Using Processor Idle C-States with Linux on ThinkSystem Servers

Planning / Implementation

When a processor has no task to run, or is sleeping or waiting some resource to be available, it is reasonable in most cases to move the processor to a lower frequency or even an inactive state, where part or all of the processor components stop running. The longer the processor stays in an inactive state, the greater the power savings are.

There are two types of C-States, ACPI and MWAIT.

Overview of ACPI C-States

ACPI C-States

Processor inactive states are called C-States in ACPI specification. ACPI defines multiple C-States, i.e. C0, C1, …, Cn. C0 is the active state where the processor executes instructions. C1 through Cn are processor sleeping states. Below are the definitions of the first three C-states.

- **C1** must be supported by all processors, and the default support is implemented through a native instruction of the processor, for example HLT for IA 32-bit processors. The cache contents and coherency are maintained in the C-state.
- **C2** is optionally supported and improves power savings over C1. The processor is assumed capable of keeping its caches coherent; for example, bus master and multiprocessor activity can take place without corrupting cache context.
- **C3** is optionally supported and improves power savings over C2. The cache contents are maintained but coherency is not because the C-state does not require the processor to snoop on memory accesses.

The C-states differ in their power savings, but more power saving means more entering and exiting latency, because the time taken is proportional to the number of the processor parts involved. This is the major concern of the processor C-state management.

ACPI C-State Enumeration

ACPI exposes C-state information to OS through _CST (C1 to C3 is also enumerable through FADT) which is defined as below _CST is an ACPI object of type Package. A Package represents an array of ACPI objects. It has three fields, as follows:

- **Type** indicates the type of the ACPI object and here it must be of type Package.
- **Count** contains the size of the ACPI object array.
- **Pointer** points to the ACPI object array.

```
Package {
    Type              // Object Type (PACKAGE)
    Count             // Integer
    Pointer           // Pointer to an ACPI object array
}
```

Of the ACPI object array, the first element is an integer object which indicates the number of CState objects that follow.

```
CStateACPIObjectArray {
    Register          // Buffer (Resource Descriptor) Object
    Type              // Integer (BYTE) Object
    Latency           // Integer (WORD) Object
    Power             // Integer (DWORD) Object
}
```

Of the properties,

- **Register** describes the register that OS must use to place the processor in the corresponding C-state. The register may locate in system IO space or is a Functional Fixed Hardware (FFH).
- **Type** is the type of the C-state with 1=C1, 2=C2, 3=C3, etc.
- **Latency** is the worst-case latency in microseconds to enter and exit the C-state.
- **Power** is the average power consumption in milliwatts of the processor in the C-state.

ACPI C-State Entering

OS accesses the register of an ACPI C-state to enter it, but the access method depends on the register type. For a system IO register, property Register saves its address, and OS reads the address for entering the C-state via the following CPU instruction ‘inb’.

```
inb register-address
```

For a Functional Fixed Hardware (FFH) register, property Register saves the processor specific C-state index, and MWAIT will be used to enter the C-state, which is described in the next section.

Note that C1 is always valid even if _CST does not exist, and it is always supported to enter C1 by using native instructions like HLT for x86.

ACPI C-State Exiting

The processor can exit C1, C2, and C3 for any reason, but must always exit this state whenever an interrupt is to be presented to the processor.

For FFH C-states, to satisfy this requirement, MWAIT must be checked to ensure that it supports treating interrupts as break-event, even when the interrupts are disabled.

Overview of MWAIT C-States

Besides the standard way of ACPI C-states, CPUs based on Intel Core microarchitecture provide processor-specific C-states, which might include additional deeper C-states that are unavailable in ACPI ones.

The processor specific C-states are accessible using MWAIT extensions, whose availability can be checked with CPUID. If `CPUID.05H.ECX[Bit 0]` = 1, the processor supports MWAIT extension. All ThinkSystem servers with Intel Xeon or AMD EPYC processors support MWAIT. Use the following command to verify MWAIT support:

```
lspci|grep monitor
```

MWAIT uses EAX to communicate a hint that CPU can enter a specified processor specific target C-state. Unlike ACPI C-states, a processor specific C-state may have sub C-states. Thus, a processor specific C-state index comprises two parts, i.e., C-state and sub C-state as the figure below shows. As ACPI C-states, the numerically higher processor specific C-state has higher power savings and latency.

