Interaction of Suspend code (S3) with the CPU hotplug infrastructure

C. 2011 - 2014 Srivatsa S. Bhat <srivatsa.bhat@linux.vnet.ibm.com>

I. Differences between CPU hotplug and Suspend-to-RAM

How does the regular CPU hotplug code differ from how the Suspend-to-RAM infrastructure uses it internally? And where do they share common code?

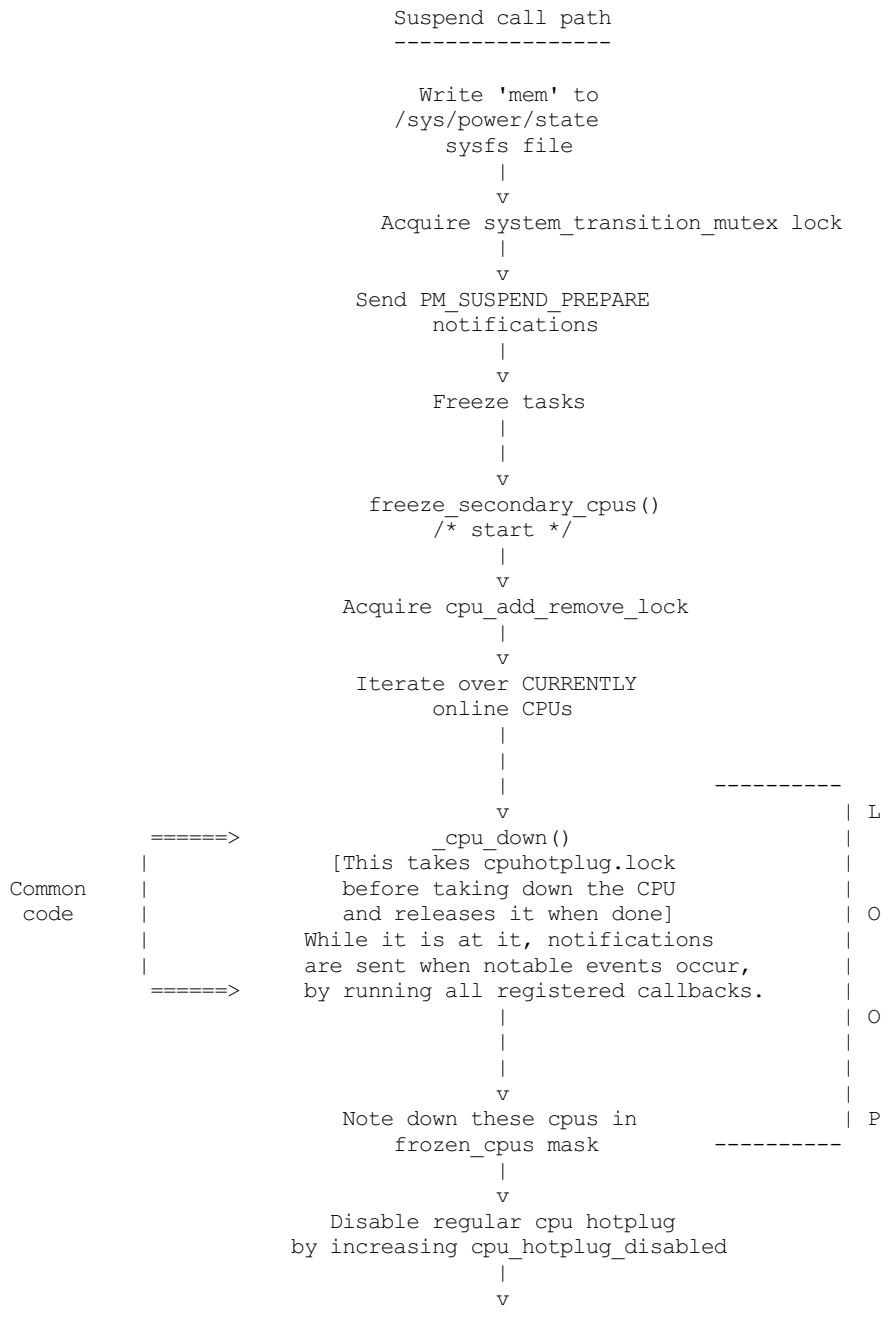
Well, a picture is worth a thousand words... So ASCII art follows :-)

