CPU Idle Time Management

Author: Rafael J. Wysocki <rafael.j.wysocki@intel.com>

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 4)

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 4)

Substitution definition contains illegal element <problematic>:

```
<problematic ids="id2" refid="id1">
    :c:type:`struct cpuidle_state <cpuidle_state>`

.. |struct cpuidle_state| replace:: :c:type:`struct cpuidle_state <cpuidle_state>`
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 5)

Unknown interpreted text role "doc".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 5)

Substitution definition contains illegal element <problematic>:

```
<problematic ids="id4" refid="id3">
    :doc:`CPU Performance Scaling <cpufreq>`

.. |cpufreq| replace:: :doc:`CPU Performance Scaling <cpufreq>`
```

Concepts

Modern processors are generally able to enter states in which the execution of a program is suspended and instructions belonging to it are not fetched from memory or executed. Those states are the *idle* states of the processor.

Since part of the processor hardware is not used in idle states, entering them generally allows power drawn by the processor to be reduced and, in consequence, it is an opportunity to save energy.

CPU idle time management is an energy-efficiency feature concerned about using the idle states of processors for this purpose.

Logical CPUs

CPU idle time management operates on CPUs as seen by the *CPU scheduler* (that is the part of the kernel responsible for the distribution of computational work in the system). In its view, CPUs are *logical* units. That is, they need not be separate physical entities and may just be interfaces appearing to software as individual single-core processors. In other words, a CPU is an entity which appears to be fetching instructions that belong to one sequence (program) from memory and executing them, but it need not work this way physically. Generally, three different cases can be consider here.

First, if the whole processor can only follow one sequence of instructions (one program) at a time, it is a CPU. In that case, if the hardware is asked to enter an idle state, that applies to the processor as a whole.

Second, if the processor is multi-core, each core in it is able to follow at least one program at a time. The cores need not be entirely independent of each other (for example, they may share caches), but still most of the time they work physically in parallel with each other, so if each of them executes only one program, those programs run mostly independently of each other at the same time. The entire cores are CPUs in that case and if the hardware is asked to enter an idle state, that applies to the core that asked for it in the first place, but it also may apply to a larger unit (say a "package" or a "cluster") that the core belongs to (in fact, it may apply to an entire hierarchy of larger units containing the core). Namely, if all of the cores in the larger unit except for one have been put into idle states at the "core level" and the remaining core asks the processor to enter an idle state, that may trigger it to put the whole larger

unit into an idle state which also will affect the other cores in that unit.

Finally, each core in a multi-core processor may be able to follow more than one program in the same time frame (that is, each core may be able to fetch instructions from multiple locations in memory and execute them in the same time frame, but not necessarily entirely in parallel with each other). In that case the cores present themselves to software as "bundles" each consisting of multiple individual single-core "processors", referred to as *hardware threads* (or hyper-threads specifically on Intel hardware), that each can follow one sequence of instructions. Then, the hardware threads are CPUs from the CPU idle time management perspective and if the processor is asked to enter an idle state by one of them, the hardware thread (or CPU) that asked for it is stopped, but nothing more happens, unless all of the other hardware threads within the same core also have asked the processor to enter an idle state. In that situation, the core may be put into an idle state individually or a larger unit containing it may be put into an idle state as a whole (if the other cores within the larger unit are in idle states already).

Idle CPUs

Logical CPUs, simply referred to as "CPUs" in what follows, are regarded as *idle* by the Linux kernel when there are no tasks to run on them except for the special "idle" task.

Tasks are the CPU scheduler's representation of work. Each task consists of a sequence of instructions to execute, or code, data to be manipulated while running that code, and some context information that needs to be loaded into the processor every time the task's code is run by a CPU. The CPU scheduler distributes work by assigning tasks to run to the CPUs present in the system.

Tasks can be in various states. In particular, they are *runnable* if there are no specific conditions preventing their code from being run by a CPU as long as there is a CPU available for that (for example, they are not waiting for any events to occur or similar). When a task becomes runnable, the CPU scheduler assigns it to one of the available CPUs to run and if there are no more runnable tasks assigned to it, the CPU will load the given task's context and run its code (from the instruction following the last one executed so far, possibly by another CPU). [If there are multiple runnable tasks assigned to one CPU simultaneously, they will be subject to prioritization and time sharing in order to allow them to make some progress over time.]