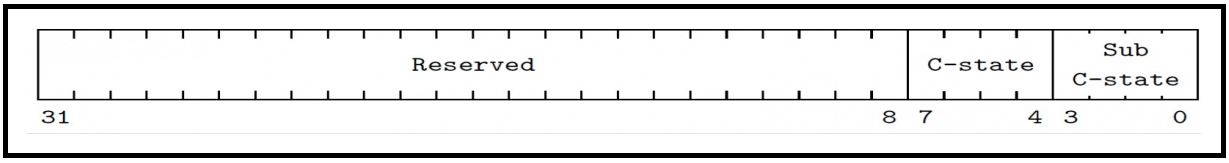

Figure 1. Processor specific C-state and sub C-state in EAX register

The processor specific C-states for a CPU model are defined in CPU model's specification. Below are processor specific and ACPI C-states for Ice Lake. If the index of a processor specific C-state and the address of an ACPI C-state are equal, the two C-states are the same though they might have different names. This is the case for the two C-states of the same row in the figure below. Note that C-state C1E exists only for processor specific ones.

Processor Specific C-state Name	C-state Index		ACPI C-state Name	Type	Address
	C-state	Sub C-state			
C1	0	0	C1	1	0
C1E	0	1	-	-	-
C6	2	0	C2	2	0x20

Figure 2. Processor specific C-state and ACPI C-state

MWAIT C-State Entering and Exiting

MONITOR/MWAIT instructions should be used in pair. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. But setting up MONITOR without executing MWAIT has no adverse effects.

Below is an example of the MONITOR/MWAIT pair used to put the processor to a C-state when there is no work to be scheduled. Register eax provides the current thread's flag address to arm the address monitoring hardware. When a task is to be scheduled to the idle processor, the member flags of thread_info of the idle task on the processor is set accordingly. This causes the processor to exit the C-state.

Register ecx is set to 1 for MWAIT, so that interrupts can break the C-state even if masked.

```
eax = ¤t_thread_info->flags
ecx = 0
edx = 0
monitor   eax, ecx, edx
if (!need_resched()) {
    eax = (processor-specific-c-state-index)
    ecx = 1
    mwait   eax, ecx
}
```

Linux Processor C-States Management Subsystem

On Linux, processor C-States are managed via the idle task and CPUIdle subsystem. When there is nothing to do for a CPU, Linux assigns the CPU the special "idle" task to do the CPU idle time management. The CPU is then regarded as idle. The "idle" task is also called the idle loop because it repeats finding opportunities to put the idle CPU to deep sleep states when there are no other tasks to run. It completes sleep states entering with the help of CPUIdle subsystem.

Idle Task

Idle task is a dead loop. In each loop cycle, it begins by checking if there are other processes that need to be scheduled to run. If yes, it calls the scheduler to do task scheduling. If not, it calls the CPUIdle subsystem to choose a sleep state appropriate for the current CPU running state, and then calls CPUIdle subsystem to drive the CPU to enter the selected sleep state.

When it receives an interrupt, the system exits the sleep state and starts to handle the interrupt. The interrupt could be from another CPU and fired by the scheduler because the idle CPU needs to handle new tasks. The interrupt could be due to a completed IO command or the scheduler tick.

After interrupt handling, the idle task resumes running the instructions following the sleep state entering call. It rechecks for possible rescheduling. And hence another loop cycle begins.

CPUIdle Subsystem

Sleep states entering is initiated by the idle task, since only the idle task knows when the system is going idle. But it does not know what sleep state fits the current system running status best, and how to put the system into a sleep state. All these issues are addressed by the CPUIdle subsystem.

The CPUIdle subsystem is composed of CPUIdle core, CPUIdle drivers, and CPUIdle governors as shown in the figure below. The CPUIdle governor finds the sleep state most suitable for the condition at hand. The CPUIdle driver is responsible for C-state enumeration and entering.

