Cluster-wide Power-up/power-down race avoidance algorithm

This file documents the algorithm which is used to coordinate CPU and cluster setup and teardown operations and to manage hardware coherency controls safely.

The section "Rationale" explains what the algorithm is for and why it is needed. "Basic model" explains general concepts using a simplified view of the system. The other sections explain the actual details of the algorithm in use.

Rationale

In a system containing multiple CPUs, it is desirable to have the ability to turn off individual CPUs when the system is idle, reducing power consumption and thermal dissipation.

In a system containing multiple clusters of CPUs, it is also desirable to have the ability to turn off entire clusters.

Turning entire clusters off and on is a risky business, because it involves performing potentially destructive operations affecting a group of independently running CPUs, while the OS continues to run. This means that we need some coordination in order to ensure that critical cluster-level operations are only performed when it is truly safe to do so.

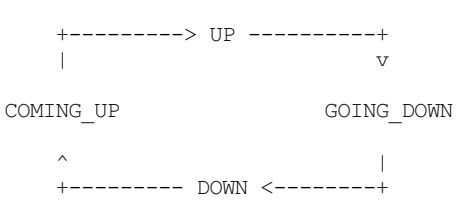
Simple locking may not be sufficient to solve this problem, because mechanisms like Linux spinlocks may rely on coherency mechanisms which are not immediately enabled when a cluster powers up. Since enabling or disabling those mechanisms may itself be a non-atomic operation (such as writing some hardware registers and invalidating large caches), other methods of coordination are required in order to guarantee safe power-down and power-up at the cluster level.

The mechanism presented in this document describes a coherent memory based protocol for performing the needed coordination. It aims to be as lightweight as possible, while providing the required safety properties.

Basic model

Each cluster and CPU is assigned a state, as follows:

- DOWN
- COMING_UP
- UP
- GOING_DOWN



DOWN:
: The CPU or cluster is not coherent, and is either powered off or suspended, or is ready to be powered off or suspended.

COMING_UP:
: The CPU or cluster has committed to moving to the UP state. It may be part way through the process of initialisation and enabling coherency.

UP:
: The CPU or cluster is active and coherent at the hardware level. A CPU in this state is not necessarily being used actively by the kernel.

GOING_DOWN:
: The CPU or cluster has committed to moving to the DOWN state. It may be part way through the process of teardown and coherency exit.

Each CPU has one of these states assigned to it at any point in time. The CPU states are described in the "CPU state" section, below.

Each cluster is also assigned a state, but it is necessary to split the state value into two parts (the "cluster" state and "inbound" state) and to introduce additional states in order to avoid races between different CPUs in the cluster simultaneously modifying the state. The cluster- level states are described in the "Cluster state" section.

To help distinguish the CPU states from cluster states in this discussion, the state names are given a *CPU_* prefix for the CPU states, and a *CLUSTER_* or *INBOUND_* prefix for the cluster states.

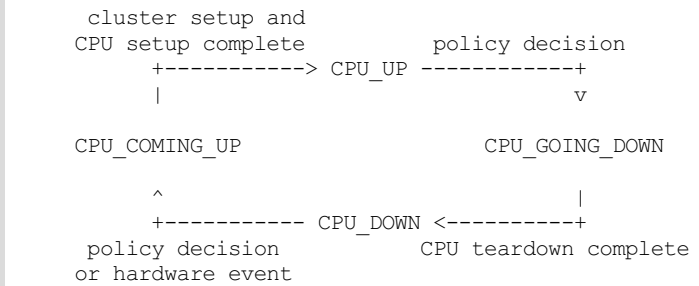
CPU state

In this algorithm, each individual core in a multi-core processor is referred to as a "CPU". CPUs are assumed to be single-threaded: therefore, a CPU can only be doing one thing at a single point in time.

This means that CPUs fit the basic model closely.

The algorithm defines the following states for each CPU in the system:

- CPU_DOWN
- CPU_COMING_UP
- CPU_UP
- CPU_GOING_DOWN



The definitions of the four states correspond closely to the states of the basic model.