The special "idle" task becomes runnable if there are no other runnable tasks assigned to the given CPU and the CPU is then regarded as idle. In other words, in Linux idle CPUs run the code of the "idle" task called *the idle loop*. That code may cause the processor to be put into one of its idle states, if they are supported, in order to save energy, but if the processor does not support any idle states, or there is not enough time to spend in an idle state before the next wakeup event, or there are strict latency constraints preventing any of the available idle states from being used, the CPU will simply execute more or less useless instructions in a loop until it is assigned a new task to run.

The Idle Loop

The idle loop code takes two major steps in every iteration of it. First, it calls into a code module referred to as the *governor* that belongs to the CPU idle time management subsystem called `CPUIdle` to select an idle state for the CPU to ask the hardware to enter. Second, it invokes another code module from the `CPUIdle` subsystem, called the *driver*, to actually ask the processor hardware to enter the idle state selected by the governor.

The role of the governor is to find an idle state most suitable for the conditions at hand. For this purpose, idle states that the hardware can be asked to enter by logical CPUs are represented in an abstract way independent of the platform or the processor architecture and organized in a one-dimensional (linear) array. That array has to be prepared and supplied by the `CPUIdle` driver matching the platform the kernel is running on at the initialization time. This allows `CPUIdle` governors to be independent of the underlying hardware and to work with any platforms that the Linux kernel can run on.

Each idle state present in that array is characterized by two parameters to be taken into account by the governor, the *target residency* and the (worst-case) *exit latency*. The target residency is the minimum time the hardware must spend in the given state, including the time needed to enter it (which may be substantial), in order to save more energy than it would save by entering one of the shallower idle states instead. [The "depth" of an idle state roughly corresponds to the power drawn by the processor in that state.] The exit latency, in turn, is the maximum time it will take a CPU asking the processor hardware to enter an idle state to start executing the first instruction after a wakeup from that state. Note that in general the exit latency also must cover the time needed to enter the given state in case the wakeup occurs when the hardware is entering it and it must be entered completely to be exited in an ordered manner.

There are two types of information that can influence the governor's decisions. First of all, the governor knows the time until the closest timer event. That time is known exactly, because the kernel programs timers and it knows exactly when they will trigger, and it is the maximum time the hardware that the given CPU depends on can spend in an idle state, including the time necessary to enter and exit it. However, the CPU may be woken up by a non-timer event at any time (in particular, before the closest timer triggers) and it generally is not known when that may happen. The governor can only see how much time the CPU actually was idle after it has been woken up (that time will be referred to as the *idle duration* from now on) and it can use that information somehow along with the time until the closest timer to estimate the idle duration in future. How the governor uses that information depends on what algorithm is implemented by it and that is the primary reason for having more than one governor in the `CPUIdle` subsystem.

There are four `CPUIdle` governors available, `menu`, TEO, `ladder` and `haltpoll`. Which of them is used by default depends on the configuration of the kernel and in particular on whether or not the scheduler tick can be stopped by the idle loop. Available governors can be read from the :file:`available_governors`, and the governor can be changed at runtime. The name of the `CPUIdle` governor currently used by the kernel can be read from the :file:`current_governor_ro` or :file:`current_governor` file under

:file:`/sys/devices/system/cpu/cpuidle/` in `sysfs`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 162); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 162); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 162); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 162); *backlink*

Unknown interpreted text role "file".

Which `CPUIdle` driver is used, on the other hand, usually depends on the platform the kernel is running on, but there are platforms with more than one matching driver. For example, there are two drivers that can work with the majority of Intel platforms, `intel_idle` and `acpi_idle`, one with hardcoded idle states information and the other able to read that information from the system's ACPI tables, respectively. Still, even in those cases, the driver chosen at the system initialization time cannot be replaced later, so the decision on which one of them to use has to be made early (on Intel platforms the `acpi_idle` driver will be used if `intel_idle` is disabled for some reason or if it does not recognize the processor). The name of the `CPUIdle` driver currently used by the kernel can be read from the :file:`current_driver` file under :file:`/sys/devices/system/cpu/cpuidle/` in `sysfs`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 172); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 172); *backlink*

Unknown interpreted text role "file".

