Linux's Bedtime Routine

2024-09-08 hibernate

How does Linux move from an awake machine to a hibernating one? How does it then manage to restore all state? These questions led me to read way too much C in trying to figure out how this particular hardware/software boundary is navigated.

This investigation will be split into a few parts, with the first one going from invocation of hibernation to synchronizing all filesystems to disk.

This article has been written using Linux version 6.9.9, the source of which can be found in many places, but can be navigated easily through the Bootlin Elixir Cross-Referencer:

https://elixir.bootlin.com/linux/v6.9.9/source

Each code snippet will begin with a link to the above giving the file path and the line number of the beginning of the snippet.

A Starting Point for Investigation: `/sys/power/state` and `/sys/power/disk`

These two system files exist to allow debugging of hibernation, and thus control the exact state used directly. Writing specific values to the `state` file controls the exact sleep mode used and `disk` controls the specific hibernation mode [1] .

This is extremely handy as an entry point to understand how these systems work, since we can just follow what happens when they are written to.

Show and Store Functions

These two files are defined using the `power_attr` macro:

kernel/power/power.h:80

```
#define power_attr(_name) \
static struct kobj_attribute _name##_attr = {   \
    .attr   = {             \
        .name = __stringify(_name), \
        .mode = 0644,           \
    },                  \
    .show   = _name##_show,         \
    .store  = _name##_store,        \
}
```

`show` is called on reads and `store` on writes.

`state_show` is a little boring for our purposes, as it just prints all the available sleep states.

kernel/power/main.c:657

```
/*
 * state - control system sleep states.
 *
 * show() returns available sleep state labels, which may be "mem", "standby",
 * "freeze" and "disk" (hibernation).
 * See Documentation/admin-guide/pm/sleep-states.rst for a description of
 * what they mean.
 *
 * store() accepts one of those strings, translates it into the proper
 * enumerated value, and initiates a suspend transition.
 */
static ssize_t state_show(struct kobject *kobj, struct kobj_attribute *attr,
                          char *buf)
{
        char *s = buf;
#ifdef CONFIG_SUSPEND
        suspend_state_t i;

        for (i = PM_SUSPEND_MIN; i < PM_SUSPEND_MAX; i++)
                if (pm_states[i])
                        s += sprintf(s,"%s ", pm_states[i]);
```

```
#endif
        if (hibernation_available())
                s += sprintf(s, "disk ");
        if (s != buf)
                /* convert the last space to a newline */
                *(s-1) = '\n';
        return (s - buf);
}
```

`state_store`, however, provides our entry point. If the string “disk” is written to the `state` file, it calls `hibernate()`. This is our entry point.

kernel/power/main.c:715

```
static ssize_t state_store(struct kobject *kobj, struct kobj_attribute *attr,
                           const char *buf, size_t n)
{
        suspend_state_t state;
        int error;

        error = pm_autosleep_lock();
        if (error)
                return error;

        if (pm_autosleep_state() > PM_SUSPEND_ON) {
                error = -EBUSY;
                goto out;
        }

        state = decode_state(buf, n);
        if (state < PM_SUSPEND_MAX) {
                if (state == PM_SUSPEND_MEM)
                        state = mem_sleep_current;

                error = pm_suspend(state);
        } else if (state == PM_SUSPEND_MAX) {
                error = hibernate();
        } else {
                error = -EINVAL;
        }

 out:
```

```
        pm_autosleep_unlock();
        return error ? error : n;
}
```

kernel/power/main.c:688

```
static suspend_state_t decode_state(const char *buf, size_t n)
{
#ifdef CONFIG_SUSPEND
        suspend_state_t state;
#endif
        char *p;
        int len;

        p = memchr(buf, '\n', n);
        len = p ? p - buf : n;

        /* Check hibernation first. */
        if (len == 4 && str_has_prefix(buf, "disk"))
                return PM_SUSPEND_MAX;

#ifdef CONFIG_SUSPEND
        for (state = PM_SUSPEND_MIN; state < PM_SUSPEND_MAX; state++) {
                const char *label = pm_states[state];

                if (label && len == strlen(label) && !strncmp(buf, label, len))
                        return state;
        }
#endif

        return PM_SUSPEND_ON;
}
```

Could we have figured this out just via function names? Sure, but this way we know for sure that nothing else is happening before this function is called.

Autosleep

Our first detour is into the autosleep system. When checking the state above, you may notice that the kernel grabs the `pm_autosleep_lock` before checking the current state.

autosleep is a mechanism originally from Android that sends the entire system to either suspend or hibernate whenever it is not actively working on anything.

This is not enabled for most desktop configurations, since it's primarily for mobile systems and inverts the standard suspend and hibernate interactions.