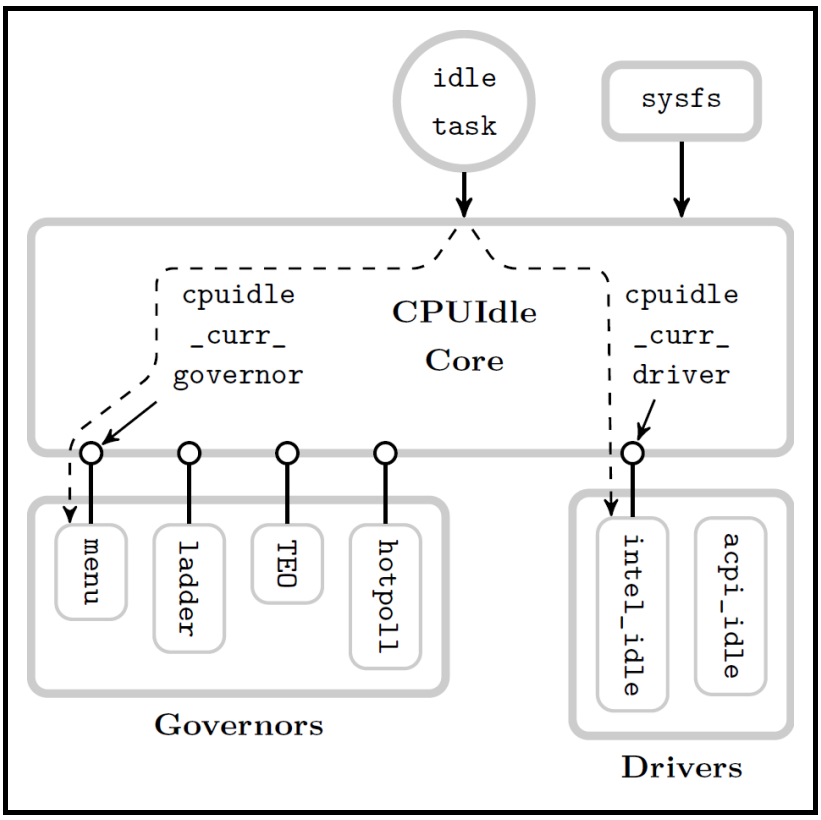

Figure 3. CPUIdle subsystem architecture

This isolates the idle task from the duty of sleep state selection and from specifics and varieties of the underlying CPU hardware. But it is still necessary to modify the idle task to support new governors or drivers.

The CPUIdle core is designed as a framework that decouples the hard-coded interconnections between the idle task and the governors and drivers. It presents a unified and abstract interface to the idle task, governors, and drivers.

The idle task calls CPUIdle core interfaces to select the sleep state and execute it without direct contact with the governors and drivers. This keeps the idle task insulated from CPUIdle subsystem implementation modifications only if the interface is stable.

The governors and drivers benefit from the framework design as well. Each can concentrate on implementing the interface functions defined by the framework needing no knowledge of the idle task and how it will use them.

CPUIdle Governors

Unlike CPUidle drivers, CPUIdle governors are platform neutral. They use the abstract C-state array provided by the driver, which contains data that is generic to all platforms, to do C-state selection.

The best C-state for a CPU is the one that saves the largest power with an acceptable latency. Every C-state has a target residency, which is the smallest time CPU is expected to stay in the C-state. Besides, there is adjustable CPU latency requirement from PM QoS to be met, and this means that the C-state exit latency should not exceed the CPU latency requirement. Normally, the deeper a C-state is, the longer its target residency and exit latency become. Thus, the governor chooses the deepest C-state with target residency less than CPU idle time and exit latency less than CPU latency requirement.

However, the CPU idle time is not available, and so the governor has to predict it. There are many factors that the governor can use for prediction. Of them, two time factors are the most important:

- The time period from the current time to the closest timer event when the governor is invoked to select a C-state. It is the maximum time that the CPU can spend in an idle state, which covers the time necessary to enter and exit the idle state. However, the CPU may be woken up by a non-timer event at any time.
- The time period in which the CPU stayed idle, when the CPU exited from the selected C-state. The governor can use it along with the time of the closest timer to predict the idle duration in future.

How the governor performs prediction depends on what algorithm it uses, and that is the primary reason for having more than one governor.

There are four governors available:

- menu
- ladder
- TEO
- haltpoll

Haltpoll is special in that it is for virtualization. There can be only one governor active, but it can be changed on demand.

The governor simply registers with the core for it to be available. There can be only one governor in use at a time. If there are multiple governors registered, the one with the highest rating is active.

