CPU hotplug in the Kernel

Date: September, 2021
Author: Sebastian Andrzej Siewior <bigeasy@linutronix.de>, Rusty Russell <rusty@rustcorp.com.au>, Srivatsa Vaddagiri <vatsa@in.ibm.com>, Ashok Raj <ashok.raj@intel.com>, Joel Schopp <jschopp@austin.ibm.com>, Thomas Gleixner <tglx@linutronix.de>

Introduction

Modern advances in system architectures have introduced advanced error reporting and correction capabilities in processors. There are couple OEMS that support NUMA hardware which are hot pluggable as well, where physical node insertion and removal require support for CPU hotplug.

Such advances require CPUs available to a kernel to be removed either for provisioning reasons, or for RAS purposes to keep an offending CPU off system execution path. Hence the need for CPU hotplug support in the Linux kernel.

A more novel use of CPU-hotplug support is its use today in suspend resume support for SMP. Dual-core and HT support makes even a laptop run SMP kernels which didn't support these methods.

Command Line Switches

`maxcpus=n`

Restrict boot time CPUs to *n*. Say if you have four CPUs, using `maxcpus=2` will only boot two. You can choose to bring the other CPUs later online.

`nr_cpus=n`

Restrict the total amount of CPUs the kernel will support. If the number supplied here is lower than the number of physically available CPUs, then those CPUs can not be brought online later.

`additional_cpus=n`

Use this to limit hotpluggable CPUs. This option sets `cpu_possible_mask = cpu_present_mask + additional_cpus`

This option is limited to the IA64 architecture.

`possible_cpus=n`

This option sets `possible_cpus` bits in `cpu_possible_mask`.

This option is limited to the X86 and S390 architecture.

`cpu0_hotplug`

Allow to shutdown CPU0.

This option is limited to the X86 architecture.

CPU maps

`cpu_possible_mask`
Bitmap of possible CPUs that can ever be available in the system. This is used to allocate some boot time memory for per_cpu variables that aren't designed to grow/shrink as CPUs are made available or removed. Once set during boot time discovery phase, the map is static, i.e no bits are added or removed anytime. Trimming it accurately for your system needs upfront can save some boot time memory.

`cpu_online_mask`
Bitmap of all CPUs currently online. Its set in `__cpu_up()` after a CPU is available for kernel scheduling and ready to receive interrupts from devices. Its cleared when a CPU is brought down using `__cpu_disable()`, before which all OS services including interrupts are migrated to another target CPU.

`cpu_present_mask`
Bitmap of CPUs currently present in the system. Not all of them may be online. When physical hotplug is processed by the relevant subsystem (e.g ACPI) can change and new bit either be added or removed from the map depending on the event is hot-add/hot-remove. There are currently no locking rules as of now. Typical usage is to init topology during boot, at which time hotplug is disabled.

You really don't need to manipulate any of the system CPU maps. They should be read-only for most use. When setting up per-cpu resources almost always use `cpu_possible_mask` or `for_each_possible_cpu()` to iterate. To macro `for_each_cpu()` can be used to iterate over a custom CPU mask.

Never use anything other than `cpumask_t` to represent bitmap of CPUs.

Using CPU hotplug

The kernel option *CONFIG_HOTPLUG_CPU* needs to be enabled. It is currently available on multiple architectures including ARM, MIPS, PowerPC and X86. The configuration is done via the sysfs interface:

```
$ ls -lh /sys/devices/system/cpu
total 0
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu0
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu1
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu2
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu3
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu4
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu5
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu6
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu7
drwxr-xr-x  2 root root    0 Dec 21 16:33 hotplug
-r--r--r--  1 root root 4.0K Dec 21 16:33 offline
-r--r--r--  1 root root 4.0K Dec 21 16:33 online
-r--r--r--  1 root root 4.0K Dec 21 16:33 possible
-r--r--r--  1 root root 4.0K Dec 21 16:33 present
```