This system is implemented as a workqueue [2] that checks the current number of wakeup events, processes and drivers that need to run [3], and if there aren't any, then the system is put into the autosleep state, typically suspend. However, it could be hibernate if configured that way via `/sys/power/autosleep` in a similar manner to using `/sys/power/state` to manually enable hibernation.

kernel/power/main.c:841

```
static ssize_t autosleep_store(struct kobject *kobj,
                               struct kobj_attribute *attr,
                               const char *buf, size_t n)
{
        suspend_state_t state = decode_state(buf, n);
        int error;

        if (state == PM_SUSPEND_ON
            && strcmp(buf, "off") && strcmp(buf, "off\n"))
                return -EINVAL;

        if (state == PM_SUSPEND_MEM)
                state = mem_sleep_current;

        error = pm_autosleep_set_state(state);
        return error ? error : n;
}

power_attr(autosleep);
#endif /* CONFIG_PM_AUTOSLEEP */
```

kernel/power/autosleep.c:24

```
static DEFINE_MUTEX(autosleep_lock);
static struct wakeup_source *autosleep_ws;

static void try_to_suspend(struct work_struct *work)
```

```
{
        unsigned int initial_count, final_count;

        if (!pm_get_wakeup_count(&initial_count, true))
                goto out;

        mutex_lock(&autosleep_lock);

        if (!pm_save_wakeup_count(initial_count) ||
                system_state != SYSTEM_RUNNING) {
                mutex_unlock(&autosleep_lock);
                goto out;
        }

        if (autosleep_state == PM_SUSPEND_ON) {
                mutex_unlock(&autosleep_lock);
                return;
        }
        if (autosleep_state >= PM_SUSPEND_MAX)
                hibernate();
        else
                pm_suspend(autosleep_state);

        mutex_unlock(&autosleep_lock);

        if (!pm_get_wakeup_count(&final_count, false))
                goto out;

        /*
         * If the wakeup occurred for an unknown reason, wait to prevent the
         * system from trying to suspend and waking up in a tight loop.
         */
        if (final_count == initial_count)
                schedule_timeout_uninterruptible(HZ / 2);

 out:
        queue_up_suspend_work();
}

static DECLARE_WORK(suspend_work, try_to_suspend);

void queue_up_suspend_work(void)
{
```

```
        if (autosleep_state > PM_SUSPEND_ON)
                queue_work(autosleep_wq, &suspend_work);
}
```

The Steps of Hibernation

Hibernation Kernel Config

It's important to note that most of the hibernate-specific functions below do nothing unless you've defined `CONFIG_HIBERNATION` in your Kconfig [4]. As an example, `hibernate` itself is defined as the following if `CONFIG_HIBERNATE` is not set.

include/linux/suspend.h:407

```
static inline int hibernate(void) { return -ENOSYS; }
```

Check if Hibernation is Available

We begin by confirming that we actually can perform hibernation, via the `hibernation_available` function.

kernel/power/hibernate.c:742

```
if (!hibernation_available()) {
        pm_pr_dbg("Hibernation not available.\n");
        return -EPERM;
}
```

kernel/power/hibernate.c:92

```
bool hibernation_available(void)
{
        return nohibernate == 0 &&
                !security_locked_down(LOCKDOWN_HIBERNATION) &&
                !secretmem_active() && !cxl_mem_active();
}
```

`nohibernate` is controlled by the kernel command line, it's set via either `nohibernate` or `hibernate=no`.

`security_locked_down` is a hook for Linux Security Modules to prevent hibernation. This is used to prevent hibernating to an unencrypted storage device, as specified in the manual page `kernel_lockdown(7)`. Interestingly, either level of lockdown, integrity or confidentiality, locks down hibernation because with the ability to hibernate you can extract bascially anything from memory and even reboot into a modified kernel image.

`secretmem_active` checks whether there is any active use of `memfd_secret`, and if so it prevents hibernation. `memfd_secret` returns a file descriptor that can be mapped into a process but is specifically unmapped from the kernel's memory space. Hibernating with memory that not even the kernel is supposed to access would expose that memory to whoever could access the hibernation image. This particular feature of secret memory was apparently controversial, though not as controversial as performance concerns around fragmentation when unmapping kernel memory (which did not end up being a real problem).

`cxl_mem_active` just checks whether any CXL memory is active. A full explanation is provided in the commit introducing this check but there's also a shortened explanation from `cxl_mem_probe` that sets the relevant flag when initializing a CXL memory device.

drivers/cxl/mem.c:186

```
* The kernel may be operating out of CXL memory on this device,
* there is no spec defined way to determine whether this device
* preserves contents over suspend, and there is no simple way
* to arrange for the suspend image to avoid CXL memory which
* would setup a circular dependency between PCI resume and save
* state restoration.
```

Check Compression

The next check is for whether compression support is enabled, and if so whether the requested algorithm is enabled.

kernel/power/hibernate.c:747

```
/*
 * Query for the compression algorithm support if compression is enabled.
 */
```

```
if (!nocompress) {
        strscpy(hib_comp_algo, hibernate_compressor, sizeof(hib_comp_algo));
        if (crypto_has_comp(hib_comp_algo, 0, 0) != 1) {
                pr_err("%s compression is not available\n", hib_comp_algo);
                return -EOPNOTSUPP;
        }
}
```

The `nocompress` flag is set via the `hibernate` command line parameter, setting `hibernate=nocompress`.