CPUIdle Drivers

A CPUIdle driver first gets the C-state information from the underlying hardware or firmware and saves the C-states in a linear array. A C-state array element records a C-state's name, its power usage, the latencies to enter and exit it, how to enter and exit it, etc. The elements in the array are arranged in descending order of power consumption.

After the C-state array is prepared, the driver registers itself with the CPUIdle core, and passes the C-state array to the core. The core will use the first registered driver as the current driver to drive C-states. That is, there is at most one driver active, and the active driver cannot be changed once assigned.

Finally, the driver registers CPUs as CPUIdle devices with the CPUIdle core. After this, CPUs are ready for C-state management.

Currently, two drivers are available for x86 platforms:

- acpi_idle, which supports ACPI C-States and is supported on Intel and AMD processors
- intel_idle, which supports MWAIT C-States and is supported only on Intel processors

intel_idle is hard coded to register before acpi_idle. Therefore, if the Intel platforms that support both ACPI and MWAIT C-state, intel_idle will be used. acpi_idle is used when the platform supports only the ACPI C-state or the intel_idle driver is disabled.

Managing Linux Processor C-States

Linux provides many ways to manage CPU C-states and the CPUIdle subsystem itself. Using them, we can customize the system to best fit the application environment.

Most information of the CPUIdle subsystem is exposed via sysfs as files under directory /sys/devices/system/cpu/cpuidle/.

Available governors can be read from the file available_governors. The name of the gorvernor currently used can be read from current_governor_ro or current_governor. The latter can also be used to switch governors: writing a governor's name to current_governor switches the currently used governor to it. The name of the currently used driver can be read from current_driver.

For each CPU n, there is a sysfs directory /sys/devices/system/cpu/cpun/cpuidle/, which contains a set of sub-directories "state*i*" corresponding to the C-states of CPU n. Each of the sub-directories contains several files representing the attributes of the C-state:

- **desc:** description of the C-state.
- **disable:** whether or not the C-state is disabled.
- **latency:** exit latency of the C-state in microseconds.
- **name:** name of the C-state.
- **power:** power drawn by hardware in the C-state in milliwatts if specified; 0 otherwise.
- **residency:** target residency of the C-state in microseconds.
- **time:** total time spent in the C-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 the C-state.

Of the attribute files, **disable** is the only writable one. If it contains 1, the given C-state is disabled for this particular CPU, which means that the governor will never select it for the CPU, and hence the CPU will never enter the C-state. If **disable** contains 0, the given C-state is ready for selection and entrance for the CPU. **disable** is per-CPU, i.e., setting for one CPU does not affect C-state usage for other CPUs.

Cpupower is the Linux utility doing various power management works. It has two subcommands for C-state management.

The cpupower idle-info subcommand retrieves and outputs CPUIdle subsystem information from sysfs. It is a handy way to dump CPUIdle subsystem information in one-shot.

```
# cpupower idle-info
CPUidle driver: intel_idle
CPUidle governor: menu
analyzing CPU 0:

Number of idle states: 4
Available idle states: POLL C1 C1E C6
POLL:
Flags/Description: CPUIDLE CORE POLL IDLE
Latency: 0
Usage: 39
Duration: 2036
C1:
Flags/Description: MWAIT 0x00
Latency: 1
Usage: 401
Duration: 176878
C1E:
Flags/Description: MWAIT 0x01
Latency: 4
Usage: 12648
Duration: 1373816
C6:
Flags/Description: MWAIT 0x20
Latency: 170
Usage: 18540
Duration: 112338563
```

The `cpupower idle-set` subcommand can be used to disable/enable C-states. Since the subcommand can handle multiple CPUs in batch, it frees us from the burden to write disable sysfs files one by one.

Disable the n-th C-state of the CPUs in cpulist:

```
cpupower idle-set -c cpulist -d n
```

Enable the n-th C-state of the CPUs in cpulist:

```
cpupower idle-set -c cpulist -e n
```

Enable all C-states of the CPUs in cpulist:

```
cpupower idle-set -c cpulist -E
```

Note that if no cpulist is provided with option `-c`, all CPUs are set. And the C-state number in the subcommand is the ordinal number of the C-state in sysfs. That is, the following command will disable /sys/devices/system/cpu/cpu0/cpuidle/state2:

```
cpupower idle-set -c 0 -d 2
```

Kernel Command Line Parameters

Instead of normal C-state control, Linux allows the user to configure CPUIdle subsystem level settings with kernel command line parameters.

