Freezing of tasks
        (C) 2007 Rafael J. Wysocki <rjw@sisk.pl>, GPL

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 four per-task flags used for that, PF_NOFREEZE, PF_FROZEN, TIF_FREEZE
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. It executes
try_to_freeze_tasks() that sets TIF_FREEZE for all of the freezable tasks and
either wakes them up, if they are kernel threads, or sends fake signals to them,
if they are user space processes. A task that has TIF_FREEZE set, should react
to it by calling the function called refrigerator() (defined in
kernel/power/process.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 and include/linux/freezer.h). User space
processes are generally frozen before kernel threads.

It is not recommended to call refrigerator() directly. Instead, it is
recommended to use the try_to_freeze() function (defined in
include/linux/freezer.h), that checks the task's TIF_FREEZE flag and makes the
task enter refrigerator() if the flag is set.

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 TIF_FREEZE is set 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
set TIF_FREEZE for it, 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.

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 strongly discouraged). 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 e 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 (http://lkml.org/lkml/2007/4/27/608):

"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 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
notifiers.txt.