The files *offline*, *online*, *possible*, *present* represent the CPU masks. Each CPU folder contains an *online* file which controls the logical on (1) and off (0) state. To logically shutdown CPU4:

```
$ echo 0 > /sys/devices/system/cpu/cpu4/online
 smpboot: CPU 4 is now offline
```

Once the CPU is shutdown, it will be removed from */proc/interrupts*, */proc/cpuinfo* and should also not be shown visible by the *top* command. To bring CPU4 back online:

```
$ echo 1 > /sys/devices/system/cpu/cpu4/online
smpboot: Booting Node 0 Processor 4 APIC 0x1
```

The CPU is usable again. This should work on all CPUs. CPU0 is often special and excluded from CPU hotplug. On X86 the kernel option *CONFIG_BOOTPARAM_HOTPLUG_CPU0* has to be enabled in order to be able to shutdown CPU0. Alternatively the kernel command option *cpu0_hotplug* can be used. Some known dependencies of CPU0:

- Resume from hibernate/suspend. Hibernate/suspend will fail if CPU0 is offline.
- PIC interrupts. CPU0 can't be removed if a PIC interrupt is detected.

Please let Fenghua Yu <fenghua.yu@intel.com> know if you find any dependencies on CPU0.

The CPU hotplug coordination

The offline case

Once a CPU has been logically shutdown the teardown callbacks of registered hotplug states will be invoked, starting with `CPUHP_ONLINE` and terminating at state `CPUHP_OFFLINE`. This includes:

- If tasks are frozen due to a suspend operation then *cpuhp_tasks_frozen* will be set to true.
- All processes are migrated away from this outgoing CPU to new CPUs. The new CPU is chosen from each process' current cpuset, which may be a subset of all online CPUs.
- All interrupts targeted to this CPU are migrated to a new CPU
- timers are also migrated to a new CPU
- Once all services are migrated, kernel calls an arch specific routine `__cpu_disable()` to perform arch specific cleanup.

The CPU hotplug API

CPU hotplug state machine

CPU hotplug uses a trivial state machine with a linear state space from CPUHP_OFFLINE to CPUHP_ONLINE. Each state has a startup and a teardown callback.

When a CPU is onlined, the startup callbacks are invoked sequentially until the state CPUHP_ONLINE is reached. They can also be invoked when the callbacks of a state are set up or an instance is added to a multi-instance state.

When a CPU is offlined the teardown callbacks are invoked in the reverse order sequentially until the state CPUHP_OFFLINE is reached. They can also be invoked when the callbacks of a state are removed or an instance is removed from a multi-instance state.

If a usage site requires only a callback in one direction of the hotplug operations (CPU online or CPU offline) then the other not-required callback can be set to NULL when the state is set up.

The state space is divided into three sections:

- The PREPARE section

  The PREPARE section covers the state space from CPUHP_OFFLINE to CPUHP_BRINGUP_CPU.

  The startup callbacks in this section are invoked before the CPU is started during a CPU online operation. The teardown callbacks are invoked after the CPU has become dysfunctional during a CPU offline operation.

The callbacks are invoked on a control CPU as they can't obviously run on the hotplugged CPU which is either not yet started or has become dysfunctional already.

The startup callbacks are used to setup resources which are required to bring a CPU successfully online. The teardown callbacks are used to free resources or to move pending work to an online CPU after the hotplugged CPU became dysfunctional.

The startup callbacks are allowed to fail. If a callback fails, the CPU online operation is aborted and the CPU is brought down to the previous state (usually CPUHP_OFFLINE) again.

The teardown callbacks in this section are not allowed to fail.

- The STARTING section

  The STARTING section covers the state space between CPUHP_BRINGUP_CPU + 1 and CPUHP_AP_ONLINE.