If compression is enabled, then `hibernate_compressor` is copied to `hib_comp_algo`. This synchronizes the current requested compression setting (`hibernate_compressor`) with the current compression setting (`hib_comp_algo`).

Both values are character arrays of size `CRYPTO_MAX_ALG_NAME` (128 in this kernel).

kernel/power/hibernate.c:50

```
static char hibernate_compressor[CRYPTO_MAX_ALG_NAME] = CONFIG_HIBERNATION_DEF_COMP;

/*
 * Compression/decompression algorithm to be used while saving/loading
 * image to/from disk. This would later be used in 'kernel/power/swap.c'
 * to allocate comp streams.
 */
char hib_comp_algo[CRYPTO_MAX_ALG_NAME];
```

`hibernate_compressor` defaults to `lzo` if that algorithm is enabled, otherwise to `lz4` if enabled [5]. It can be overwritten using the `hibernate.compressor` setting to either `lzo` or `lz4`.

kernel/power/Kconfig:95

```
choice
        prompt "Default compressor"
        default HIBERNATION_COMP_LZO
        depends on HIBERNATION

config HIBERNATION_COMP_LZO
        bool "lzo"
```

```
        depends on CRYPTO_LZO

config HIBERNATION_COMP_LZ4
        bool "lz4"
        depends on CRYPTO_LZ4

endchoice

config HIBERNATION_DEF_COMP
        string
        default "lzo" if HIBERNATION_COMP_LZO
        default "lz4" if HIBERNATION_COMP_LZ4
        help
          Default compressor to be used for hibernation.
```

kernel/power/hibernate.c:1425

```
static const char * const comp_alg_enabled[] = {
#if IS_ENABLED(CONFIG_CRYPTO_LZO)
        COMPRESSION_ALGO_LZO,
#endif
#if IS_ENABLED(CONFIG_CRYPTO_LZ4)
        COMPRESSION_ALGO_LZ4,
#endif
};

static int hibernate_compressor_param_set(const char *compressor,
                const struct kernel_param *kp)
{
        unsigned int sleep_flags;
        int index, ret;

        sleep_flags = lock_system_sleep();

        index = sysfs_match_string(comp_alg_enabled, compressor);
        if (index >= 0) {
                ret = param_set_copystring(comp_alg_enabled[index], kp);
                if (!ret)
                        strscpy(hib_comp_algo, comp_alg_enabled[index],
                                sizeof(hib_comp_algo));
        } else {
                ret = index;
        }
```

```
        unlock_system_sleep(sleep_flags);

        if (ret)
                pr_debug("Cannot set specified compressor %s\n",
                         compressor);

        return ret;
}
static const struct kernel_param_ops hibernate_compressor_param_ops = {
        .set    = hibernate_compressor_param_set,
        .get    = param_get_string,
};

static struct kparam_string hibernate_compressor_param_string = {
        .maxlen = sizeof(hibernate_compressor),
        .string = hibernate_compressor,
};
```

We then check whether the requested algorithm is supported via `crypto_has_comp`. If not, we bail out of the whole operation with `EOPNOTSUPP`.

As part of `crypto_has_comp` we perform any needed initialization of the algorithm, loading kernel modules and running initialization code as needed [6].

Grab Locks

The next step is to grab the sleep and hibernation locks via `lock_system_sleep` and `hibernate_acquire`.

kernel/power/hibernate.c:758

```
sleep_flags = lock_system_sleep();
/* The snapshot device should not be opened while we're running */
if (!hibernate_acquire()) {
        error = -EBUSY;
        goto Unlock;
}
```

First, `lock_system_sleep` marks the current thread as not freezable, which will be important later [7]. It then grabs the `system_transistion_mutex`, which locks taking

snapshots or modifying how they are taken, resuming from a hibernation image, entering any suspend state, or rebooting.

The GFP Mask

The kernel also issues a warning if the `gfp` mask is changed via either `pm_restore_gfp_mask` or `pm_restrict_gfp_mask` without holding the `system_transistion_mutex`.

GFP flags tell the kernel how it is permitted to handle a request for memory.

include/linux/gfp_types.h:12

```
* GFP flags are commonly used throughout Linux to indicate how memory
* should be allocated.  The GFP acronym stands for get_free_pages(),
* the underlying memory allocation function.  Not every GFP flag is
* supported by every function which may allocate memory.
```

In the case of hibernation specifically we care about the `IO` and `FS` flags, which are reclaim operators, ways the system is permitted to attempt to free up memory in order to satisfy a specific request for memory.

include/linux/gfp_types.h:176

```
* Reclaim modifiers
* -----------------
* Please note that all the following flags are only applicable to sleepable
* allocations (e.g. %GFP_NOWAIT and %GFP_ATOMIC will ignore them).
*
* %__GFP_IO can start physical IO.
*
* %__GFP_FS can call down to the low-level FS. Clearing the flag avoids the
* allocator recursing into the filesystem which might already be holding
* locks.
```

`gfp_allowed_mask` sets which flags are permitted to be set at the current time.