Transitions between states occur as follows.

A trigger event (spontaneous) means that the CPU can transition to the next state as a result of making local progress only, with no requirement for any external event to happen.

CPU_DOWN:

A CPU reaches the CPU_DOWN state when it is ready for power-down. On reaching this state, the CPU will typically power itself down or suspend itself, via a WFI instruction or a firmware call.

Next state:
CPU_COMING_UP
Conditions:
none
Trigger events:
a. an explicit hardware power-up operation, resulting from a policy decision on another CPU;
b. a hardware event, such as an interrupt.

CPU_COMING_UP:

A CPU cannot start participating in hardware coherency until the cluster is set up and coherent. If the cluster is not ready, then the CPU will wait in the CPU_COMING_UP state until the cluster has been set up.

Next state:
CPU_UP
Conditions:
The CPU's parent cluster must be in CLUSTER_UP.
Trigger events:
Transition of the parent cluster to CLUSTER_UP.

Refer to the "Cluster state" section for a description of the CLUSTER_UP state.

CPU_UP:

When a CPU reaches the CPU_UP state, it is safe for the CPU to start participating in local coherency.

This is done by jumping to the kernel's CPU resume code.

Note that the definition of this state is slightly different from the basic model definition: CPU_UP does not mean that the CPU is coherent yet, but it does mean that it is safe to resume the kernel. The kernel handles the rest of the resume procedure, so the remaining steps are not visible as part of the race avoidance algorithm.

The CPU remains in this state until an explicit policy decision is made to shut down or suspend the CPU.

Next state:
CPU_GOING_DOWN
Conditions:
none
Trigger events:
explicit policy decision

CPU_GOING_DOWN:

While in this state, the CPU exits coherency, including any operations required to achieve this (such as cleaning data caches).

Next state:
CPU_DOWN
Conditions:
local CPU teardown complete
Trigger events:
(spontaneous)

Cluster state

A cluster is a group of connected CPUs with some common resources. Because a cluster contains multiple CPUs, it can be doing multiple things at the same time. This has some implications. In particular, a CPU can start up while another CPU is tearing the cluster down.

In this discussion, the "outbound side" is the view of the cluster state as seen by a CPU tearing the cluster down. The "inbound side" is the view of the cluster state as seen by a CPU setting the CPU up.

In order to enable safe coordination in such situations, it is important that a CPU which is setting up the cluster can advertise its state independently of the CPU which is tearing down the cluster. For this reason, the cluster state is split into two parts:

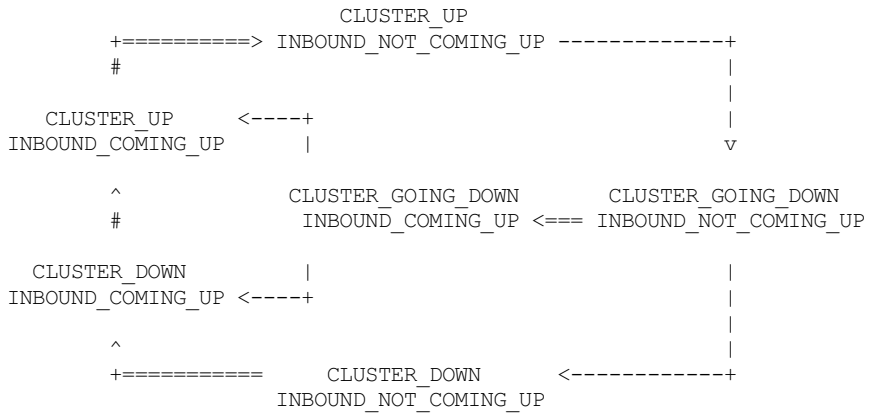
"cluster" state: The global state of the cluster; or the state on the outbound side:

- CLUSTER_DOWN
- CLUSTER_UP
- CLUSTER_GOING_DOWN

"inbound" state: The state of the cluster on the inbound side.