Idle CPUs and The Scheduler Tick

The scheduler tick is a timer that triggers periodically in order to implement the time sharing strategy of the CPU scheduler. Of course, if there are multiple runnable tasks assigned to one CPU at the same time, the only way to allow them to make reasonable progress in a given time frame is to make them share the available CPU time. Namely, in rough approximation, each task is given a slice of the CPU time to run its code, subject to the scheduling class, prioritization and so on and when that time slice is used up, the CPU should be switched over to running (the code of) another task. The currently running task may not want to give the CPU away voluntarily, however, and the scheduler tick is there to make the switch happen regardless. That is not the only role of the tick, but it is the primary reason for using it.

The scheduler tick is problematic from the CPU idle time management perspective, because it triggers periodically and relatively often (depending on the kernel configuration, the length of the tick period is between 1 ms and 10 ms). Thus, if the tick is allowed to trigger on idle CPUs, it will not make sense for them to ask the hardware to enter idle states with target residencies above the tick period length. Moreover, in that case the idle duration of any CPU will never exceed the tick period length and the energy used for entering and exiting idle states due to the tick wakeups on idle CPUs will be wasted.

Fortunately, it is not really necessary to allow the tick to trigger on idle CPUs, because (by definition) they have no tasks to run except for the special "idle" one. In other words, from the CPU scheduler perspective, the only user of the CPU time on them is the idle loop. Since the time of an idle CPU need not be shared between multiple runnable tasks, the primary reason for using the tick goes away if the given CPU is idle. Consequently, it is possible to stop the scheduler tick entirely on idle CPUs in principle, even

though that may not always be worth the effort.

Whether or not it makes sense to stop the scheduler tick in the idle loop depends on what is expected by the governor. First, if there is another (non-tick) timer due to trigger within the tick range, stopping the tick clearly would be a waste of time, even though the timer hardware may not need to be reprogrammed in that case. Second, if the governor is expecting a non-timer wakeup within the tick range, stopping the tick is not necessary and it may even be harmful. Namely, in that case the governor will select an idle state with the target residency within the time until the expected wakeup, so that state is going to be relatively shallow. The governor really cannot select a deep idle state then, as that would contradict its own expectation of a wakeup in short order. Now, if the wakeup really occurs shortly, stopping the tick would be a waste of time and in this case the timer hardware would need to be reprogrammed, which is expensive. On the other hand, if the tick is stopped and the wakeup does not occur any time soon, the hardware may spend indefinite amount of time in the shallow idle state selected by the governor, which will be a waste of energy. Hence, if the governor is expecting a wakeup of any kind within the tick range, it is better to allow the tick trigger. Otherwise, however, the governor will select a relatively deep idle state, so the tick should be stopped so that it does not wake up the CPU too early.

In any case, the governor knows what it is expecting and the decision on whether or not to stop the scheduler tick belongs to it. Still, if the tick has been stopped already (in one of the previous iterations of the loop), it is better to leave it as is and the governor needs to take that into account.

The kernel can be configured to disable stopping the scheduler tick in the idle loop altogether. That can be done through the build-time configuration of it (by unsetting the `CONFIG_NO_HZ_IDLE` configuration option) or by passing `nohz=off` to it in the command line. In both cases, as the stopping of the scheduler tick is disabled, the governor's decisions regarding it are simply ignored by the idle loop code and the tick is never stopped.

The systems that run kernels configured to allow the scheduler tick to be stopped on idle CPUs are referred to as *tickless* systems and they are generally regarded as more energy-efficient than the systems running kernels in which the tick cannot be stopped. If the given system is tickless, it will use the `menu` governor by default and if it is not tickless, the default `CPUIdle` governor on it will be `ladder`.

The `menu` Governor

The `menu` governor is the default `CPUIdle` governor for tickless systems. It is quite complex, but the basic principle of its design is straightforward. Namely, when invoked to select an idle state for a CPU (i.e. an idle state that the CPU will ask the processor hardware to enter), it attempts to predict the idle duration and uses the predicted value for idle state selection.

It first obtains the time until the closest timer event with the assumption that the scheduler tick will be stopped. That time, referred to as the *sleep length* in what follows, is the upper bound on the time before the next CPU wakeup. It is used to determine the sleep length range, which in turn is needed to get the sleep length correction factor.

The `menu` governor maintains two arrays of sleep length correction factors. One of them is used when tasks previously running on the given CPU are waiting for some I/O operations to complete and the other one is used when that is not the case. Each array contains several correction factor values that correspond to different sleep length ranges organized so that each range represented in the array is approximately 10 times wider than the previous one.