As the comment below outlines, preventing these flags from being set avoids situations where the kernel needs to do I/O to allocate memory (e.g. read/writing swap [8]) but the devices it needs to read/write to/from are not currently available.

kernel/power/main.c:24

```
/*
 * The following functions are used by the suspend/hibernate code to temporarily
 * change gfp_allowed_mask in order to avoid using I/O during memory allocations
 * while devices are suspended.  To avoid races with the suspend/hibernate code,
 * they should always be called with system_transition_mutex held
 * (gfp_allowed_mask also should only be modified with system_transition_mutex
 * held, unless the suspend/hibernate code is guaranteed not to run in parallel
 * with that modification).
 */
static gfp_t saved_gfp_mask;

void pm_restore_gfp_mask(void)
{
        WARN_ON(!mutex_is_locked(&system_transition_mutex));
        if (saved_gfp_mask) {
                gfp_allowed_mask = saved_gfp_mask;
                saved_gfp_mask = 0;
        }
}

void pm_restrict_gfp_mask(void)
{
        WARN_ON(!mutex_is_locked(&system_transition_mutex));
        WARN_ON(saved_gfp_mask);
        saved_gfp_mask = gfp_allowed_mask;
        gfp_allowed_mask &= ~(__GFP_IO | __GFP_FS);
}
```

Sleep Flags

After grabbing the `system_transition_mutex` the kernel then returns and captures the previous state of the threads flags in `sleep_flags`. This is used later to remove `PF_NOFREEZE` if it wasn't previously set on the current thread.

kernel/power/main.c:52

```
unsigned int lock_system_sleep(void)
{
        unsigned int flags = current->flags;
        current->flags |= PF_NOFREEZE;
```

```
        mutex_lock(&system_transition_mutex);
        return flags;
}
EXPORT_SYMBOL_GPL(lock_system_sleep);
```

include/linux/sched.h:1633

```
#define PF_NOFREEZE             0x00008000      /* This thread should not be frozen *
```

Then we grab the hibernate-specific semaphore to ensure no one can open a snapshot or resume from it while we perform hibernation. Additionally this lock is used to prevent `hibernate_quiet_exec`, which is used by the `nvdimm` driver to active its firmware with all processes and devices frozen, ensuring it is the only thing running at that time [9].

kernel/power/hibernate.c:82

```
bool hibernate_acquire(void)
{
        return atomic_add_unless(&hibernate_atomic, -1, 0);
}
```

Prepare Console

The kernel next calls `pm_prepare_console`. This function only does anything if `CONFIG_VT_CONSOLE_SLEEP` has been set.

This prepares the virtual terminal for a suspend state, switching away to a console used only for the suspend state if needed.

kernel/power/console.c:130

```
void pm_prepare_console(void)
{
        if (!pm_vt_switch())
                return;

        orig_fgconsole = vt_move_to_console(SUSPEND_CONSOLE, 1);
        if (orig_fgconsole < 0)
                return;
```

```
        orig_kmsg = vt_kmsg_redirect(SUSPEND_CONSOLE);
        return;
}
```

The first thing is to check whether we actually need to switch the VT

kernel/power/console.c:94

```
/*
 * There are three cases when a VT switch on suspend/resume are required:
 *   1) no driver has indicated a requirement one way or another, so preserve
 *      the old behavior
 *   2) console suspend is disabled, we want to see debug messages across
 *      suspend/resume
 *   3) any registered driver indicates it needs a VT switch
 *
 * If none of these conditions is present, meaning we have at least one driver
 * that doesn't need the switch, and none that do, we can avoid it to make
 * resume look a little prettier (and suspend too, but that's usually hidden,
 * e.g. when closing the lid on a laptop).
 */
static bool pm_vt_switch(void)
{
        struct pm_vt_switch *entry;
        bool ret = true;

        mutex_lock(&vt_switch_mutex);
        if (list_empty(&pm_vt_switch_list))
                goto out;

        if (!console_suspend_enabled)
                goto out;

        list_for_each_entry(entry, &pm_vt_switch_list, head) {
                if (entry->required)
                        goto out;
        }

        ret = false;
out:
        mutex_unlock(&vt_switch_mutex);
        return ret;
}
```

There is an explanation of the conditions under which a switch is performed in the comment above the function, but we'll also walk through the steps here.

Firstly we grab the `vt_switch_mutex` to ensure nothing will modify the list while we're looking at it.

We then examine the `pm_vt_switch_list`. This list is used to indicate the drivers that require a switch during suspend. They register this requirement, or the lack thereof, via `pm_vt_switch_required`.

kernel/power/console.c:31

```
/**
 * pm_vt_switch_required - indicate VT switch at suspend requirements
 * @dev: device
 * @required: if true, caller needs VT switch at suspend/resume time
 *
 * The different console drivers may or may not require VT switches across
 * suspend/resume, depending on how they handle restoring video state and
 * what may be running.
 *
 * Drivers can indicate support for switchless suspend/resume, which can
 * save time and flicker, by using this routine and passing 'false' as
 * the argument.  If any loaded driver needs VT switching, or the
 * no_console_suspend argument has been passed on the command line, VT
 * switches will occur.
 */
void pm_vt_switch_required(struct device *dev, bool required)
```

Next, we check `console_suspend_enabled`. This is set to false by the kernel parameter `no_console_suspend`, but defaults to true.

