zuul:cpuset-trunk]# cset proc -k -f root -t two
cset: moving all kernel threads from / to /two
cset: moving 70 kernel threads to: /two
cset: --> not moving 76 threads (not unbound, use --force)
[==================================================]%
cset: done
```

You will note that we used the fromset→toset facility of the proc subcommand and we only specified the -k option (not the -m option). This has the effect of moving all kernel threads only.

Note that only 70 actual kernel threads were moved and 76 were not. The reason that 76 kernel threads were not moved was because they are bound to specific CPUs. Now, let’s move those kernel threads back to root.

```
[zuul:cpuset-trunk]# cset proc -k -f two -t root
cset: moving all kernel threads from /two to /
cset: ** no task matched move criteria
cset: **> kernel tasks are bound, use --force if ok

[zuul:cpuset-trunk]# cset set -l -s two
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         two          2 n       0 n    70    0 /two
```

What’s this? Cset refused to move the kernel threads back to root because it says that they are "bound." Let’s check this with the Linux taskset command.

```
[zuul:cpuset-trunk]# cset proc -l -s two | head -5
cset: "two" cpuset of CPUSPEC(2) with 70 tasks running
 USER       PID  PPID SPPr TASK NAME
 -------- ----- ----- ---- ---------
 root         2     0 Soth [kthreadd]
 root        55     2 Soth [khelper]

[zuul:cpuset-trunk]# taskset -p 2
pid 2's current affinity mask: 4

[zuul:cpuset-trunk]# cset set -l -s two
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         two          2 n       0 n    70    0 /two
```

Of course, since the cpuset named two only has CPU2 assigned to it, once we moved the unbound kernel threads from root to two, their affinity masks got automatically changed to only use CPU2. This is evident from the taskset output which is a hex value. To really move these threads back to root, we need to force the move as follows.

```
[zuul:cpuset-trunk]# cset proc -k -f two -t root --force
cset: moving all kernel threads from /two to /
cset: moving 70 kernel threads to: /
[==================================================]%
cset: done
```

Destroying Tasks. There actually is no cset subcommand or option to destroy tasks—it’s not really needed. Tasks exist and are accessible on the system as normal, even if they happen to be running in one cpuset or another. To destroy tasks, use the usual ^C method or by using the kill(1) command.

Implementing "Shielding" with Set and Proc

With the preceding material on the set and proc subcommands, we now have the background to implement the basic shielding model, just like the shield subcommand.

One may pose the question as to why we want to do this, especially since shield already does it? The answer is that sometimes one needs more functionality than shield has to implement one's shielding strategy. In those cases you need to first stop using shield since that subcommand will interfere with the further application of set and proc; however, you will still need to implement the functionality of shield in order to implement successful shielding.

Remember from the above sections describing shield, that shielding has at minimum three cpusets: root, which is always present and contains all CPUs; system which is the "non-shielded" set of CPUs and runs unimportant system tasks; and user, which is the "shielded" set of CPUs and runs your important tasks. Remember also that shield moves all movable tasks into system and, optionally, moves unbound kernel threads into system as well.

We start first by creating the system and user cpusets as follows. We assume that the machine is a four-CPU machine without NUMA memory features. The system cpuset should hold only CPU0 while the user cpuset should hold the rest of the CPUs.

```
[zuul:cpuset-trunk]# cset set -c 0 -s system
cset: --> created cpuset "system"

[zuul:cpuset-trunk]# cset set -c 1-3 -s user
cset: --> created cpuset "user"

[zuul:cpuset-trunk]# cset set -l
cset:
         Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         root        0-3 y       0 y   333    2 /
         user        1-3 n       0 n     0    0 /user
       system          0 n       0 n     0    0 /system
```

Now, we need to move all running user processes into the system cpuset.

```
[zuul:cpuset-trunk]# cset proc -m -f root -t system
cset: moving all tasks from root to /system
cset: moving 188 userspace tasks to /system
[==================================================]%
cset: done

[zuul:cpuset-trunk]# cset set -l
cset:
         Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         root        0-3 y       0 y   146    2 /
         user        1-3 n       0 n     0    0 /user
       system          0 n       0 n   187    0 /system
```

We now have the basic shielding set up. Since all userspace tasks are running in system, anything that is spawned from them will also run in system. The user cpuset has nothing running in it unless you put tasks there with the proc subcommand as described above. If you also want to move movable kernel threads from root to system (in order to achieve a form of "interrupt shielding" on a real time Linux kernel for example), you would execute the following command as well.