The correction factor for the given sleep length range (determined before selecting the idle state for the CPU) is updated after the CPU has been woken up and the closer the sleep length is to the observed idle duration, the closer to 1 the correction factor becomes (it must fall between 0 and 1 inclusive). The sleep length is multiplied by the correction factor for the range that it falls into to obtain the first approximation of the predicted idle duration.

Next, the governor uses a simple pattern recognition algorithm to refine its idle duration prediction. Namely, it saves the last 8 observed idle duration values and, when predicting the idle duration next time, it computes the average and variance of them. If the variance is small (smaller than 400 square milliseconds) or it is small relative to the average (the average is greater that 6 times the standard deviation), the average is regarded as the "typical interval" value. Otherwise, the longest of the saved observed idle duration values is discarded and the computation is repeated for the remaining ones. Again, if the variance of them is small (in the above sense), the average is taken as the "typical interval" value and so on, until either the "typical interval" is determined or too many data points are disregarded, in which case the "typical interval" is assumed to equal "infinity" (the maximum unsigned integer value). The "typical interval" computed this way is compared with the sleep length multiplied by the correction factor and the minimum of the two is taken as the predicted idle duration.

Then, the governor computes an extra latency limit to help "interactive" workloads. It uses the observation that if the exit latency of the selected idle state is comparable with the predicted idle duration, the total time spent in that state probably will be very short and the amount of energy to save by entering it will be relatively small, so likely it is better to avoid the overhead related to entering that state and exiting it. Thus selecting a shallower state is likely to be a better option then. The first approximation of the extra latency limit is the predicted idle duration itself which additionally is divided by a value depending on the number of tasks that previously ran on the given CPU and now they are waiting for I/O operations to complete. The result of that division is compared with the latency limit coming from the power management quality of service, or PM QoS, framework and the minimum of the two is taken as the limit for the idle states' exit latency.

Now, the governor is ready to walk the list of idle states and choose one of them. For this purpose, it compares the target residency of each state with the predicted idle duration and the exit latency of it with the computed latency limit. It selects the state with the target residency closest to the predicted idle duration, but still below it, and exit latency that does not exceed the limit.

In the final step the governor may still need to refine the idle state selection if it has not decided to stop the scheduler tick. That happens if the idle duration predicted by it is less than the tick period and the tick has not been stopped already (in a previous iteration of the idle loop). Then, the sleep length used in the previous computations may not reflect the real time until the closest timer event and if it really is greater than that time, the governor may need to select a shallower state with a suitable target residency.

The Timer Events Oriented (TEO) Governor

The timer events oriented (TEO) governor is an alternative `CPUIdle` governor for tickless systems. It follows the same basic strategy as the `menu` one: it always tries to find the deepest idle state suitable for the given conditions. However, it applies a different approach to that problem.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 350)**
>
> Unknown directive type "kernel-doc".
>
> ```
> .. kernel-doc:: drivers/cpuidle/governors/teo.c
>    :doc: teo-description
> ```

Representation of Idle States

For the CPU idle time management purposes all of the physical idle states supported by the processor have to be represented as a one-dimensional array of |struct cpuidle_state| objects each allowing an individual (logical) CPU to ask the processor hardware to enter an idle state of certain properties. If there is a hierarchy of units in the processor, one |struct cpuidle_state| object can cover a combination of idle states supported by the units at different levels of the hierarchy. In that case, the target residency and exit latency parameters of it, must reflect the properties of the idle state at the deepest level (i.e. the idle state of the unit containing all of the other units).

For example, take a processor with two cores in a larger unit referred to as a "module" and suppose that asking the hardware to enter a specific idle state (say "X") at the "core" level by one core will trigger the module to try to enter a specific idle state of its own (say "MX") if the other core is in idle state "X" already. In other words, asking for idle state "X" at the "core" level gives the hardware a license to go as deep as to idle state "MX" at the "module" level, but there is no guarantee that this is going to happen (the core asking for idle state "X" may just end up in that state by itself instead). Then, the target residency of the |struct cpuidle_state| object representing idle state "X" must reflect the minimum time to spend in idle state "MX" of the module (including the time needed to enter it), because that is the minimum time the CPU needs to be idle to save any energy in case the hardware enters that state. Analogously, the exit latency parameter of that object must cover the exit time of idle state "MX" of the module (and usually its entry time too), because that is the maximum delay between a wakeup signal and the time the CPU will start to execute the first new instruction (assuming that both cores in the module will always be ready to execute instructions as soon as the module becomes operational as a whole).