Finally, if there are any entries in the `pm_vt_switch_list`, then we check to see if any of them require a VT switch.

Only if none of these conditions apply, then we return false.

If a VT switch is in fact required, then we move first the currently active virtual terminal/console [10] (`vt_move_to_console`) and then the current location of kernel messages (`vt_kmsg_redirect`) to the `SUSPEND_CONSOLE`. The `SUSPEND_CONSOLE` is the

last entry in the list of possible consoles, and appears to just be a black hole to throw away messages.

kernel/power/console.c:16

```
#define SUSPEND_CONSOLE (MAX_NR_CONSOLES-1)
```

Interestingly, these are separate functions because you can use `TIOCL_SETKMSGREDIRECT` (an `ioctl` [11]) to send kernel messages to a specific virtual terminal, but by default its the same as the currently active console.

The locations of the previously active console and the previous kernel messages location are stored in `orig_fgconsole` and `orig_kmsg`, to restore the state of the console and kernel messages after the machine wakes up again. Interestingly, this means `orig_fgconsole` also ends up storing any errors, so has to be checked to ensure it's not less than zero before we try to do anything with the kernel messages on both suspend and resume.

drivers/tty/vt/vt_ioctl.c:1268

```
/* Perform a kernel triggered VT switch for suspend/resume */

static int disable_vt_switch;

int vt_move_to_console(unsigned int vt, int alloc)
{
        int prev;

        console_lock();
        /* Graphics mode - up to X */
        if (disable_vt_switch) {
                console_unlock();
                return 0;
        }
        prev = fg_console;

        if (alloc && vc_allocate(vt)) {
                /* we can't have a free VC for now. Too bad,
                 * we don't want to mess the screen for now. */
                console_unlock();
```

```
                return -ENOSPC;
        }

        if (set_console(vt)) {
                /*
                 * We're unable to switch to the SUSPEND_CONSOLE.
                 * Let the calling function know so it can decide
                 * what to do.
                 */
                console_unlock();
                return -EIO;
        }
        console_unlock();
        if (vt_waitactive(vt + 1)) {
                pr_debug("Suspend: Can't switch VCs.");
                return -EINTR;
        }
        return prev;
}
```

Unlike most other locking functions we've seen so far, `console_lock` needs to be careful to ensure nothing else is panicking and needs to dump to the console before grabbing the semaphore for the console and setting a couple flags.

Panics

Panics are tracked via an atomic integer set to the id of the processor currently panicking.

kernel/printk/printk.c:2649

```
/**
 * console_lock - block the console subsystem from printing
 *
 * Acquires a lock which guarantees that no consoles will
 * be in or enter their write() callback.
 *
 * Can sleep, returns nothing.
 */
void console_lock(void)
{
        might_sleep();
```

```
        /* On panic, the console_lock must be left to the panic cpu. */
        while (other_cpu_in_panic())
                msleep(1000);

        down_console_sem();
        console_locked = 1;
        console_may_schedule = 1;
}
EXPORT_SYMBOL(console_lock);
```

kernel/printk/printk.c:362

```
/*
 * Return true if a panic is in progress on a remote CPU.
 *
 * On true, the local CPU should immediately release any printing resources
 * that may be needed by the panic CPU.
 */
bool other_cpu_in_panic(void)
{
        return (panic_in_progress() && !this_cpu_in_panic());
}
```

kernel/printk/printk.c:345

```
static bool panic_in_progress(void)
{
        return unlikely(atomic_read(&panic_cpu) != PANIC_CPU_INVALID);
}
```

kernel/printk/printk.c:350

```
/* Return true if a panic is in progress on the current CPU. */
bool this_cpu_in_panic(void)
{
        /*
         * We can use raw_smp_processor_id() here because it is impossible for
         * the task to be migrated to the panic_cpu, or away from it. If
         * panic_cpu has already been set, and we're not currently executing on
         * that CPU, then we never will be.
         */
```

```
        return unlikely(atomic_read(&panic_cpu) == raw_smp_processor_id());
}
```

`console_locked` is a debug value, used to indicate that the lock should be held, and our first indication that this whole virtual terminal system is more complex than might initially be expected.

kernel/printk/printk.c:373

```
/*
 * This is used for debugging the mess that is the VT code by
 * keeping track if we have the console semaphore held. It's
 * definitely not the perfect debug tool (we don't know if _WE_
 * hold it and are racing, but it helps tracking those weird code
 * paths in the console code where we end up in places I want
 * locked without the console semaphore held).
 */
static int console_locked;
```

`console_may_schedule` is used to see if we are permitted to sleep and schedule other work while we hold this lock. As we'll see later, the virtual terminal subsystem is not re-entrant, so there's all sorts of hacks in here to ensure we don't leave important code sections that can't be safely resumed.