- INBOUND_NOT_COMING_UP
- INBOUND_COMING_UP

The different pairings of these states results in six possible states for the cluster as a whole:



Transitions -----> can only be made by the outbound CPU, and only involve changes to the "cluster" state.

Transitions ====##> can only be made by the inbound CPU, and only involve changes to the "inbound" state, except where there is no further transition possible on the outbound side (i.e., the outbound CPU has put the cluster into the CLUSTER_DOWN state).

The race avoidance algorithm does not provide a way to determine which exact CPUs within the cluster play these roles. This must be decided in advance by some other means. Refer to the section "Last man and first man selection" for more explanation.

CLUSTER_DOWN/INBOUND_NOT_COMING_UP is the only state where the cluster can actually be powered down.

The parallelism of the inbound and outbound CPUs is observed by the existence of two different paths from CLUSTER_GOING_DOWN/ INBOUND_NOT_COMING_UP (corresponding to GOING_DOWN in the basic model) to CLUSTER_DOWN/INBOUND_COMING_UP (corresponding to COMING_UP in the basic model). The second path avoids cluster teardown completely.

CLUSTER_UP/INBOUND_COMING_UP is equivalent to UP in the basic model. The final transition to CLUSTER_UP/INBOUND_NOT_COMING_UP is trivial and merely resets the state machine ready for the next cycle.

Details of the allowable transitions follow.

The next state in each case is notated

<cluster state>/<inbound state> (<transitioner>)

where the <transitioner> is the side on which the transition can occur; either the inbound or the outbound side.

CLUSTER_DOWN/INBOUND_NOT_COMING_UP:

Next state:
CLUSTER_DOWN/INBOUND_COMING_UP (inbound)
Conditions:
none
Trigger events:
a. an explicit hardware power-up operation, resulting from a policy decision on another CPU;
b. a hardware event, such as an interrupt.

CLUSTER_DOWN/INBOUND_COMING_UP:

In this state, an inbound CPU sets up the cluster, including enabling of hardware coherency at the cluster level and any other operations (such as cache invalidation) which are required in order to achieve this.

The purpose of this state is to do sufficient cluster-level setup to enable other CPUs in the cluster to enter coherency safely.

Next state:
CLUSTER_UP/INBOUND_COMING_UP (inbound)
Conditions:
cluster-level setup and hardware coherency complete
Trigger events:
(spontaneous)

CLUSTER_UP/INBOUND_COMING_UP:

Cluster-level setup is complete and hardware coherency is enabled for the cluster. Other CPUs in the cluster can safely enter coherency.

This is a transient state, leading immediately to CLUSTER_UP/INBOUND_NOT_COMING_UP. All other CPUs on the cluster should consider treat these two states as equivalent.

Next state:
CLUSTER_UP/INBOUND_NOT_COMING_UP (inbound)
Conditions:
none
Trigger events:
(spontaneous)

CLUSTER_UP/INBOUND_NOT_COMING_UP:

Cluster-level setup is complete and hardware coherency is enabled for the cluster. Other CPUs in the cluster can safely enter coherency.

The cluster will remain in this state until a policy decision is made to power the cluster down.

Next state:
CLUSTER_GOING_DOWN/INBOUND_NOT_COMING_UP (outbound)
Conditions:
none
Trigger events:
policy decision to power down the cluster

CLUSTER_GOING_DOWN/INBOUND_NOT_COMING_UP:

An outbound CPU is tearing the cluster down. The selected CPU must wait in this state until all CPUs in the cluster are in the CPU_DOWN state.

When all CPUs are in the CPU_DOWN state, the cluster can be torn down, for example by cleaning data caches and exiting cluster-level coherency.

To avoid wasteful unnecessary teardown operations, the outbound should check the inbound cluster state for asynchronous transitions to INBOUND_COMING_UP. Alternatively, individual CPUs can be checked for entry into CPU_COMING_UP or CPU_UP.

