Freezing of tasks

I. What is the freezing of tasks?

The freezing of tasks is a mechanism by which user space processes and some kernel threads are controlled during hibernation or system-wide suspend (on some architectures).

II. How does it work?

There are three per-task flags used for that, PF_NOFREEZE, PF_FROZEN and PF_FREEZER_SKIP (the last one is auxiliary). The tasks that have PF_NOFREEZE unset (all user space processes and some kernel threads) are regarded as 'freezable' and treated in a special way before the system enters a suspend state as well as before a hibernation image is created (in what follows we only consider hibernation, but the description also applies to suspend).

Namely, as the first step of the hibernation procedure the function freeze_processes() (defined in kernel/power/process.c) is called. A system-wide variable system_freezing_cnt (as opposed to a per-task flag) is used to indicate whether the system is to undergo a freezing operation. And freeze_processes() sets this variable. After this, it executes try_to_freeze_tasks() that sends a fake signal to all user space processes, and wakes up all the kernel threads. All freezable tasks must react to that by calling try_to_freeze(), which results in a call to __refrigerator() (defined in kernel/freezer.c), which sets the task's PF_FROZEN flag, changes its state to TASK_UNINTERRUPTIBLE and makes it loop until PF_FROZEN is cleared for it. Then, we say that the task is 'frozen' and therefore the set of functions handling this mechanism is referred to as 'the freezer' (these functions are defined in kernel/power/process.c, kernel/freezer.c & include/linux/freezer.h). User space processes are generally frozen before kernel threads.

__refrigerator() must not be called directly. Instead, use the try_to_freeze() function (defined in include/linux/freezer.h), that checks if the task is to be frozen and makes the task enter __refrigerator().

For user space processes try_to_freeze() is called automatically from the signal-handling code, but the freezable kernel threads need to call it explicitly in suitable places or use the wait_event_freezable() or wait_event_freezable_timeout() macros (defined in include/linux/freezer.h) that combine interruptible sleep with checking if the task is to be frozen and calling try_to_freeze(). The main loop of a freezable kernel thread may look like the following one:

```
set_freezable();
do {
        hub_events();
        wait_event_freezable(khubd_wait,
                        !list_empty(&hub_event_list) ||
                        kthread_should_stop());
} while (!kthread_should_stop() || !list_empty(&hub_event_list));
```

(from drivers/usb/core/hub.c::hub_thread()).

If a freezable kernel thread fails to call try_to_freeze() after the freezer has initiated a freezing operation, the freezing of tasks will fail and the entire hibernation operation will be cancelled. For this reason, freezable kernel threads must call try_to_freeze() somewhere or use one of the wait_event_freezable() and wait_event_freezable_timeout() macros.

After the system memory state has been restored from a hibernation image and devices have been reinitialized, the function thaw_processes() is called in order to clear the PF_FROZEN flag for each frozen task. Then, the tasks that have been frozen leave __refrigerator() and continue running.

Rationale behind the functions dealing with freezing and thawing of tasks

freeze_processes():
- freezes only userspace tasks

freeze_kernel_threads():
- freezes all tasks (including kernel threads) because we can't freeze kernel threads without freezing userspace tasks

thaw_kernel_threads():
- thaws only kernel threads; this is particularly useful if we need to do anything special in between thawing of kernel threads and thawing of userspace tasks, or if we want to postpone the thawing of userspace tasks

thaw_processes():
- thaws all tasks (including kernel threads) because we can't thaw userspace tasks without thawing kernel threads

III. Which kernel threads are freezable?

Kernel threads are not freezable by default. However, a kernel thread may clear PF_NOFREEZE for itself by calling set_freezable() (the resetting of PF_NOFREEZE directly is not allowed). From this point it is regarded as freezable and must call try_to_freeze() in a suitable place.

IV. Why do we do that?

Generally speaking, there is a couple of reasons to use the freezing of tasks:

1. The principal reason is to prevent filesystems from being damaged after hibernation. At the moment we have no simple means of checkpointing filesystems, so if there are any modifications made to filesystem data and/or metadata on disks, we cannot bring them back to the state from before the modifications. At the same time each hibernation image contains some filesystem-related information that must be consistent with the state of the on-disk data and metadata after the system memory state has been restored from the image (otherwise the filesystems will be damaged in a nasty way, usually making them almost impossible to repair). We therefore freeze tasks that might cause the on-disk filesystems' data and metadata to be modified after the hibernation image has been created and before the system is finally powered off. The majority of these are user space processes, but if any of the kernel threads may cause something like this to happen, they have to be freezable.
2. Next, to create the hibernation image we need to free a sufficient amount of memory (approximately 50% of available RAM) and we need to do that before devices are deactivated, because we generally need them for swapping out. Then, after the memory for the image has been freed, we don't want tasks to allocate additional memory and we prevent them from doing that by freezing them earlier. [Of course, this also means that device drivers should not allocate substantial amounts of memory from their .suspend() callbacks before hibernation, but this is a separate issue.]
3. The third reason is to prevent user space processes and some kernel threads from interfering with the suspending and resuming of devices. A user space process running on a second CPU while we are suspending devices may, for example, be troublesome and without the freezing of tasks we would need some safeguards against race conditions that might occur in such a case.