[This depicts the current design in the kernel, and focusses only on the interactions involving the freezer and CPU hotplug and also tries to explain the locking involved. It outlines the notifications involved as well. But please note that here, only the call paths are illustrated, with the aim of describing where they take different paths and where they share code. What happens when regular CPU hotplug and Suspend-to-RAM race with each other is not depicted here.]

On a high level, the suspend-resume cycle goes like this:

```
|Freeze| -> |Disable nonboot| -> |Do suspend| -> |Enable nonboot| -> |Thaw |
|tasks |    |     cpus      |    |          |    |     cpus     |    |tasks|
```

More details follow:



```
                        Release cpu_add_remove_lock
                                    |
                                    v
                        /* freeze_secondary_cpus() complete */
                                    |
                                    v
                                 Do suspend
```

Resuming back is likewise, with the counterparts being (in the order of execution during resume):

- thaw_secondary_cpus() which involves:

```
|  Acquire cpu_add_remove_lock
|  Decrease cpu_hotplug_disabled, thereby enabling regular cpu hotplug
|  Call _cpu_up() [for all those cpus in the frozen_cpus mask, in a loop]
|  Release cpu_add_remove_lock
v
```

- thaw tasks
- send PM_POST_SUSPEND notifications
- Release system_transition_mutex lock.

It is to be noted here that the system_transition_mutex lock is acquired at the very beginning, when we are just starting out to suspend, and then released only after the entire cycle is complete (i.e., suspend + resume).

```
                          Regular CPU hotplug call path
                          -----------------------------

                                Write 0 (or 1) to
                        /sys/devices/system/cpu/cpu*/online
                                    sysfs file
                                        |
                                        |
                                        v
                                    cpu_down()
                                        |
                                        v
                             Acquire cpu_add_remove_lock
                                        |
                                        v
                          If cpu_hotplug_disabled > 0
                                return gracefully
                                        |
                                        |
                                        v
           ======>                  _cpu_down()
          |                       [This takes cpuhotplug.lock
Common    |                        before taking down the CPU
 code     |                        and releases it when done]
          |                     While it is at it, notifications
          |                    are sent when notable events occur,
           ======>             by running all registered callbacks.
                                        |
                                        |
                                        v
                          Release cpu_add_remove_lock
                                [That's it!, for
                               regular CPU hotplug]
```

So, as can be seen from the two diagrams (the parts marked as "Common code"), regular CPU hotplug and the suspend code path converge at the _cpu_down() and _cpu_up() functions. They differ in the arguments passed to these functions, in that during regular CPU hotplug, 0 is passed for the 'tasks_frozen' argument. But during suspend, since the tasks are already frozen by the time the non-boot CPUs are offlined or onlined, the _cpu_*() functions are called with the 'tasks_frozen' argument set to 1. [See below for some known issues regarding this.]

Important files and functions/entry points:

- kernel/power/process.c : freeze_processes(), thaw_processes()
- kernel/power/suspend.c : suspend_prepare(), suspend_enter(), suspend_finish()
- kernel/cpu.c: cpu_[up|down](), _cpu_[up|down](), [disable|enable]_nonboot_cpus()

II. What are the issues involved in CPU hotplug?

There are some interesting situations involving CPU hotplug and microcode update on the CPUs, as discussed below:

[Please bear in mind that the kernel requests the microcode images from userspace, using the request_firmware() function defined in drivers/base/firmware_loader/main.c]

a. When all the CPUs are identical:

   This is the most common situation and it is quite straightforward: we want to apply the same microcode revision to each of the CPUs. To give an example of x86, the collect_cpu_info() function defined in arch/x86/kernel/microcode_core.c helps in discovering the type of the CPU and thereby in applying the correct microcode revision to it. But note that the kernel does not maintain a common microcode image for the all CPUs, in order to handle case 'b' described below.

b. When some of the CPUs are different than the rest:

   In this case since we probably need to apply different microcode revisions to different CPUs, the kernel maintains a copy of the correct microcode image for each CPU (after appropriate CPU type/model discovery using functions such as collect_cpu_info()).

c. When a CPU is physically hot-unplugged and a new (and possibly different type of) CPU is hot-plugged into the system:

   In the current design of the kernel, whenever a CPU is taken offline during a regular CPU hotplug operation, upon receiving the CPU_DEAD notification (which is sent by the CPU hotplug code), the microcode update driver's callback for that event reacts by freeing the kernel's copy of the microcode image for that CPU.

   Hence, when a new CPU is brought online, since the kernel finds that it doesn't have the microcode image, it does the CPU type/model discovery afresh and then requests the userspace for the appropriate microcode image for that CPU, which is subsequently applied.

   For example, in x86, the mc_cpu_callback() function (which is the microcode update driver's callback registered for CPU hotplug events) calls microcode_update_cpu() which would call microcode_init_cpu() in this case, instead of microcode_resume_cpu() when it finds that the kernel doesn't have a valid microcode image. This ensures that the CPU type/model discovery is performed and the right microcode is applied to the CPU after getting it from userspace.

d. Handling microcode update during suspend/hibernate:

   Strictly speaking, during a CPU hotplug operation which does not involve physically removing or inserting CPUs, the CPUs are not actually powered off during a CPU offline. They are just put to the lowest C-states possible. Hence, in such a case, it is not really necessary to re-apply microcode when the CPUs are brought back online, since they wouldn't have lost the image during the CPU offline operation.

   This is the usual scenario encountered during a resume after a suspend. However, in the case of hibernation, since all the CPUs are completely powered off, during restore it becomes necessary to apply the microcode images to all the CPUs.

   [Note that we don't expect someone to physically pull out nodes and insert nodes with a different type of CPUs in-between a suspend-resume or a hibernate/restore cycle.]

   In the current design of the kernel however, during a CPU offline operation as part of the suspend/hibernate cycle (cpuhp_tasks_frozen is set), the existing copy of microcode image in the kernel is not freed up. And during the CPU online operations (during resume/restore), since the kernel finds that it already has copies of the microcode images for all the CPUs, it just applies them to the CPUs, avoiding any re-discovery of CPU type/model and the need for validating whether the microcode revisions are right for the CPUs or not (due to the above assumption that physical CPU hotplug will not be done in-between suspend/resume or hibernate/restore cycles).

III. Known problems

Are there any known problems when regular CPU hotplug and suspend race with each other?

Yes, they are listed below:

1. When invoking regular CPU hotplug, the 'tasks_frozen' argument passed to the _cpu_down() and _cpu_up() functions is *always* 0. This might not reflect the true current state of the system, since the tasks could have been frozen by an out-of-band event such as a suspend operation in progress. Hence, the cpuhp_tasks_frozen variable will not reflect the frozen state and the CPU hotplug callbacks which evaluate that variable might execute the wrong code path.
2. If a regular CPU hotplug stress test happens to race with the freezer due to a suspend operation in progress at the same time, then we could hit the situation described below:

   - A regular cpu online operation continues its journey from userspace into the kernel, since the freezing has not yet begun.
   - Then freezer gets to work and freezes userspace.
   - If cpu online has not yet completed the microcode update stuff by now, it will now start waiting on the frozen userspace in the TASK_UNINTERRUPTIBLE state, in order to get the microcode image.
   - Now the freezer continues and tries to freeze the remaining tasks. But due to this wait mentioned above, the freezer won't be able to freeze the cpu online hotplug task and hence freezing of tasks fails.

   As a result of this task freezing failure, the suspend operation gets aborted.