  The startup callbacks in this section are invoked on the hotplugged CPU with interrupts disabled during a CPU online operation in the early CPU setup code. The teardown callbacks are invoked with interrupts disabled on the hotplugged CPU during a CPU offline operation shortly before the CPU is completely shut down.

  The callbacks in this section are not allowed to fail.

  The callbacks are used for low level hardware initialization/shutdown and for core subsystems.

- The ONLINE section

  The ONLINE section covers the state space between CPUHP_AP_ONLINE + 1 and CPUHP_ONLINE.

  The startup callbacks in this section are invoked on the hotplugged CPU during a CPU online operation. The teardown callbacks are invoked on the hotplugged CPU during a CPU offline operation.

  The callbacks are invoked in the context of the per CPU hotplug thread, which is pinned on the hotplugged CPU. The callbacks are invoked with interrupts and preemption enabled.

  The callbacks are allowed to fail. When a callback fails the hotplug operation is aborted and the CPU is brought back to the previous state.

CPU online/offline operations

A successful online operation looks like this:

```
[CPUHP_OFFLINE]
[CPUHP_OFFLINE + 1]->startup()       -> success
[CPUHP_OFFLINE + 2]->startup()       -> success
[CPUHP_OFFLINE + 3]                  -> skipped because startup == NULL
...
[CPUHP_BRINGUP_CPU]->startup()       -> success
=== End of PREPARE section
[CPUHP_BRINGUP_CPU + 1]->startup()   -> success
...
[CPUHP_AP_ONLINE]->startup()         -> success
=== End of STARTUP section
[CPUHP_AP_ONLINE + 1]->startup()     -> success
...
[CPUHP_ONLINE - 1]->startup()        -> success
[CPUHP_ONLINE]
```

A successful offline operation looks like this:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
[CPUHP_AP_ONLINE + 1]->teardown()    -> success
=== Start of STARTUP section
[CPUHP_AP_ONLINE]->teardown()        -> success
...
[CPUHP_BRINGUP_ONLINE - 1]->teardown()
...
=== Start of PREPARE section
[CPUHP_BRINGUP_CPU]->teardown()
[CPUHP_OFFLINE + 3]->teardown()
[CPUHP_OFFLINE + 2]                  -> skipped because teardown == NULL
[CPUHP_OFFLINE + 1]->teardown()
[CPUHP_OFFLINE]
```

A failed online operation looks like this:

```
[CPUHP_OFFLINE]
[CPUHP_OFFLINE + 1]->startup()       -> success
[CPUHP_OFFLINE + 2]->startup()       -> success
[CPUHP_OFFLINE + 3]                  -> skipped because startup == NULL
...
[CPUHP_BRINGUP_CPU]->startup()       -> success
=== End of PREPARE section
```

```
[CPUHP_BRINGUP_CPU + 1]->startup()   -> success
...
[CPUHP_AP_ONLINE]->startup()         -> success
=== End of STARTUP section
[CPUHP_AP_ONLINE + 1]->startup()     -> success
---
[CPUHP_AP_ONLINE + N]->startup()     -> fail
[CPUHP_AP_ONLINE + (N - 1)]->teardown()
...
[CPUHP_AP_ONLINE + 1]->teardown()
=== Start of STARTUP section
[CPUHP_AP_ONLINE]->teardown()
...
[CPUHP_BRINGUP_ONLINE - 1]->teardown()
...
=== Start of PREPARE section
[CPUHP_BRINGUP_CPU]->teardown()
[CPUHP_OFFLINE + 3]->teardown()
[CPUHP_OFFLINE + 2]                  -> skipped because teardown == NULL
[CPUHP_OFFLINE + 1]->teardown()
[CPUHP_OFFLINE]
```

A failed offline operation looks like this:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()
...
[CPUHP_ONLINE - 1]->startup()
[CPUHP_ONLINE]
```