```
[zuul:cpuset-trunk]# cset proc -k -f root -t system
cset: moving all kernel threads from / to /system
cset: moving 70 kernel threads to: /system
cset: --> not moving 76 threads (not unbound, use --force)
[==================================================]%
cset: done

[zuul:cpuset-trunk]# cset set -l
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y    76    2 /
        user        1-3 n       0 n     0    0 /user
      system          0 n       0 n   257    0 /system
```

At this point, you have achieved the simple shielding model that the shield subcommand provides. You can now add other cpuset definitions to expand your shielding strategy beyond that simple model.

Implementing Hierarchy with Set and Proc

One popular extended "shielding" model is based on hierarchical cpusets, each with diminishing numbers of CPUs. This model is used to create "priority cpusets" that allow assignment of CPU resources to tasks based on some arbitrary priority definition. The idea is that a higher priority task will get access to more CPU resources than a lower priority task.

The example provided here once again assumes a machine with four CPUs and no NUMA memory features. This base serves to illustrate the point well; however, note that if your machine has (many) more CPUs, then strategies such as this and others get more interesting.

We define a shielding set up as in the previous section where we have a system cpuset with just CPU0 that takes care of "unimportant" system tasks. You will usually require this type of cpuset since it forms the basis of shielding. We modify the strategy to not use a user cpuset; instead we create a number of new cpusets each holding one more CPU than the other. These cpusets will be called prio_low with one CPU, prio_med with two CPUs, prio_high with three CPUs, and prio_all with all CPUs.

NOTE: You may ask, why create a prio_all with all CPUs when that is substantially the definition of the root cpuset? The answer is that it is best to keep a separation between the root cpuset and everything else, even if a particular cpuset duplicates root exactly. Usually, automation is build on top of a cpuset strategy. In these cases, it is best to avoid using invariant names of cpusets, such as root for example, in this automation.

All of these prio_* cpusets can be created under root, in a flat way; however, it is advantageous to create them as a hierarchy. The reasoning for this is twofold: first, if a cpuset is destroyed, all its tasks are moved to its parent; second, one can use exclusive CPUs in a hierarchy.

There is a planned addition to the proc subcommand that will allow moving a specified PIDSPEC of tasks running in a specified cpuset to its parent. This addition will ease the automation burden.

If a cpuset has CPUs that are exclusive to it, then other cpusets may not make use of those CPUs unless they are children of that cpuset. This has more relevance to machines with many CPUs and more complex strategies.

Now, we start with a clean slate and build the appropriate cpusets as follows.

```
[zuul:cpuset-trunk]# cset set -r
cset:
     Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
     root        0-3 y       0 y   344    0 /

[zuul:cpuset-trunk]# cset set -c 0-3 prio_all
cset: --> created cpuset "prio_all"


[zuul:cpuset-trunk]# cset set -c 1-3 /prio_all/prio_high
cset: --> created cpuset "/prio_all/prio_high"

[zuul:cpuset-trunk]# cset set -c 2-3 /prio_all/prio_high/prio_med
cset: --> created cpuset "/prio_all/prio_high/prio_med"

[zuul:cpuset-trunk]# cset set -c 3 /prio_all/prio_high/prio_med/prio_low
cset: --> created cpuset "/prio_all/prio_high/prio_med/prio_low"

[zuul:cpuset-trunk]# cset set -c 0 system
cset: --> created cpuset "system"

[zuul:cpuset-trunk]# cset set -l -r
cset:
     Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
     root        0-3 y       0 y   344    2 /
   system          0 n       0 n     0    0 /system
  prio_all       0-3 n       0 n     0    1 /prio_all
 prio_high       1-3 n       0 n     0    1 /prio_all/prio_high
  prio_med       2-3 n       0 n     0    1 /prio_all/prio_high/prio_med
  prio_low         3 n       0 n     0    0 /prio_all/pr...rio_med/prio_low
```

NOTE: We used the -r/--recurse switch to list all the sets in the last command above. If we had not, then the prio_med and prio_low cpusets would not have been listed.

The strategy is now implemented and we now move all userspace tasks as well as all movable kernel threads into the system cpuset to activate the shield.

```
[zuul:cpuset-trunk]# cset proc -m -k -f root -t system
cset: moving all tasks from root to /system
cset: moving 198 userspace tasks to /system
cset: moving 70 kernel threads to: /system
cset: --> not moving 76 threads (not unbound, use --force)
[==================================================]%
cset: done

[zuul:cpuset-trunk]# cset set -l -r
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y    76    2 /
      system          0 n       0 n   268    0 /system
    prio_all        0-3 n       0 n     0    1 /prio_all
   prio_high        1-3 n       0 n     0    1 /prio_all/prio_high
    prio_med        2-3 n       0 n     0    1 /prio_all/prio_high/prio_med
    prio_low          3 n       0 n     0    0 /prio_all/pr...rio_med/prio_low
```

The shield is now active. Since the prio_* cpuset names are unique, you can assign tasks to them either via their simple name, or their full path (as described in the proc section above).

You may have noted that there is an ellipsis in the path of the prio_low cpuset in the listing above. This is done in order to fit the output onto an 80 character screen. If you want to see the entire line, then you need to use the -v/--verbose flag as follows.