Although Linus Torvalds doesn't like the freezing of tasks, he said this in one of the discussions on LKML (https://lore.kernel.org/r/alpine.LFD.0.98.0704271801020.9964@woody.linux-foundation.org):

"RJW:> Why we freeze tasks at all or why we freeze kernel threads?

Linus: In many ways, 'at all'.

I **do** realize the IO request queue issues, and that we cannot actually do s2ram with some devices in the middle of a DMA. So we want to be able to avoid *that*, there's no question about that. And I suspect that stopping user threads and then waiting for a sync is practically one of the easier ways to do so.

So in practice, the 'at all' may become a 'why freeze kernel threads?' and freezing user threads I don't find really objectionable."

Still, there are kernel threads that may want to be freezable. For example, if a kernel thread that belongs to a device driver accesses the device directly, it in principle needs to know when the device is suspended, so that it doesn't try to access it at that time. However, if the kernel thread is freezable, it will be frozen before the driver's .suspend() callback is executed and it will be thawed after the driver's .resume() callback has run, so it won't be accessing the device while it's suspended.

4. Another reason for freezing tasks is to prevent user space processes from realizing that hibernation (or suspend) operation takes place. Ideally, user space processes should not notice that such a system-wide operation has occurred and should continue running without any problems after the restore (or resume from suspend). Unfortunately, in the most general case this is quite difficult to achieve without the freezing of tasks. Consider, for example, a process that depends on all CPUs being online while it's running. Since we need to disable nonboot CPUs during the hibernation, if this process is not frozen, it may notice that the number of CPUs has changed and may start to work incorrectly because of that.

V. Are there any problems related to the freezing of tasks?

Yes, there are.

First of all, the freezing of kernel threads may be tricky if they depend one on another. For example, if kernel thread A waits for a completion (in the TASK_UNINTERRUPTIBLE state) that needs to be done by freezable kernel thread B and B is frozen in the meantime, then A will be blocked until B is thawed, which may be undesirable. That's why kernel threads are not freezable by default.

Second, there are the following two problems related to the freezing of user space processes:

1. Putting processes into an uninterruptible sleep distorts the load average.
2. Now that we have FUSE, plus the framework for doing device drivers in userspace, it gets even more complicated because some userspace processes are now doing the sorts of things that kernel threads do (https://lists.linux-foundation.org/pipermail/linux-pm/2007-May/012309.html).

The problem 1. seems to be fixable, although it hasn't been fixed so far. The other one is more serious, but it seems that we can work around it by using hibernation (and suspend) notifiers (in that case, though, we won't be able to avoid the realization by the user space processes that the hibernation is taking place).

There are also problems that the freezing of tasks tends to expose, although they are not directly related to it. For example, if request_firmware() is called from a device driver's .resume() routine, it will timeout and eventually fail, because the user land process that should respond to the request is frozen at this point. So, seemingly, the failure is due to the freezing of tasks. Suppose, however, that the firmware file is located on a filesystem accessible only through another device that hasn't been resumed yet. In that case, request_firmware() will fail regardless of whether or not the freezing of tasks is used. Consequently, the problem is not really related

to the freezing of tasks, since it generally exists anyway.

A driver must have all firmwares it may need in RAM before suspend() is called. If keeping them is not practical, for example due to their size, they must be requested early enough using the suspend notifier API described in Documentation/driver-api/pm/notifiers.rst.

VI. Are there any precautions to be taken to prevent freezing failures?

Yes, there are.

First of all, grabbing the 'system_transition_mutex' lock to mutually exclude a piece of code from system-wide sleep such as suspend/hibernation is not encouraged. If possible, that piece of code must instead hook onto the suspend/hibernation notifiers to achieve mutual exclusion. Look at the CPU-Hotplug code (kernel/cpu.c) for an example.

However, if that is not feasible, and grabbing 'system_transition_mutex' is deemed necessary, it is strongly discouraged to directly call mutex_[un]lock(&system_transition_mutex) since that could lead to freezing failures, because if the suspend/hibernate code successfully acquired the 'system_transition_mutex' lock, and hence that other entity failed to acquire the lock, then that task would get blocked in TASK_UNINTERRUPTIBLE state. As a consequence, the freezer would not be able to freeze that task, leading to freezing failure.

However, the [un]lock_system_sleep() APIs are safe to use in this scenario, since they ask the freezer to skip freezing this task, since it is anyway "frozen enough" as it is blocked on 'system_transition_mutex', which will be released only after the entire suspend/hibernation sequence is complete. So, to summarize, use [un]lock_system_sleep() instead of directly using mutex_[un]lock(&system_transition_mutex). That would prevent freezing failures.

V. Miscellaneous

/sys/power/pm_freeze_timeout controls how long it will cost at most to freeze all user space processes or all freezable kernel threads, in unit of millisecond. The default value is 20000, with range of unsigned integer.