Disable VT Switch

As the comment below lays out, when another program is handling graphical display anyway, there's no need to do any of this, so the kernel provides a switch to turn the whole thing off. Interestingly, this appears to only be used by three drivers, so the specific hardware support required must not be particularly common.

```
drivers/gpu/drm/omapdrm/dss
drivers/video/fbdev/geode
drivers/video/fbdev/omap2
```

drivers/tty/vt/vt_ioctl.c:1308

```
/*
 * Normally during a suspend, we allocate a new console and switch to it.
 * When we resume, we switch back to the original console.  This switch
```

```
 * can be slow, so on systems where the framebuffer can handle restoration
 * of video registers anyways, there's little point in doing the console
 * switch.  This function allows you to disable it by passing it '0'.
 */
void pm_set_vt_switch(int do_switch)
{
        console_lock();
        disable_vt_switch = !do_switch;
        console_unlock();
}
EXPORT_SYMBOL(pm_set_vt_switch);
```

The rest of the `vt_switch_console` function is pretty normal, however, simply allocating space if needed to create the requested virtual terminal and then setting the current virtual terminal via `set_console`.

Virtual Terminal Set Console

With `set_console`, we begin (as if we haven't been already) to enter the madness that is the virtual terminal subsystem. As mentioned previously, modifications to its state must be made very carefully, as other stuff happening at the same time could create complete messes.

All this to say, calling `set_console` does not actually perform any work to change the state of the current console. Instead it indicates what changes it wants and then schedules that work.

drivers/tty/vt/vt.c:3153

```
int set_console(int nr)
{
        struct vc_data *vc = vc_cons[fg_console].d;

        if (!vc_cons_allocated(nr) || vt_dont_switch ||
                (vc->vt_mode.mode == VT_AUTO && vc->vc_mode == KD_GRAPHICS)) {

                /*
                 * Console switch will fail in console_callback() or
                 * change_console() so there is no point scheduling
                 * the callback
                 *
```

```
                 * Existing set_console() users don't check the return
                 * value so this shouldn't break anything
                 */
                return -EINVAL;
        }

        want_console = nr;
        schedule_console_callback();

        return 0;
}
```

The check for `vc->vc_mode == KD_GRAPHICS` is where most end-user graphical desktops will bail out of this change, as they're in graphics mode and don't need to switch away to the suspend console.

`vt_dont_switch` is a flag used by the `ioctl`s [11] `VT_LOCKSWITCH` and `VT_UNLOCKSWITCH` to prevent the system from switching virtual terminal devices when the user has explicitly locked it.

`VT_AUTO` is a flag indicating that automatic virtual terminal switching is enabled [12], and thus deliberate switching to a suspend terminal is not required.

However, if you do run your machine from a virtual terminal, then we indicate to the system that we want to change to the requested virtual terminal via the `want_console` variable and schedule a callback via `schedule_console_callback`.

drivers/tty/vt/vt.c:315

```
void schedule_console_callback(void)
{
        schedule_work(&console_work);
}
```

`console_work` is a workqueue [2] that will execute the given task asynchronously.

Console Callback

drivers/tty/vt/vt.c:3109

```
/*
 * This is the console switching callback.
 *
 * Doing console switching in a process context allows
 * us to do the switches asynchronously (needed when we want
 * to switch due to a keyboard interrupt).  Synchronization
 * with other console code and prevention of re-entrancy is
 * ensured with console_lock.
 */
static void console_callback(struct work_struct *ignored)
{
        console_lock();

        if (want_console >= 0) {
                if (want_console != fg_console &&
                    vc_cons_allocated(want_console)) {
                        hide_cursor(vc_cons[fg_console].d);
                        change_console(vc_cons[want_console].d);
                        /* we only changed when the console had already
                           been allocated - a new console is not created
                           in an interrupt routine */
                }
                want_console = -1;
        }
...
```

`console_callback` first looks to see if there is a console change wanted via `want_console` and then changes to it if it's not the current console and has been allocated already. We do first remove any cursor state with `hide_cursor`.

drivers/tty/vt/vt.c:841

```
static void hide_cursor(struct vc_data *vc)
{
        if (vc_is_sel(vc))
                clear_selection();

        vc->vc_sw->con_cursor(vc, false);
        hide_softcursor(vc);
}
```

A full dive into the `tty` driver is a task for another time, but this should give a general sense of how this system interacts with hibernation.