There are processors without direct coordination between different levels of the hierarchy of units inside them, however. In those cases asking for an idle state at the "core" level does not automatically affect the "module" level, for example, in any way and the `CPUIdle` driver is responsible for the entire handling of the hierarchy. Then, the definition of the idle state objects is entirely up to the driver, but still the physical properties of the idle state that the processor hardware finally goes into must always follow the parameters used by the governor for idle state selection (for instance, the actual exit latency of that idle state must not exceed the exit latency parameter of the idle state object selected by the governor).

In addition to the target residency and exit latency idle state parameters discussed above, the objects representing idle states each contain a few other parameters describing the idle state and a pointer to the function to run in order to ask the hardware to enter that state. Also, for each |struct cpuidle_state| object, there is a corresponding :c:type:`struct cpuidle_state_usage <cpuidle_state_usage>` one containing usage statistics of the given idle state. That information is exposed by the kernel via `sysfs`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 399); *backlink***
>
> Unknown interpreted text role "c:type".

For each CPU in the system, there is a :file:`/sys/devices/system/cpu/cpu<N>/cpuidle/` directory in `sysfs`, where the number `<N>` is assigned to the given CPU at the initialization time. That directory contains a set of subdirectories called :file:`state0`, :file:`state1` and so on, up to the number of idle state objects defined for the given CPU minus one. Each of these directories corresponds to one idle state object and the larger the number in its name, the deeper the (effective) idle state represented by it. Each of them contains a number of files (attributes) representing the properties of the idle state object corresponding to it, as follows:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-`**

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 408); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 408); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 408); *backlink*

Unknown interpreted text role "file".

`above`
: Total number of times this idle state had been asked for, but the observed idle duration was certainly too short to match its target residency.

`below`
: Total number of times this idle state had been asked for, but certainly a deeper idle state would have been a better match for the observed idle duration.

`desc`
: Description of the idle state.

`disable`
: Whether or not this idle state is disabled.

`default_status`
: The default status of this state, "enabled" or "disabled".

`latency`
: Exit latency of the idle state in microseconds.

`name`
: Name of the idle state.

`power`
: Power drawn by hardware in this idle state in milliwatts (if specified, 0 otherwise).

`residency`
: Target residency of the idle state in microseconds.

`time`
: Total time spent in this idle state by the given CPU (as measured by the kernel) in microseconds.

`usage`
: Total number of times the hardware has been asked by the given CPU to enter this idle state.

`rejected`
: Total number of times a request to enter this idle state on the given CPU was rejected.

The :file:`desc` and :file:`name` files both contain strings. The difference between them is that the name is expected to be more concise, while the description may be longer and it may contain white space or special characters. The other files listed above contain integer numbers.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 462); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 462); *backlink*

Unknown interpreted text role "file".

The :file:`disable` attribute is the only writeable one. If it contains 1, the given idle state is disabled for this particular CPU, which means that the governor will never select it for this particular CPU and the `CPUIdle` driver will never ask the hardware to enter it for that CPU as a result. However, disabling an idle state for one CPU does not prevent it from being asked for by the other CPUs, so it must be disabled for all of them in order to never be asked for by any of them. [Note that, due to the way the `ladder` governor is implemented, disabling an idle state prevents that governor from selecting any idle states deeper than the disabled one too.]

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 467); *backlink*

Unknown interpreted text role "file".

If the :file:`disable` attribute contains 0, the given idle state is enabled for this particular CPU, but it still may be disabled for some or all of the other CPUs in the system at the same time. Writing 1 to it causes the idle state to be disabled for this particular CPU and writing 0 to it allows the governor to take it into consideration for the given CPU and the driver to ask for it, unless that state was disabled globally in the driver (in which case it cannot be used at all).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 477); *backlink*

Unknown interpreted text role "file".

The :file:`power` attribute is not defined very well, especially for idle state objects representing combinations of idle states at different levels of the hierarchy of units in the processor, and it generally is hard to obtain idle state power numbers for complex hardware, so :file:`power` often contains 0 (not available) and if it contains a nonzero number, that number may not be very accurate and it should not be relied on for anything meaningful.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 485); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 485); *backlink*

Unknown interpreted text role "file".