Next states:

CLUSTER_DOWN/INBOUND_NOT_COMING_UP (outbound)

Conditions:

cluster torn down and ready to power off

Trigger events:

(spontaneous)

CLUSTER_GOING_DOWN/INBOUND_COMING_UP (inbound)

Conditions:

none

Trigger events:

a. an explicit hardware power-up operation, resulting from a policy decision on another CPU;
b. a hardware event, such as an interrupt.

CLUSTER_GOING_DOWN/INBOUND_COMING_UP:

The cluster is (or was) being torn down, but another CPU has come online in the meantime and is trying to set up the cluster again.

If the outbound CPU observes this state, it has two choices:

a. back out of teardown, restoring the cluster to the CLUSTER_UP state;
b. finish tearing the cluster down and put the cluster in the CLUSTER_DOWN state; the inbound CPU will set up the cluster again from there.

Choice (a) permits the removal of some latency by avoiding unnecessary teardown and setup operations in situations where the cluster is not really going to be powered down.

Next states:

CLUSTER_UP/INBOUND_COMING_UP (outbound)

Conditions:

cluster-level setup and hardware coherency complete

Trigger events:

(spontaneous)

CLUSTER_DOWN/INBOUND_COMING_UP (outbound)

Conditions:

cluster torn down and ready to power off

Trigger events:

(spontaneous)

Last man and First man selection

The CPU which performs cluster tear-down operations on the outbound side is commonly referred to as the "last man".

The CPU which performs cluster setup on the inbound side is commonly referred to as the "first man".

The race avoidance algorithm documented above does not provide a mechanism to choose which CPUs should play these roles.

Last man:

When shutting down the cluster, all the CPUs involved are initially executing Linux and hence coherent. Therefore, ordinary spinlocks can be used to select a last man safely, before the CPUs become non-coherent.

First man:

Because CPUs may power up asynchronously in response to external wake-up events, a dynamic mechanism is needed to make sure that only one CPU attempts to play the first man role and do the cluster-level initialisation: any other CPUs must wait for this to complete before proceeding.

Cluster-level initialisation may involve actions such as configuring coherency controls in the bus fabric.

The current implementation in mcpm_head.S uses a separate mutual exclusion mechanism to do this arbitration. This mechanism is documented in detail in vlocks.txt.

Features and Limitations

Implementation:

The current ARM-based implementation is split between arch/arm/common/mcpm_head.S (low-level inbound CPU operations) and arch/arm/common/mcpm_entry.c (everything else):

__mcpm_cpu_going_down() signals the transition of a CPU to the CPU_GOING_DOWN state.

__mcpm_cpu_down() signals the transition of a CPU to the CPU_DOWN state.

A CPU transitions to CPU_COMING_UP and then to CPU_UP via the low-level power-up code in mcpm_head.S. This could involve CPU-specific setup code, but in the current implementation it does not.

__mcpm_outbound_enter_critical() and __mcpm_outbound_leave_critical() handle transitions from CLUSTER_UP to CLUSTER_GOING_DOWN and from there to CLUSTER_DOWN or back to CLUSTER_UP (in the case of an aborted cluster power-down).

These functions are more complex than the __mcpm_cpu_*() functions due to the extra inter-CPU coordination which is needed for safe transitions at the cluster level.

A cluster transitions from CLUSTER_DOWN back to CLUSTER_UP via the low-level power-up code in mcpm_head.S. This typically involves platform-specific setup code, provided by the platform-specific power_up_setup function registered via mcpm_sync_init.

Deep topologies:

As currently described and implemented, the algorithm does not support CPU topologies involving more than two levels (i.e., clusters of clusters are not supported). The algorithm could be extended by replicating the cluster-level states for the additional topological levels, and modifying the transition rules for the intermediate (non-outermost) cluster levels.

Colophon

Originally created and documented by Dave Martin for Linaro Limited, in collaboration with Nicolas Pitre and Achin Gupta.