Notify Power Management Call Chain

kernel/power/hibernate.c:767

```
pm_notifier_call_chain_robust(PM_HIBERNATION_PREPARE, PM_POST_HIBERNATION)
```

This will call a chain of power management callbacks, passing first `PM_HIBERNATION_PREPARE` and then `PM_POST_HIBERNATION` on startup or on error with another callback.

kernel/power/main.c:98

```
int pm_notifier_call_chain_robust(unsigned long val_up, unsigned long val_down)
{
        int ret;

        ret = blocking_notifier_call_chain_robust(&pm_chain_head, val_up, val_down, N

        return notifier_to_errno(ret);
}
```

The power management notifier is a blocking notifier chain, which means it has the following properties.

include/linux/notifier.h:23

```
 *      Blocking notifier chains: Chain callbacks run in process context.
 *              Callouts are allowed to block.
```

The callback chain is a linked list with each entry containing a priority and a function to call. The function technically takes in a data value, but it is always `NULL` for the power management chain.

include/linux/notifier.h:49

```
struct notifier_block;
```

```
typedef int (*notifier_fn_t)(struct notifier_block *nb,
                        unsigned long action, void *data);

struct notifier_block {
        notifier_fn_t notifier_call;
        struct notifier_block __rcu *next;
        int priority;
};
```

The head of the linked list is protected by a read-write semaphore.

include/linux/notifier.h:65

```
struct blocking_notifier_head {
        struct rw_semaphore rwsem;
        struct notifier_block __rcu *head;
};
```

Because it is prioritized, appending to the list requires walking it until an item with lower [13] priority is found to insert the current item before.

kernel/notifier.c:252

```
/*
 *      Blocking notifier chain routines.  All access to the chain is
 *      synchronized by an rwsem.
 */

static int __blocking_notifier_chain_register(struct blocking_notifier_head *nh,
                                              struct notifier_block *n,
                                              bool unique_priority)
{
        int ret;

        /*
         * This code gets used during boot-up, when task switching is
         * not yet working and interrupts must remain disabled.  At
         * such times we must not call down_write().
         */
        if (unlikely(system_state == SYSTEM_BOOTING))
                return notifier_chain_register(&nh->head, n, unique_priority);
```

```
        down_write(&nh->rwsem);
        ret = notifier_chain_register(&nh->head, n, unique_priority);
        up_write(&nh->rwsem);
        return ret;
}
```

kernel/notifier.c:20

```
/*
 *      Notifier chain core routines.  The exported routines below
 *      are layered on top of these, with appropriate locking added.
 */

static int notifier_chain_register(struct notifier_block **nl,
                                   struct notifier_block *n,
                                   bool unique_priority)
{
        while ((*nl) != NULL) {
                if (unlikely((*nl) == n)) {
                        WARN(1, "notifier callback %ps already registered",
                             n->notifier_call);
                        return -EEXIST;
                }
                if (n->priority > (*nl)->priority)
                        break;
                if (n->priority == (*nl)->priority && unique_priority)
                        return -EBUSY;
                nl = &((*nl)->next);
        }
        n->next = *nl;
        rcu_assign_pointer(*nl, n);
        trace_notifier_register((void *)n->notifier_call);
        return 0;
}
```

Each callback can return one of a series of options.

include/linux/notifier.h:18

```
#define NOTIFY_DONE             0x0000          /* Don't care */
#define NOTIFY_OK               0x0001          /* Suits me */
#define NOTIFY_STOP_MASK        0x8000          /* Don't call further */
```

```
#define NOTIFY_BAD              (NOTIFY_STOP_MASK|0x0002)
                                                /* Bad/Veto action */
```

When notifying the chain, if a function returns `STOP` or `BAD` then the previous parts of the chain are called again with `PM_POST_HIBERNATION` [14] and an error is returned.

kernel/notifier.c:107

```
/**
 * notifier_call_chain_robust - Inform the registered notifiers about an event
 *                              and rollback on error.
 * @nl:         Pointer to head of the blocking notifier chain
 * @val_up:     Value passed unmodified to the notifier function
 * @val_down:   Value passed unmodified to the notifier function when recovering
 *              from an error on @val_up
 * @v:          Pointer passed unmodified to the notifier function
 *
 * NOTE:        It is important the @nl chain doesn't change between the two
 *              invocations of notifier_call_chain() such that we visit the
 *              exact same notifier callbacks; this rules out any RCU usage.
 *
 * Return:      the return value of the @val_up call.
 */
static int notifier_call_chain_robust(struct notifier_block **nl,
                                      unsigned long val_up, unsigned long val_down,
                                      void *v)
{
        int ret, nr = 0;

        ret = notifier_call_chain(nl, val_up, v, -1, &nr);
        if (ret & NOTIFY_STOP_MASK)
                notifier_call_chain(nl, val_down, v, nr-1, NULL);

        return ret;
}
```

Each of these callbacks tends to be quite driver-specific, so we'll cease discussion of this here.

Sync Filesystems

The next step is to ensure all filesystems have been synchronized to disk.

This is performed via a simple helper function that times how long the full synchronize operation, `ksys_sync` takes.

kernel/power/main.c:69

```
void ksys_sync_helper(void)
{
        ktime_t start;
        long elapsed_msecs;

        start = ktime_get();
        ksys_sync();
        elapsed_msecs = ktime_to_ms(ktime_sub(ktime_get(), start));
        pr_info("Filesystems sync: %ld.%03ld seconds\n",
                elapsed_msecs / MSEC_PER_SEC, elapsed_msecs % MSEC_PER_SEC);
}
EXPORT_SYMBOL_GPL(ksys_sync_helper);
```

`ksys_sync` wakes and instructs a set of flusher threads to write out every filesystem, first their inodes [15] , then the full filesystem, and then finally all block devices, to ensure all pages are written out to disk.