There are four kernel parameters controlling availability of different parts of the idle subsystem. In the table below, "•" means a kernel parameter or module is enabled, "◦"means it is not, and "-" means it does not matter.

intel_idle. max_cstate=0	processor. max_cstate=0	cpuidle. off=1	idle	intel_idle	acpi_idle	idle core	idle thread
•	•	○	○	○	•	•	C0,C1
•	○	○	○	○	•	•	C0,C1,C2
-	-	•	○	○	○	○	C0,C1
-	-	-	poll	○	○	-	C0
-	-	-	halt	○	○	-	C0,C1
-	•	○	nomwait	○	•	•	C0,C1
	○						C0,C1,C2

Figure 4. CPUIdle subsystem parameter in Linux kernel

intel_idle.max_cstate=n restricts the C-states supported by intel_idle to C0, C1, ..., Cn. If it is set to 0, the intel_idle driver is disabled.

processor.max_cstate is similar to intel_idle.max_cstate, but it is for the acpi_idle driver. Setting processor.max_cstate to 0 does not disable acpi_idle; instead, acpi_idle treats it as processor.max_cstate=1.

cpuidle.off=1 disables the CPUIdle subsystem. The disabled CPUIdle core does not accept new registrations from governors and drivers. Hence this effectively disables all governors and drivers.

However, none of the above kernel parameters prevent system entering C1 state. This is because if CPUIdle functionality is not available, the idle task tries the default CPU idle routine, which is specific to CPU architecture. For x86, the default CPU idle routine is to execute MWAIT or HLT instruction, and this will make the CPU enter C1. Because MWAIT is more efficient than HLT, the default CPU idle routine prefers MWAIT over HLT, and HLT is used only when MWAIT is not available. And the default CPU idle routine also supports MWAIT for AMD processors since Linux v6.0.

Thus, to disable C-states totally, the default CPU idle routine needs to be disabled as well. This can be done by kernel parameter idle=…. It is specific to X86 platforms and accepts three parameters.

- **idle=poll:** disables CPUIdle drivers and the default CPU idle routine. The idle task is forced to run a polling idle loop.
- **idle=halt:** disables CPUIdle drivers, and HALT is forced to be used for CPU idle loop.
- **idle=nowait:** disables MWAIT for CPU C-state, and hence disables the intel_idle driver.

Therefore, "idle=poll" alone disables all C-states on Linux.

Authors

Peng Liu is an experienced Linux Engineer at the Lenovo Infrastructure Solutions Group, based in Beijing, China. He focuses on storage device drivers and but his interest areas include topics such as I/O frameworks and memory management.

Xiaochun Li is a Linux engineer in the Lenovo Infrastructure Solutions Group based in Beijing, China. He specializes in development related to Linux kernel storage and memory management, as well as kernel DRM. Before joining Lenovo, he was an operating system engineer for INSPUR. With eight years of industry experience, he now focuses on Linux kernel RAS, storage, security and virtualization.

Thanks to Adrian Huang from Lenovo ISG in Taipei for his contributions to the development of this paper.

Related product families

Product families related to this document are the following:

- Processors

Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service. Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
8001 Development Drive
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION ”AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary. Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk. Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

This document, LP1945, was created or updated on April 25, 2024.

Send us your comments in one of the following ways:

- Use the online Contact us review form found at:
  https://lenovopress.lenovo.com/LP1945
- Send your comments in an e-mail to:
  comments@lenovopress.com

This document is available online at https://lenovopress.lenovo.com/LP1945.

Trademarks

Lenovo and the Lenovo logo are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. A current list of Lenovo trademarks is available on the Web at https://www.lenovo.com/us/en/legal/copytrade/.

The following terms are trademarks of Lenovo in the United States, other countries, or both:
Lenovo®
ThinkSystem®

The following terms are trademarks of other companies:

Intel®, Intel Core™, and Xeon® are trademarks of Intel Corporation or its subsidiaries.

Linux® is the trademark of Linus Torvalds in the U.S. and other countries.

Other company, product, or service names may be trademarks or service marks of others.