The number in the :file:`time` file generally may be greater than the total time really spent by the given CPU in the given idle state, because it is measured by the kernel and it may not cover the cases in which the hardware refused to enter this idle state and entered a shallower one instead of it (or even it did not enter any idle state at all). The kernel can only measure the time span between asking the hardware to enter an idle state and the subsequent wakeup of the CPU and it cannot say what really happened in the meantime at the hardware level. Moreover, if the idle state object in question represents a combination of idle states at different levels of the hierarchy of units in the processor, the kernel can never say how deep the hardware went down the hierarchy in any particular case. For these reasons, the only reliable way to find out how much time has been spent by the hardware in different idle states supported by it is to use idle state residency counters in the hardware, if available.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 492); *backlink*

Unknown interpreted text role "file".

Generally, an interrupt received when trying to enter an idle state causes the idle state entry request to be rejected, in which case the `CPUIdle` driver may return an error code to indicate that this was the case. The :file:`usage` and :file:`rejected` files report the number of times the given idle state was entered successfully or rejected, respectively.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 506); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 506); *backlink*

Unknown interpreted text role "file".

Power Management Quality of Service for CPUs

The power management quality of service (PM QoS) framework in the Linux kernel allows kernel code and user space processes to set constraints on various energy-efficiency features of the kernel to prevent performance from dropping below a required level.

CPU idle time management can be affected by PM QoS in two ways, through the global CPU latency limit and through the resume latency constraints for individual CPUs. Kernel code (e.g. device drivers) can set both of them with the help of special internal interfaces provided by the PM QoS framework. User space can modify the former by opening the :file:`cpu_dma_latency` special device file under :file:`/dev/` and writing a binary value (interpreted as a signed 32-bit integer) to it. In turn, the resume latency constraint for a CPU can be modified from user space by writing a string (representing a signed 32-bit integer) to the :file:`power/pm_qos_resume_latency_us` file under :file:`/sys/devices/system/cpu/cpu<N>/` in `sysfs`, where the CPU number `<N>` is allocated at the system initialization time. Negative values will be rejected in both cases and, also in both cases, the written integer number will be interpreted as a requested PM QoS constraint in microseconds.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 522); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 522); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 522); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 522); *backlink*

Unknown interpreted text role "file".

The requested value is not automatically applied as a new constraint, however, as it may be less restrictive (greater in this particular case) than another constraint previously requested by someone else. For this reason, the PM QoS framework maintains a list of requests that have been made so far for the global CPU latency limit and for each individual CPU, aggregates them and applies the effective (minimum in this particular case) value as the new constraint.

In fact, opening the :file:`cpu_dma_latency` special device file causes a new PM QoS request to be created and added to a global priority list of CPU latency limit requests and the file descriptor coming from the "open" operation represents that request. If that file descriptor is then used for writing, the number written to it will be associated with the PM QoS request represented by it as a new requested limit value. Next, the priority list mechanism will be used to determine the new effective value of the entire list of requests and that effective value will be set as a new CPU latency limit. Thus requesting a new limit value will only change the real limit if the effective "list" value is affected by it, which is the case if it is the minimum of the requested values in the list.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 544); *backlink*

Unknown interpreted text role "file".

The process holding a file descriptor obtained by opening the :file:`cpu_dma_latency` special device file controls the PM QoS request associated with that file descriptor, but it controls this particular PM QoS request only.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst`, line 556); *backlink*

Unknown interpreted text role "file".

Closing the :file:`cpu_dma_latency` special device file or, more precisely, the file descriptor obtained while opening it, causes the PM

QoS request associated with that file descriptor to be removed from the global priority list of CPU latency limit requests and destroyed. If that happens, the priority list mechanism will be used again, to determine the new effective value for the whole list and that value will become the new limit.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 561); *backlink***
>
> Unknown interpreted text role "file".

In turn, for each CPU there is one resume latency PM QoS request associated with the :file:`power/pm_qos_resume_latency_us` file under :file:`/sys/devices/system/cpu/cpu<N>/` in `sysfs` and writing to it causes this single PM QoS request to be updated regardless of which user space process does that. In other words, this PM QoS request is shared by the entire user space, so access to the file associated with it needs to be arbitrated to avoid confusion. [Arguably, the only legitimate use of this mechanism in practice is to pin a process to the CPU in question and let it use the `sysfs` interface to control the resume latency constraint for it.] It is still only a request, however. It is an entry in a priority list used to determine the effective value to be set as the resume latency constraint for the CPU in question every time the list of requests is updated this way or another (there may be other requests coming from kernel code in that list).

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 568); *backlink***
>
> Unknown interpreted text role "file".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 568); *backlink***
>
> Unknown interpreted text role "file".