```
[zuul:cpuset-trunk]# cset set -l -r -v
cset:
        Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
        root        0-3 y       0 y    76    2 /
      system          0 n       0 n   268    0 /system
    prio_all        0-3 n       0 n     0    1 /prio_all
   prio_high        1-3 n       0 n     0    1 /prio_all/prio_high
    prio_med        2-3 n       0 n     0    1 /prio_all/prio_high/prio_med
    prio_low          3 n       0 n     0    0 /prio_all/prio_high/prio_med/prio_low
```

Using Shortcuts

The commands listed in the previous sections always used all the required options. However, Cset does have a shortcut facility that will execute certain commands without specifying all options. An effort has been made to do this with the "principle of least surprise." This means that if you do not specify options, but do specify parameters, then the outcome of the command should be intuitive as possible.

Using shortcuts is not necessary. In fact, you can use either shortcuts or long options. However, using long options instead of shortcuts does have a use case: when you write a script intended to be self-documenting, or perhaps when you generate cset documentation.

To begin, the subcommands shield, set and proc can themselves be shortened to the fewest number of characters that are unambiguous. For example, the following commands are identical:

```
# cset shield -s -p 1234        <-->   # cset sh -s -p 1234
# cset set -c 1,3 -s newset     <-->   # cset se -c 1,3 -s newset
# cset proc -s newset -e bash   <-->   # cset p -s newset -e bash
```

Note that proc can be shortened to just p, while shield and set need two letters to disambiguate.

Shield Subcommand Shortcuts

The shield subcommand supports two areas with shortcuts: the case when there are no options given where to shield is the common use case, and making the -p/--pid option optional for the -s/--shield and -u/--unshield options.

For the common use case of actually shielding either a PIDSPEC or execing a command into the shield, the following cset commands are equivalent.

```
# cset shield -s -p 1234,500-649  <-->   # cset sh 1234,500-649
# cset shield -s -e bash          <-->   # cset sh bash
```

When using the -s or -u shield/unshield options, it is optional to use the -p option to specify a PIDSPEC. For example:

```
# cset shield -s -p 1234   <-->   # cset sh -s 1234
# cset shield -u -p 1234   <-->   # cset sh -u 1234
```

Set Subcommand Shortcuts

The set subcommand has a limited number of shortcuts. Basically, the -s/--set option is optional in most cases and the -l/--list option is also optional if you want to list sets. For example, these commands are equivalent.

```
# cset set -l -s myset          <-->  # cset se -l myset
# cset se -l myset              <-->  # cset se myset

# cset set -c 1,2,3 -s newset   <-->  # cset se -c 1,2,3 newset
# cset set -d -s newset         <-->  # cset se -d newset

# cset set -n newname -s oldname <-->  # cset se -n newname oldname
```

In fact, if you want to apply either the list or the destroy options to multiple cpusets with one cset command, you will not need to use the -s option. For example:

```
# cset se -d myset yourset ourset
  --> destroys cpusets: myset, yourset and ourset

# cset se -l prio_high prio_med prio_low
  --> lists only cpusets prio_high, prio_med and prio_low
  --> the -l is optional in this case since list is default
```

Proc Subcommand Shortcuts

For the proc subcommand, the -s, -t and -f options to specify the cpuset, the origination cpuset and the destination cpuset can sometimes be optional. For example, the following commands are equivalent.

To list tasks in cpusets:

```
# cset proc -l -s myset        \
# cset proc -l -f myset         -->  # cset p -l myset
# cset proc -l -t myset        /

# cset p -l myset              <-->  # cset p myset

# cset proc -l -s one two      <-->  # cset p -l one two
# cset p -l one two            <--> # cset p one two
```

To exec a process into a cpuset:

```
# cset proc -s myset -e bash   <-->  # cset p myset -e bash
```

Movement of tasks into and out of cpusets have the following shortcuts.

To move a PIDSPEC into a cpuset:

```
# cset proc -m -p 4242,4243 -s myset <--> # cset p -m 4242,4243 myset
# cset proc -m -p 12 -t myset        <--> # cset p -m 12 myset
```

To move all tasks from one cpuset to another:

```
# cset proc -m -f set1 -t set2      \
# cset proc -m -s set1 -t set2       --> # cset p -m set1 set2
# cset proc -m -f set1 -s set2      /
```

What To Do If There Are Problems

If you encounter problems with the cset application, the best option is to log a bug with the cset bugzilla instance found here:

http://code.google.com/p/cpuset/issues/list

If you are using cset on a supported operating system such as SUSE Linux Enterprise Server or SUSE Linux Enterprise Server Real Time Extension from Novell, then please use that bugzilla instead at:

http://bugzilla.novell.com

If the problem is repeatable, there is an excellent chance that it will get fixed quickly. Also, cset contains a logging facility that is invaluable for the developers to diagnose problems. To create a log of a run, use the -l/--log option with a file name as an argument to the main cset application. For example.