Recursive failures cannot be handled sensibly. Look at the following example of a recursive fail due to a failed offline operation:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()  -> success
[CPUHP_ONLINE - (N - 2)]->startup()  -> fail
```

The CPU hotplug state machine stops right here and does not try to go back down again because that would likely result in an endless loop:

```
[CPUHP_ONLINE - (N - 1)]->teardown() -> success
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()  -> success
[CPUHP_ONLINE - (N - 2)]->startup()  -> fail
[CPUHP_ONLINE - (N - 1)]->teardown() -> success
[CPUHP_ONLINE - N]->teardown()       -> fail
```

Lather, rinse and repeat. In this case the CPU left in state:

```
[CPUHP_ONLINE - (N - 1)]
```

which at least lets the system make progress and gives the user a chance to debug or even resolve the situation.

Allocating a state

There are two ways to allocate a CPU hotplug state:

- Static allocation

  Static allocation has to be used when the subsystem or driver has ordering requirements versus other CPU hotplug states. E.g. the PERF core startup callback has to be invoked before the PERF driver startup callbacks during a CPU online operation. During a CPU offline operation the driver teardown callbacks have to be invoked before the core teardown callback. The statically allocated states are described by constants in the cpuhp_state enum which can be found in include/linux/cpuhotplug.h.

  Insert the state into the enum at the proper place so the ordering requirements are fulfilled. The state constant has to be used for state setup and removal.

  Static allocation is also required when the state callbacks are not set up at runtime and are part of the initializer of the CPU hotplug state array in kernel/cpu.c.

- Dynamic allocation

  When there are no ordering requirements for the state callbacks then dynamic allocation is the preferred method. The state number is allocated by the setup function and returned to the caller on success.

  Only the PREPARE and ONLINE sections provide a dynamic allocation range. The STARTING section does not as most of the callbacks in that section have explicit ordering requirements.

Setup of a CPU hotplug state

The core code provides the following functions to setup a state:

- cpuhp_setup_state(state, name, startup, teardown)
- cpuhp_setup_state_nocalls(state, name, startup, teardown)
- cpuhp_setup_state_cpuslocked(state, name, startup, teardown)
- cpuhp_setup_state_nocalls_cpuslocked(state, name, startup, teardown)

For cases where a driver or a subsystem has multiple instances and the same CPU hotplug state callbacks need to be invoked for each instance, the CPU hotplug core provides multi-instance support. The advantage over driver specific instance lists is that the instance related functions are fully serialized against CPU hotplug operations and provide the automatic invocations of the state callbacks on add and removal. To set up such a multi-instance state the following function is available:

- cpuhp_setup_state_multi(state, name, startup, teardown)

The @state argument is either a statically allocated state or one of the constants for dynamically allocated states - CPUHP_PREPARE_DYN, CPUHP_ONLINE_DYN - depending on the state section (PREPARE, ONLINE) for which a dynamic state should be allocated.

The @name argument is used for sysfs output and for instrumentation. The naming convention is "subsys:mode" or "subsys/driver:mode", e.g. "perf:mode" or "perf/x86:mode". The common mode names are:

prepare	For states in the PREPARE section
dead	For states in the PREPARE section which do not provide a startup callback
starting	For states in the STARTING section
dying	For states in the STARTING section which do not provide a startup callback
online	For states in the ONLINE section
offline	For states in the ONLINE section which do not provide a startup callback

As the @name argument is only used for sysfs and instrumentation other mode descriptors can be used as well if they describe the nature of the state better than the common ones.

Examples for @name arguments: "perf/online", "perf/x86:prepare", "RCU/tree:dying", "sched/waitempty"

The @startup argument is a function pointer to the callback which should be invoked during a CPU online operation. If the usage site does not require a startup callback set the pointer to NULL.

The @teardown argument is a function pointer to the callback which should be invoked during a CPU offline operation. If the usage site does not require a teardown callback set the pointer to NULL.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_setup_state_nocalls(), cpuhp_setup_state_nocalls_cpuslocked() and cpuhp_setup_state_multi() only install the callbacks