CPU idle time governors are expected to regard the minimum of the global (effective) CPU latency limit and the effective resume latency constraint for the given CPU as the upper limit for the exit latency of the idle states that they are allowed to select for that CPU. They should never select any idle states with exit latency beyond that limit.

Idle States Control Via Kernel Command Line

In addition to the `sysfs` interface allowing individual idle states to be disabled for individual CPUs, there are kernel command line parameters affecting CPU idle time management.

The `cpuidle.off=1` kernel command line option can be used to disable the CPU idle time management entirely. It does not prevent the idle loop from running on idle CPUs, but it prevents the CPU idle time governors and drivers from being invoked. If it is added to the kernel command line, the idle loop will ask the hardware to enter idle states on idle CPUs via the CPU architecture support code that is expected to provide a default mechanism for this purpose. That default mechanism usually is the least common denominator for all of the processors implementing the architecture (i.e. CPU instruction set) in question, however, so it is rather crude and not very energy-efficient. For this reason, it is not recommended for production use.

The `cpuidle.governor=` kernel command line switch allows the `CPUIdle` governor to use to be specified. It has to be appended with a string matching the name of an available governor (e.g. `cpuidle.governor=menu`) and that governor will be used instead of the default one. It is possible to force the `menu` governor to be used on the systems that use the `ladder` governor by default this way, for example.

The other kernel command line parameters controlling CPU idle time management described below are only relevant for the *x86* architecture and some of them affect Intel processors only.

The *x86* architecture support code recognizes three kernel command line options related to CPU idle time management: `idle=poll`, `idle=halt`, and `idle=nomwait`. The first two of them disable the `acpi_idle` and `intel_idle` drivers altogether, which effectively causes the entire `CPUIdle` subsystem to be disabled and makes the idle loop invoke the architecture support code to deal with idle CPUs. How it does that depends on which of the two parameters is added to the kernel command line. In the `idle=halt` case, the architecture support code will use the `HLT` instruction of the CPUs (which, as a rule, suspends the execution of the program and causes the hardware to attempt to enter the shallowest available idle state) for this purpose, and if `idle=poll` is used, idle CPUs will execute a more or less "lightweight" sequence of instructions in a tight loop. [Note that using `idle=poll` is somewhat drastic in many cases, as preventing idle CPUs from saving almost any energy at all may not be the only effect of it. For example, on Intel hardware it effectively prevents CPUs from using P-states (see |cpufreq|) that require any number of CPUs in a package to be idle, so it very well may hurt single-thread computations performance as well as energy-efficiency. Thus using it for performance reasons may not be a good idea at all.]

The `idle=nomwait` option disables the `intel_idle` driver and causes `acpi_idle` to be used (as long as all of the information

needed by it is there in the system's ACPI tables), but it is not allowed to use the `MWAIT` instruction of the CPUs to ask the hardware to enter idle states.

In addition to the architecture-level kernel command line options affecting CPU idle time management, there are parameters affecting individual `CPUIdle` drivers that can be passed to them via the kernel command line. Specifically, the `intel_idle.max_cstate=<n>` and `processor.max_cstate=<n>` parameters, where `<n>` is an idle state index also used in the name of the given state's directory in `sysfs` (see Representation of Idle States), causes the `intel_idle` and `acpi_idle` drivers, respectively, to discard all of the idle states deeper than idle state `<n>`. In that case, they will never ask for any of those idle states or expose them to the governor. [The behavior of the two drivers is different for `<n>` equal to `0`. Adding `intel_idle.max_cstate=0` to the kernel command line disables the `intel_idle` driver and allows `acpi_idle` to be used, whereas `processor.max_cstate=0` is equivalent to `processor.max_cstate=1`. Also, the `acpi_idle` driver is part of the `processor` kernel module that can be loaded separately and `max_cstate=<n>` can be passed to it as a module parameter when it is loaded.]

Docutils System Messages

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 358); *backlink*

Undefined substitution referenced: "struct cpuidle_state".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 358); *backlink*

Undefined substitution referenced: "struct cpuidle_state".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 369); *backlink*

Undefined substitution referenced: "struct cpuidle_state".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 399); *backlink*

Undefined substitution referenced: "struct cpuidle_state".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\pm\[linux-master][Documentation][admin-guide][pm]cpuidle.rst, line 618); *backlink*

Undefined substitution referenced: "cpufreq".