```
# cset -l logfile.txt set -n newname oldname
```

That command saves a lot of debugging information in the logfile.txt file. Please attach this file to the bug.

Name

cset - manage cpusets functions in the Linux kernel

Synopsis

- cset [--version | --help | --tohex]
- cset [help <command> | <command> --help]
- cset [cset options] <command> [command options] [args]

Description

Note: In general, you need to have root permissions to run cset. The tool mounts the cpusets file system and manipulates it. Non-root users do not have permission for these actions.

Cset is a Python application to make using the cpusets facilities in the Linux kernel easier. The actual included command is called cset and it allows manipulation of cpusets on the system. It also provides higher level functions such as implementation and control of a basic CPU shielding setup.

Typical uses of cset include

- *Setting up and managing a simple shielded CPU environment.* The concept of shielded CPUs is that a certain number of CPUs are partitioned off on the system and only processes that are of interest are run on these CPUs (i.e., inside the shield).

  For a simple shielded configuration, one typically uses three cpusets: the root set, a system set and a user set. Cset includes a super command that implements this strategy and lets you easily manage it. See cset-shield(1) for more details.
- *Setting up and managing a complex shielding environment.* Shielding can be more complex when concepts such as priority cpusets and intersecting cpusets are used. You can use cset to help manage this type of shielding as well. You will need to use the cset-set(1) and cset-proc(1) subcommands directly to do that.
- *Managing cpusets on the system.* The cset subcommand cset-set(1) allows you to create and destroy arbitrary cpusets on the system and assign arbitrary CPUs and memory nodes to them. The cpusets so created have to follow the Linux kernel cpuset rules. See the cset-set(1) subcommand for more details.
- *Managing processes that run on various system cpusets.* The cset subcommand cset-proc(1) allows you to manage processes running on various cpusets created on the system. You can execute new processes in specific cpusets and move tasks around existing cpusets. See the cset-proc(1) subcommand for more details.

Options

The following generic option flags are available. Additional options are available per-command, and are documented in the command-specific documentation.

- *cset –version.* Displays version information and exits.
- *cset –help.* Prints the synopsis and a list of all commands.
- *cset --log <filename>.* Creates a log file for the current run. All manner of useful information is stored in this file. This is usually used to debug cset when things don't go as planned.
- *cset –machine.* Makes cset output information for all operations in a format that is machine readable (i.e., easy to parse).
- *cset --tohex <CPUSPEC>.* Converts a CPUSPEC (see cset-set(1) for definition) to a hexadecimal number and outputs it. Useful for setting IRQ stub affinity to a cpuset definition.

Cset Commands

The cset commands are divided into groups, according to the primary purpose of those commands. Following is a short description of each command. A more detailed description is available in individual command manpages. Those manpages are named cset-<command>(1). The first command, help, is especially useful as it prints out a long summary of what a particular command does.

- *cset help command.* Print out a lengthy summary of how the specified subcommand works
- *cset command –help.* Print out an extended synopsis of the specified subcommand
- *cset shield.* Super command to set up and manage basic shielding (see cset-shield(1))
- *cset set.* Create, modify and destroy cpusets (see cset-set(1))
- *cset proc.* Create and manage processes within cpusets (see cset-proc(1))

Persistent CPUsets

To create a persistent cpuset setup, i.e., one that survives a reboot, you need to create the file /etc/init.d/cset. This distribuition of cset includes an example cset init.d file found in /usr/share /doc/pacakges/cpuset which is called cset.init.d. You will need to alter the file to your specifications and copy it to be the file /etc/init.d/cset. See the comments in that file for more details.

Files

If used, the init.d script /etc/init.d/cset starts and stops a cpuset configuration on boot and power off. Cpuset uses a configuration file if present on the system. The file is /etc/cset.conf and may contain the following options.

mountpoint = <directory_name>. Specify the mountpoint where the cpuset file system is to be mounted. By default this is /cpusets; however, some people prefer to mount this in the more traditional /dev/cpusets.

License

Cpuset is licensed under the GNU GPL V2 only.

Author

Written by Alex Tsariounov (alext@novell.com). Some substrate code and ideas were taken from the excellent Stacked GIT (stgit) v0.13 (see http://gna.org/projects/stgit and http://www.procode.org/stgit). Stacked GIT is under GPL V2 or later.

See Also

- cset-set(1), cset-proc(1), cset-shield(1)
- /usr/share/doc/packages/cpuset/html/tutorial.html
- /usr/share/doc/packages/cpuset/cset.init.d
- taskset(1), chrt(1)
- /usr/src/linux/Documentation/cpusets.txt

Name

cset-proc - manage processes running in cpusets

Synopsis

- cset [cset options] proc [proc options] [args]
- cset proc --help
- cset proc
- cset proc my_set my_other_set
- cset proc --list --set my_set
- cset proc --exec my_set /opt/software/my_code --my_opt_1
- cset proc --set my_set --exec /opt/software/my_code --my_opt_1
- cset proc --move 2442,3000-3200 my_set
- cset proc --move --pid=2442,3000-3200 --toset=my_set
- cset proc --move --fromset=my_set_1 --toset=my_set_2
- cset proc --move --pid=42 --fromset=/group1/myset --toset=/group2/yourset

Description

This command is used to run and manage arbitrary processes on specified cpusets. It is also used to move pre-existing processes and threads to specified cpusets. You may note there is no "kill" or "destroy" option — use the standard OS ^C or kill commands for that. To list which tasks are running in a particular cpuset, use the --list command. For example:

```
# cset proc --list --set myset
```

This command will list all the tasks running in the cpuset called "myset". Processes are created by specifying the path to the executable and specifying the cpuset that the process is to be created in. For example:

```
# cset proc --set=blazing_cpuset --exec /usr/bin/fast_code
```

This command will execute the /usr/bin/fast_code program on the "blazing_cpuset" cpuset. Note that if your command takes options, then use the traditional "--" marker to separate cset's options from your command's options. For example:

```
# cset proc --set myset --exec — ls -l
```

This command will execute "ls -l" on the cpuset called "myset". The PIDSPEC argument taken for the move command is a comma separated list of PIDs or TIDs. The list can also include brackets of PIDs or TIDs (i.e., tasks) that are inclusive of the endpoints. For example:

1,2,5 Means processes 1, 2 and 5

1,2,600-700 Means processes 1, 2 and from 600 to 700

NOTE: The range of PIDs or TIDs does not need to have every position populated. In other words, for the example above, if there is only one process, say PID 57, in the range of 50-65, then only that process will be moved.

To move a PIDSPEC to a specific cpuset, you can either specify the PIDSPEC with --pid and the destination cpuset with --toset, or use the short hand and list the cpuset name after the PIDSPEC for the --move arguments. The move command accepts multiple common calling methods. For example, the following commands are equivalent:

```
# cset proc --move 2442,3000-3200 reserved_set
# cset proc --move --pid=2442,3000-3200 --toset=reserved_set
```

These commands move the tasks defined as 2442 and any running task between 3000 and 3200 inclusive of the ends to the cpuset called "reserved_set".

Specifying the --fromset is not necessary since the tasks will be moved to the destination cpuset no matter which cpuset they are currently running on.

NOTE: However, if you do specify a cpuset with the --fromset option, then only those tasks that are both in the PIDSPEC and are running in the cpuset specified by –fromset will be moved. That is, if there is a task running on the system but not in --fromset that is in PIDSPEC, it will not be moved.

If the --threads switch is used, then the proc command will gather any threads of belonging to any processes or threads that are specified in the PIDSPEC and move them. This provides an easy way to move all related threads: just pick one TID from the set and use the –threads option.

To move all userspace tasks from one cpuset to another, you need to specify the source and destination cpuset by name. For example:

```
# cset proc --move --fromset=comp1 --toset=comp42
```

This command specifies that all processes and threads running on cpuset "comp1" be moved to cpuset "comp42".

NOTE: This move command will not move kernel threads unless the -k/--kthread switch is specified. If it is, then all unbound kernel threads will be added to the move. Unbound kernel threads are those that can run on any CPU. If you also specify the –force switch, then all tasks, kernel or not, bound or not, will be moved.

CAUTION: Please be cautious with the --force switch, since moving a kernel thread that is bound to a specific CPU to a cpuset that does not include that CPU can cause a system hang.

You must specify unique cpuset names for both the exec and move commands. If a simple name passed to the --fromset, --toset and --set parameters is unique on the system, then that command executes. However, if there are multiple cpusets by that name, then you will need to specify which one you mean with a full path rooted at the base cpuset tree. For example, suppose you have the following cpuset tree:

```
/cpusets
    /group1
        /myset
        /yourset
    /group2
        /myset
        /yourset
```

Then, to move a process from myset in group1 to yourset in group2, you would have to issue the following command:

```
# cset proc --move --pid=50 --fromset=/group1/myset --toset=/group2/yourset
```

You do not have to worry about where in the Linux file system the cpuset file system is mounted. The cset command takes care of that. Any cpusets that are specified by path (such as above), are done with respect to the root of the cpuset file system.

Options

- *-h, --help.* Prints the list of options for this command
- *-l, --list*. List processes in the specified cpuset
- *-e, --exec*. Execute arguments in the specified cpuset
- *-u USER, --user=USER.* Use this USER to --exec (id or name)
- *-g GROUP, --group=GROUP*. Use this GROUP to --exec (id or name)
- *-m, --move*. Move specified tasks to specified cpuset; to move a PIDSPEC to a cpuset, use –m PIDSPEC cpuset; to move all tasks specify --fromset and --toset
- *-p PIDSPEC, --pid=PIDSPEC*. Specify pid or tid specification
- *--threads*. If specified, any processes found in the PIDSPEC to have multiple threads will automatically have all their threads added to the PIDSPEC (use to move all related threads to a cpuset)
- *-s CPUSET, --set=CPUSET*. Specify name of immediate cpuset
- *-t TOSET, --toset=TOSET*. Specify name of destination cpuset
- *-f FROMSET, --fromset=FROMSET*. Specify name of origination cpuset
- *-k, --kthread.* Move, or include moving, unbound kernel threads
- *--force*. Force all processes and threads to be moved
- *-v, --verbose*. Prints more detailed output, additive

License

Cpuset is licensed under the GNU GPL V2 only.

Author

Written by Alex Tsariounov (alext@novell.com).

See Also

- cset-set(1), cset-proc(1), cset-shield(1)
- /usr/share/doc/packages/cpuset/html/tutorial.html
- taskset(1), chrt(1)
- /usr/src/linux/Documentation/cpusets.txt

Name

cset-shield - cpuset super command which implements CPU shielding

Synopsis

- cset [cset options] shield [shield options] [args]
- cset shield --help
- cset shield
- cset shield --cpu 1-7
- cset shield --cpu 1-7 --kthread=on
- cset shield --exec /opt/software/myapp/doit --my_opt1 --my_opt2
- cset shield --user appuser --exec run_benchmark.sh
- cset shield --shield --pid 1024,2048,5000-1000
- cset shield --unshield --pid 6000-8500
- cset shield --kthread=off
- cset shield --kthread=on
- cset shield --shield bash

Description

This is a super command that creates basic CPU shielding. The normal cset commands can, of course, be used to create this basic shield, but the shield command combines many such commands to create and manage a common type of CPU shielding setup.

The concept of shielding implies at minimum three cpusets, for example: root, user and system. The root cpuset always exists in all implementations of cpusets and contains all available CPUs on the machine. The system cpuset is so named because normal system tasks are made to run on it. The user cpuset is so named because that is the "shielded" cpuset on which you would run your tasks of interest.

Usually, CPU zero would be in the system set and the rest of the CPUs would be in the user set. After creation of the cpusets, all processes running in the root cpuset are moved to the system cpuset. Thus any new processes or threads spawned from these processes will also run the system cpuset.

If the optional --kthread=on option is given to the shield command, then all kernel threads (with the exception of the per-CPU bound interrupt kernel threads) are also moved to the system set.

You can execute processes on the shielded user cpuset with the --exec subcommand or move processes or threads to the shielded cpuset with the --shield subcommand with a --pid option.

NOTE: You do not need to specify which cpuset a process or thread is running in initially when using the --shield subcommand.

To create a shield, you would execute the shield command with the --cpu option that specifies CPUSPEC argument that assigns CPUs to be under the shield (this means assigned to the user cpuset, all other cpus will be assigned to the system set). For example:

```
# cset shield --cpu=1-3
```

On a 4-way machine, this command will dedicate the first processor, CPU0, for the system set (unshielded) and the last three processors, CPU1, CPU2, CPU3, for the user set (shielded).

The CPUSPEC will accept a comma separated list of CPUs and inclusive range specifications. For example, --cpu=1,3,5-7 will assign CPU1, CPU3, CPU5, CPU6, and CPU7 to the user (or shielded) cpuset and the inverse of that to the system (or unshielded) cpuset.

If you do not like the names "system" and "user" for the unshielded and shielded sets respectively, or if those names are used already, then use the --sysset and --userset options. For example:

```
# cset shield --sysset=free --userset=cage --cpu=2,3 --kthread=on
```

The above command will use the name free for the unshielded system cpuset, the name cage for the shielded user cpuset, initialize these cpusets and dedicate CPU0 and CPU1 to the free set and (on a 4-way machine) dedicate CPU2 and CPU3 to the cage set. Further, the command moves all processes and threads, including kernel threads from the root cpuset to the free cpuset.

NOTE: If you do use the --syset/--userset options, then you must continue to use those for every invocation of the shield super command.

After initialization, you can run the process of interest on the shielded cpuset with the --exec subcommand, or move processes or threads already running to the shielded cpuset with the --shield subcommand and the --pid option.

Note that if your execute command takes options, then use the traditional "--" marker to separate cset's options from your command's options. For example:

```
# cset shield --exec — ls -l
```

This command will execute "ls -l" inside the shield.

The PIDSPEC argument taken for the --pid (or -p) option is a comma separated list of PIDs or TIDs. The list can also include brackets of PIDs or TIDs that are inclusive of the endpoints. For example:

1,2,5	*Means processes 1, 2 and 5*
1,2,600-700	*Means processes 1, 2 and from 600 to 700*

```
# cset shield --shield --pid=50-65
```

The above command moves all processes and threads with PID or TID in the range 50-65 inclusive, from the system cpuset into the shielded user cpuset. If they are running in the root cpuset, you must use the --force option to actually move them into the shield.

NOTE: The range of PIDs or TIDs does not need to have every position populated. In other words, for the example above, if there is only one process, say PID 57, in the range of 50-65, then only that process will be moved.

The --unshield (or -u) subcommand will remove the specified processes or threads from the shielded cpuset and move them into the unshielded (or system) cpuset. This command is also used in conjuction with a -p/--pid option that specifies a PIDSPEC argument, the same as for the --shield subcommand.

Both the --shield and the --unshield commands will also finally output the number of tasks running in the shield and out of the shield if you do not specify a PIDSPEC with --pid. By specifying also a --verbose in addition, then you will get a listing of every task that is running either in the shield or out of the shield.

Using no subcommand, ie., only "cset shield", will output the status of both shield and non-shield. Tasks will be listed if --verbose is used.

You can adjust which CPUs are in the shielded cpuset by issuing the --cpu subcommand again anytime after the shield has been initialized.

For example if the original shield contained CPU0 and CPU1 in the system set and CPU2 and CPU3 in the user set, if you then issue the following command:

```
# cset shield --cpu=1,2,3
```

Then that command will move CPU1 into the shielded "user" cpuset. Any processes or threads that were running on CPU1 that belonged to the unshielded "system" cpuset are migrated to CPU0 by the system.

The --reset subcommand will in essence destroy the shield. For example, if there was a shield on a 4-way machine with CPU0 in system and CPUs 1-3 in user with processes running on the user cpuset (i.e., in the shield), and a --reset subcommand was issued, then all processes running in both system and user cpusets would be migrated to the root cpuset (which has access to all CPUs and never goes away), after which both system and user cpusets would be destroyed.

NOTE: Even though you can mix general usage of cpusets with the shielding concepts described here, you generally will not want to. For more complex shielding or usage scenarios, one would usually use the normal cpuset commands (i.e., cset set and proc) directly.

Options

- *-h, --help*. Prints the list of options for this command
- *-c CPUSPEC, --cpu=CPUSPEC*. Modifies or initializes the shield cpusets
- *-r, --reset*. Destroys the shield
- *-e, --exec*. Executes args in the shield
- *--user=USER* . Use this USER for --exec (id or name)
- *--group=GROUP*. Use this GROUP for --exec (id or name)
- *-s, --shield*. Shield PIDSPEC specified with -p/--pid of processes or threads
- *-u, --unshield*. Remove PIDSPEC specified with -p/--pid of processes or threads from the shield, the tasks keep running in the unshielded cpuset
- *--threads*. If specified, any processes found in the PIDSPEC to have multiple threads will automatically have all their threads added to the PIDSPEC (use to shield or unshield all related threads)
- *-k on|off, --kthread=on|off*. Shield from unbound interrupt threads as well
- *-f, --force*. Force operation, use with care
- *-v, --verbose*. Prints more detailed output, additive
- *--sysset=SYSSET*. Optionally specify system cpuset name
- *--userset=USERSET*. Optionally specify user cpuset name

License

Cpuset is licensed under the GNU GPL V2 only.

Author

Written by Alex Tsariounov (alext@novell.com).

See Also

- cset-set(1), cset-proc(1), cset-shield(1)
- /usr/share/doc/packages/cpuset/html/tutorial.html
- taskset(1), chrt(1)
- /usr/src/linux/Documentation/cpusets.txt

Name

cset-set - manage sets of cpus

Synopsis

- cset [cset options] shield [shield options] [args]
- cset shield --help
- cset set
- cset set --recurse
- cset set --list myset
- cset set myset
- cset set --recurse --list myset
- cset set --cpu 2-5 --mem 0 --set newset
- cset set --cpu 2-5 newset
- cset set --cpu 1,2,5-7 another_set
- cset set --destroy newset
- cset set --destroy /mygroup_sets/my_set

Description

This command is used to create, modify, and destroy cpusets. Cpusets form a tree-like structure rooted at the root cpuset which always includes all system CPUs and all system memory nodes.

A cpuset is an organizational unit that defines a group of CPUs and a group of memory nodes where a process or thread (i.e., task) is allowed to run on. For non-NUMA machines, the memory node is always 0 (zero) and cannot be set to anything else. For NUMA machines, the memory node can be set to a similar specification as the CPU definition and will tie those memory nodes to that cpuset. You will usually want the memory nodes that belong to the CPUs defined to be in the same cpuset.

A cpuset can have exclusive rights to the CPUs defined in it. This means that only this cpuset can own these CPUs. Similarly, a cpuset can have exclusive rights to the memory nodes defined in it. This means that only this cpuset can own these memory nodes.

Cpusets can be specified by name or by path; however, care should be taken when specifying by name if the name is not unique. This tool will generally not let you do destructive things to non-unique cpuset names.

Cpusets are uniquely specified by path. The path starts at where the cpusets file system is mounted so you generally do not have to know where that is. For example, specify a cpuset that is called "two", which is a subset of "one", which in turn is a subset of the root cpuset. Use the path "/one/two" regardless of where the cpusets file system is mounted.

When specifying CPUs, a so-called CPUSPEC is used. The CPUSPEC will accept a comma-separated list of CPUs and inclusive range specifications. For example, --cpu=1,3,5-7 will assign CPU1, CPU3, CPU5, CPU6, and CPU7 to the specified cpuset.

Note that cpusets follow certain rules. For example, children can only include CPUs that the parents already have. If you do not follow those rules, the kernel cpuset subsystem will not let you create that cpuset. For example, if you create a cpuset that contains CPU3, and then attempt to create a child of that cpuset with a CPU other than 3, you will get an error, and the cpuset will not be active. The error is somewhat cryptic in that it is usually a "Permission denied" error.

Memory nodes are specified with a MEMSPEC in a similar way to the CPUSPEC. For example, --mem=1,3-6 will assign MEM1, MEM3, MEM4, MEM5, and MEM6 to the specified cpuset.

Note that if you attempt to create or modify a cpuset with a memory node specification that is not valid, you may get a cryptic error message, "No space left on device", and the modification will not be allowed.

When you destroy a cpuset, then the tasks running in that set are moved to the parent of that cpuset. If this is not what you want, then manually move those tasks to the cpuset of your choice with the cset proc command (see cset proc --help for more information).

Examples

Create a cpuset with the default memory specification

```
# cset set --cpu=2,4,6-8 --set=new_set
```

This command creates a cpuset called "new_set" located off the root cpuset which holds CPUS 2,4,6,7,8 and node 0 (interleaved) memory. Note that --set is optional, and you can just specify the name for the new cpuset after all arguments.

Create a cpuset that specifies both CPUs and memory nodes

```
# cset set --cpu=3 --mem=3 /rad/set_one
```

Note that this command uses the full path method to specify the name of the new cpuset "/rad/set_one". It also names the new cpuset implicitly (i.e., no --set option, although you can use that if you want to). If the "set_one" name is unique, you can subsequently refer to it just by that. Memory node 3 is assigned to this cpuset as well as CPU 3.

The above commands will create the new cpusets, or if they already exist, they will modify them to the new specifications.

Options

- *-h, --help*. Prints the list of options for this command
- -l, --list. Lists the named cpuset(s); if -a is used, will list members of named cpuset; if -r is used, will list recursively
- *-c CPUSPEC, --cpu=CPUSPEC*. Create or modify cpuset in the specified cpuset with CPUSPEC specification
- *-m MEMSPEC, --mem=MEMSPEC*. Specify which memory nodes to assign to the created or modified cpuset
- *-d, --destroy*. Destroy specified cpuset
- *-s CPUSET, --set=CPUSET*. Specify cpuset name to be acted on
- *-r, --recurse*. Do recursive listing, for use with --list
- *-v, --verbose*. Prints more detailed output, for the set command, using this flag will not chop listing to fit in 80 columns
- *--cpu_exclusive*. Mark this cpuset as owning its CPUs exclusively

- *--mem_exclusive*. Mark this cpuset as owning its MEMs exclusively

License

Cpuset is licensed under the GNU GPL V2 only.

Author

Written by Alex Tsariounov (alext@novell.com).

See Also

- cset-set(1), cset-proc(1), cset-shield(1)
- /usr/share/doc/packages/cpuset/html/tutorial.html
- taskset(1), chrt(1)
  /usr/src/linux/Documentation/cpusets.txt