- cpuhp_setup_state() and cpuhp_setup_state_cpuslocked() install the callbacks and invoke the @startup callback (if not NULL) for all online CPUs which have currently a state greater than the newly installed state. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  If a callback fails for CPU N then the teardown callback for CPU 0 .. N-1 is invoked to rollback the operation. The state setup fails, the callbacks for the state are not installed and in case of dynamic allocation the allocated state is freed.

The state setup and the callback invocations are serialized against CPU hotplug operations. If the setup function has to be called from a CPU hotplug read locked region, then the _cpuslocked() variants have to be used. These functions cannot be used from within CPU hotplug callbacks.

The function return values:

0	Statically allocated state was successfully set up
>0	Dynamically allocated state was successfully set up. The returned number is the state number which was allocated. If the state callbacks have to be removed later, e.g. module removal, then this number has to be saved by the caller and used as @state argument for the state remove function. For multi-instance states the dynamically allocated state number is also required as @state argument for the instance add/remove operations.
<0	Operation failed

Removal of a CPU hotplug state

To remove a previously set up state, the following functions are provided:

- cpuhp_remove_state(state)
- cpuhp_remove_state_nocalls(state)
- cpuhp_remove_state_nocalls_cpuslocked(state)

- cpuhp_remove_multi_state(state)

The @state argument is either a statically allocated state or the state number which was allocated in the dynamic range by cpuhp_setup_state*(). If the state is in the dynamic range, then the state number is freed and available for dynamic allocation again.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_remove_state_nocalls(), cpuhp_remove_state_nocalls_cpuslocked() and cpuhp_remove_multi_state() only remove the callbacks.
- cpuhp_remove_state() removes the callbacks and invokes the teardown callback (if not NULL) for all online CPUs which have currently a state greater than the removed state. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  In order to complete the removal, the teardown callback should not fail.

The state removal and the callback invocations are serialized against CPU hotplug operations. If the remove function has to be called from a CPU hotplug read locked region, then the _cpuslocked() variants have to be used. These functions cannot be used from within CPU hotplug callbacks.

If a multi-instance state is removed then the caller has to remove all instances first.

Multi-Instance state instance management

Once the multi-instance state is set up, instances can be added to the state:

- cpuhp_state_add_instance(state, node)
- cpuhp_state_add_instance_nocalls(state, node)

The @state argument is either a statically allocated state or the state number which was allocated in the dynamic range by cpuhp_setup_state_multi().

The @node argument is a pointer to an hlist_node which is embedded in the instance's data structure. The pointer is handed to the multi-instance state callbacks and can be used by the callback to retrieve the instance via container_of().

The functions differ in the way how the installed callbacks are treated:

- cpuhp_state_add_instance_nocalls() and only adds the instance to the multi-instance state's node list.
- cpuhp_state_add_instance() adds the instance and invokes the startup callback (if not NULL) associated with @state for all online CPUs which have currently a state greater than @state. The callback is only invoked for the to be added instance. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  If a callback fails for CPU N then the teardown callback for CPU 0 .. N-1 is invoked to rollback the operation, the function fails and the instance is not added to the node list of the multi-instance state.

To remove an instance from the state's node list these functions are available:

- cpuhp_state_remove_instance(state, node)
- cpuhp_state_remove_instance_nocalls(state, node)

The arguments are the same as for the the cpuhp_state_add_instance*() variants above.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_state_remove_instance_nocalls() only removes the instance from the state's node list.
- cpuhp_state_remove_instance() removes the instance and invokes the teardown callback (if not NULL) associated with @state for all online CPUs which have currently a state greater than @state. The callback is only invoked for the to be removed instance. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  In order to complete the removal, the teardown callback should not fail.

The node list add/remove operations and the callback invocations are serialized against CPU hotplug operations. These functions cannot be used from within CPU hotplug callbacks and CPU hotplug read locked regions.

Examples

Setup and teardown a statically allocated state in the STARTING section for notifications on online and offline operations:

```
ret = cpuhp_setup_state(CPUHP_SUBSYS_STARTING, "subsys:starting", subsys_cpu_starting, subsys_cpu_dying);
if (ret < 0)
     return ret;
....
cpuhp_remove_state(CPUHP_SUBSYS_STARTING);
```

Setup and teardown a dynamically allocated state in the ONLINE section for notifications on offline operations:

```
state = cpuhp_setup_state(CPUHP_ONLINE_DYN, "subsys:offline", NULL, subsys_cpu_offline);
if (state < 0)
    return state;
....
cpuhp_remove_state(state);
```

Setup and teardown a dynamically allocated state in the ONLINE section for notifications on online operations without invoking the callbacks:

```
state = cpuhp_setup_state_nocalls(CPUHP_ONLINE_DYN, "subsys:online", subsys_cpu_online, NULL);
if (state < 0)
    return state;
....
cpuhp_remove_state_nocalls(state);
```

Setup, use and teardown a dynamically allocated multi-instance state in the ONLINE section for notifications on online and offline operation:

```
state = cpuhp_setup_state_multi(CPUHP_ONLINE_DYN, "subsys:online", subsys_cpu_online, subsys_cpu_offline)
if (state < 0)
    return state;
....
ret = cpuhp_state_add_instance(state, &inst1->node);
if (ret)
     return ret;
....
ret = cpuhp_state_add_instance(state, &inst2->node);
if (ret)
     return ret;
....
cpuhp_remove_instance(state, &inst1->node);
....
cpuhp_remove_instance(state, &inst2->node);
....
remove_multi_state(state);
```

Testing of hotplug states

One way to verify whether a custom state is working as expected or not is to shutdown a CPU and then put it online again. It is also possible to put the CPU to certain state (for instance *CPUHP_AP_ONLINE*) and then go back to *CPUHP_ONLINE*. This would simulate an error one state after *CPUHP_AP_ONLINE* which would lead to rollback to the online state.

All registered states are enumerated in `/sys/devices/system/cpu/hotplug/states`

```
$ tail /sys/devices/system/cpu/hotplug/states
138: mm/vmscan:online
139: mm/vmstat:online
140: lib/percpu_cnt:online
141: acpi/cpu-drv:online
142: base/cacheinfo:online
143: virtio/net:online
144: x86/mce:online
145: printk:online
168: sched:active
169: online
```

To rollback CPU4 to `lib/percpu_cnt:online` and back online just issue:

```
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
$ echo 140 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
140
```

It is important to note that the teardown callback of state 140 have been invoked. And now get back online:

```
$ echo 169 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
```

With trace events enabled, the individual steps are visible, too:

```
#  TASK-PID   CPU#    TIMESTAMP  FUNCTION
#     | |       |        |         |
    bash-394  [001]  22.976: cpuhp_enter: cpu: 0004 target: 140 step: 169 (cpuhp_kick_ap_work)
 cpuhp/4-31   [004]  22.977: cpuhp_enter: cpu: 0004 target: 140 step: 168 (sched_cpu_deactivate)
 cpuhp/4-31   [004]  22.990: cpuhp_exit:  cpu: 0004  state: 168 step: 168 ret: 0
 cpuhp/4-31   [004]  22.991: cpuhp_enter: cpu: 0004 target: 140 step: 144 (mce_cpu_pre_down)
 cpuhp/4-31   [004]  22.992: cpuhp_exit:  cpu: 0004  state: 144 step: 144 ret: 0
 cpuhp/4-31   [004]  22.993: cpuhp_multi_enter: cpu: 0004 target: 140 step: 143 (virtnet_cpu_down_prep)
 cpuhp/4-31   [004]  22.994: cpuhp_exit:  cpu: 0004  state: 143 step: 143 ret: 0
```