fs/sync.c:87

```
/*
 * Sync everything. We start by waking flusher threads so that most of
 * writeback runs on all devices in parallel. Then we sync all inodes reliably
 * which effectively also waits for all flusher threads to finish doing
 * writeback. At this point all data is on disk so metadata should be stable
 * and we tell filesystems to sync their metadata via ->sync_fs() calls.
 * Finally, we writeout all block devices because some filesystems (e.g. ext2)
 * just write metadata (such as inodes or bitmaps) to block device page cache
 * and do not sync it on their own in ->sync_fs().
 */
void ksys_sync(void)
{
        int nowait = 0, wait = 1;

        wakeup_flusher_threads(WB_REASON_SYNC);
        iterate_supers(sync_inodes_one_sb, NULL);
        iterate_supers(sync_fs_one_sb, &nowait);
        iterate_supers(sync_fs_one_sb, &wait);
```

```
        sync_bdevs(false);
        sync_bdevs(true);
        if (unlikely(laptop_mode))
                laptop_sync_completion();
}
```

It follows an interesting pattern of using `iterate_supers` to run both `sync_inodes_one_sb` and then `sync_fs_one_sb` on each known filesystem [16] . It also calls both `sync_fs_one_sb` and `sync_bdevs` twice, first without waiting for any operations to complete and then again waiting for completion [17] .

When `laptop_mode` is enabled the system runs additional filesystem synchronization operations after the specified delay without any writes.

mm/page-writeback.c:111

```
/*
 * Flag that puts the machine in "laptop mode". Doubles as a timeout in jiffies:
 * a full sync is triggered after this time elapses without any disk activity.
 */
int laptop_mode;

EXPORT_SYMBOL(laptop_mode);
```

However, when running a filesystem synchronization operation, the system will add an additional timer to schedule more writes after the `laptop_mode` delay. We don't want the state of the system to change at all while performing hibernation, so we cancel those timers.

mm/page-writeback.c:2198

```
/*
 * We're in laptop mode and we've just synced. The sync's writes will have
 * caused another writeback to be scheduled by laptop_io_completion.
 * Nothing needs to be written back anymore, so we unschedule the writeback.
 */
void laptop_sync_completion(void)
{
        struct backing_dev_info *bdi;
```

```
        rcu_read_lock();

        list_for_each_entry_rcu(bdi, &bdi_list, bdi_list)
                del_timer(&bdi->laptop_mode_wb_timer);

        rcu_read_unlock();
}
```

As a side note, the `ksys_sync` function is simply called when the system call `sync` is used.

fs/sync.c:111

```
SYSCALL_DEFINE0(sync)
{
        ksys_sync();
        return 0;
}
```

The End of Preparation

With that the system has finished preparations for hibernation. This is a somewhat arbitrary cutoff, but next the system will begin a full freeze of userspace to then dump memory out to an image and finally to perform hibernation. All this will be covered in future articles!

---

1. Hibernation modes are outside of scope for this article, see the previous article for a high-level description of the different types of hibernation. ↩
2. Workqueues are a mechanism for running asynchronous tasks. A full description of them is a task for another time, but the kernel documentation on them is available here: https://www.kernel.org/doc/html/v6.9/core-api/workqueue.html ↩ ↩[2]
3. This is a bit of an oversimplification, but since this isn't the main focus of this article this description has been kept to a higher level. ↩
4. Kconfig is Linux's build configuration system that sets many different macros to enable/disable various features. ↩
5. Kconfig defaults to the first default found ↩

6. Including checking whether the algorithm is larval? Which appears to indicate that it requires additional setup, but is an interesting choice of name for such a state. ↩

7. Specifically when we get to process freezing, which we'll get to in the next article in this series. ↩

8. Swap space is outside the scope of this article, but in short it is a buffer on disk that the kernel uses to store memory not current in use to free up space for other things. See Swap Management for more details. ↩

9. The code for this is lengthy and tangential, thus it has not been included here. If you're curious about the details of this, see kernel/power/hibernate.c:858 for the details of `hibernate_quiet_exec`, and drivers/nvdimm/core.c:451 for how it is used in `nvdimm`. ↩

10. Annoyingly this code appears to use the terms "console" and "virtual terminal" interchangeably. ↩

11. `ioctl`s are special device-specific I/O operations that permit performing actions outside of the standard file interactions of read/write/seek/etc. ↩ ↩[2]

12. I'm not entirely clear on how this flag works, this subsystem is particularly complex. ↩

13. In this case a higher number is higher priority. ↩

14. Or whatever the caller passes as `val_down`, but in this case we're specifically looking at how this is used in hibernation. ↩

15. An inode refers to a particular file or directory within the filesystem. See Wikipedia for more details. ↩

16. Each active filesystem is registed with the kernel through a structure known as a superblock, which contains references to all the inodes contained within the filesystem, as well as function pointers to perform the various required operations, like sync. ↩

17. I'm including minimal code in this section, as I'm not looking to deep dive into the filesystem code at this time. ↩

< What to Do When You Forget Your Root Password