```
    cpuhp/4-31   [004]  22.995: cpuhp_enter: cpu: 0004 target: 140 step: 142 (cacheinfo_cpu_pre_down)
    cpuhp/4-31   [004]  22.996: cpuhp_exit:  cpu: 0004  state: 142 step: 142 ret: 0
       bash-394  [001]  22.997: cpuhp_exit:  cpu: 0004  state: 140 step: 169 ret: 0
       bash-394  [005]  95.540: cpuhp_enter: cpu: 0004 target: 169 step: 140 (cpuhp_kick_ap_work)
    cpuhp/4-31   [004]  95.541: cpuhp_enter: cpu: 0004 target: 169 step: 141 (acpi_soft_cpu_online)
    cpuhp/4-31   [004]  95.542: cpuhp_exit:  cpu: 0004  state: 141 step: 141 ret: 0
    cpuhp/4-31   [004]  95.543: cpuhp_enter: cpu: 0004 target: 169 step: 142 (cacheinfo_cpu_online)
    cpuhp/4-31   [004]  95.544: cpuhp_exit:  cpu: 0004  state: 142 step: 142 ret: 0
    cpuhp/4-31   [004]  95.545: cpuhp_multi_enter: cpu: 0004 target: 169 step: 143 (virtnet_cpu_online)
    cpuhp/4-31   [004]  95.546: cpuhp_exit:  cpu: 0004  state: 143 step: 143 ret: 0
    cpuhp/4-31   [004]  95.547: cpuhp_enter: cpu: 0004 target: 169 step: 144 (mce_cpu_online)
    cpuhp/4-31   [004]  95.548: cpuhp_exit:  cpu: 0004  state: 144 step: 144 ret: 0
    cpuhp/4-31   [004]  95.549: cpuhp_enter: cpu: 0004 target: 169 step: 145 (console_cpu_notify)
    cpuhp/4-31   [004]  95.550: cpuhp_exit:  cpu: 0004  state: 145 step: 145 ret: 0
    cpuhp/4-31   [004]  95.551: cpuhp_enter: cpu: 0004 target: 169 step: 168 (sched_cpu_activate)
    cpuhp/4-31   [004]  95.552: cpuhp_exit:  cpu: 0004  state: 168 step: 168 ret: 0
       bash-394  [005]  95.553: cpuhp_exit:  cpu: 0004  state: 169 step: 140 ret: 0
```

As it an be seen, CPU4 went down until timestamp 22.996 and then back up until 95.552. All invoked callbacks including their return codes are visible in the trace.

Architecture's requirements

The following functions and configurations are required:

`CONFIG_HOTPLUG_CPU`
: This entry needs to be enabled in Kconfig

`__cpu_up()`
: Arch interface to bring up a CPU

`__cpu_disable()`
: Arch interface to shutdown a CPU, no more interrupts can be handled by the kernel after the routine returns. This includes the shutdown of the timer.

`__cpu_die()`
: This actually supposed to ensure death of the CPU. Actually look at some example code in other arch that implement CPU hotplug. The processor is taken down from the `idle()` loop for that specific architecture. `__cpu_die()` typically waits for some per_cpu state to be set, to ensure the processor dead routine is called to be sure positively.

User Space Notification

After CPU successfully onlined or offline udev events are sent. A udev rule like:

```
SUBSYSTEM=="cpu", DRIVERS=="processor", DEVPATH=="/devices/system/cpu/*", RUN+="the_hotplug_receiver.sh"
```

will receive all events. A script like:

```
#!/bin/sh

if [ "${ACTION}" = "offline" ]
then
    echo "CPU ${DEVPATH##*/} offline"

elif [ "${ACTION}" = "online" ]
then
    echo "CPU ${DEVPATH##*/} online"

fi
```

can process the event further.

Kernel Inline Documentations Reference

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]cpu_hotplug.rst, line 756)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/cpuhotplug.h
```