UNIX & LINUX

How does the system shutdown of a linux kernel work internally?

Asked 8 years, 8 months ago Modified 8 years, 8 months ago Viewed 40k times

37

I have somehow an rough idea of how the userspace and init-system (be it classic init sysV /upstart/ systemd) work at system shutdown. (Essentially there is an order succession of "Stop!", "Please stop now really", "Process I need to kill you to Stop" and waiting... things going on).

I am anyhow very unaware of how the system shutdown works in the kernel (where surely there is also lots of stuff to do)?

I tried to look into the kernel documentation https://www.kernel.org/doc/htmldocs/ and even used the NSA's pal search tool to give me a head start on finding out how it works.

Also I searched on SE U+L and found nothing (did I overlook it?)

Anyways the question, though potentially a bit challenging, would merit an answer in this Q&A network as I assume more people are interested to get a sketch for what happens in the linux kernel at shutdown.

Potentially there is also change to link to some more detailed explanations.

An answer might maybe include which system-calls and which kernal signals are used?

https://github.com/torvalds/linux/blob/b3a3a9c441e2c8f6b6760de9331023a7906a4ac6/arch/x86/kernel/reboot.c seems to be the x86 used file related to reboot (already close to shutdown, eh?)

maybe the snippet found here http://lxr.free-electrons.com/source/kernel/reboot.c#L176 can be used to give an explanation

```
176 void kernel_power_off(void)
177 {
178         kernel_shutdown_prepare(SYSTEM_POWER_OFF);
179         if (pm_power_off_prepare)
180                 pm_power_off_prepare();
181         migrate_to_reboot_cpu();
182         syscore_shutdown();
183         pr_emerg("Power down\n");
184         kmsg_dump(KMSG_DUMP_POWEROFF);
185         machine_power_off();
186 }
187 EXPORT_SYMBOL_GPL(kernel_power_off);
```

linux-kernel shutdown

Share Improve this question Follow

edited Apr 1, 2014 at 12:11

asked Apr 1, 2014 at 11:30

humanityANDpeace

12.9k 11 58 98

10 may the unicorn be with you – Kiwy Apr 1, 2014 at 11:32

1 @Kiwy thanks for the suggestion. I will accept after some time has passed for potential better answers to come up. But at least some answer is now there. – humanityANDpeace Apr 1, 2014 at 13:28

Don't thank me, thanks the Unicorn ! – Kiwy Apr 1, 2014 at 13:30

Be aware that there is/was a *jump out the window* option to `shutdown(8)` i.e. the ***deprecated*** `-n` one which I *think* in old unix documentation used to read "*shutdown the system ourselves - the core unit is ON FIRE!*" effectively a messy system kill-switch which would/could leave bits scattered across the floor (or at least the file-systems in a corrupt state) - one imagines this would be used for a main-frame type system where someone has just caught their hand in a cooling fan. ☠ – SlySven Jul 13, 2017 at 14:58

2 Answers

Sorted by: Highest score (default)

37

The main resources to understand how the Linux kernel works are:

1. The documentation.
2. Linux Weekly News articles.
3. The source. This is a complex beast which is a little easier to apprehend through LXR, the Linux cross-reference. The LXR variant running on lxr.linux.no is nicer than others, but it's often down.

In this case, I can't find anything centrally relevant in the documentation or on LWN, so LXR it is.

The last thing the userland code does is call the **`reboot` system call**. It takes 4 arguments, so search for `SYSCALL_DEFINE4(reboot` on LXR, which leads to `kernel/reboot.c`. After checking the caller's privileges and the arguments, the syscall entry point calls one of several functions: `kernel_restart` to reboot, `kernel_halt` to halt on a tight loop, `kernel_poweroff` to power off the system, `kernel_kexec` to replace the kernel by a new one (if compiled in), or `hibernate` to save the memory to disk before powering off.

`kernel_restart`, `kernel_halt` and `kernel_power_off` are fairly similar:

1. Go through `reboot_notifier_list`, which is a list of hooks that kernel components can register to execute code on powerdown. Only a few drivers need to execute code at this stage, mostly watchdogs.
2. Set the `system_state` variable.
3. Disable usermode-helper, to ensure that no user code will be started anymore. (There can still be existing processes at this stage.)
4. Call `device_shutdown` to release or power down all devices on the system. A lot of drivers hook into this stage.
   Note that any filesystems that are still mounted at this point are effectively forcibly unmounted. The caller of the system call takes responsibility for any clean unmounting.
5. For power off only, if ACPI is configured in, possibly execute code to prepare going into ACPI state S5 (soft power off).
6. In a multi-CPU machine, the code could be running on any CPU, whichever invoked the system call. `migrate_to_reboot_cpu` takes care to switch to one particular CPU and prevent the scheduler from dispatching code on other CPUs. After this point, only a single CPU is running.
7. `syscore_shutdown` calls the `shutdown` method of registered syscore operations. I think this is mostly about disabling interrupts; few hooks have a `shutdown` method.
8. Log an information message — the swan's song.
9. Finally go to rest in some machine-dependent way by calling `machine_restart`

5. Finally go to rest in some machine-dependent way by calling `machine_restart`, `machine_halt` or `machine_power_off`.

The **hibernation** code goes through the following steps:

1. Iterate through the power management hooks.
2. Sync filesystems.
3. Freeze all user code.
4. Prevent device hotplugging.
5. Dump the system state to the swap space.
6. If everything succeeded, hibernate the hardware. This can involve calling `kernel_restart`, `kernel_halt` or `kernel_power_off`, or some platform-specific hibernation method.

A different way to shut the system down is `machine_emergency_restart`. This is invoked by the **magic SysRq key** B. The O key works differently: it calls `kernel_power_off`.

The system can also shut down to a panic, i.e. an unrecoverable error. Panicking attempts to log a message, then reboot the system (either via a hardware watchdog or an emergency restart).

Share Improve this answer Follow

answered Apr 2, 2014 at 1:01

Gilles 'SO- stop being evil'
778k 188 1608 2108

+1 thanks! @Gilles if you wanted to implement some code that would wipe/sanatize the RAM of the machine as a laststep you would register a syscore operation for the `syscore_shutdown` (i.e. that would solve my other question unix.stackexchange.com/q/122540/24394). Step (1) and step (7) both allow to register stuff to be executed at shutdown, not shure what is what + I had the impression the execution order of those callbacks in (1) and (7) cannot be influenced! I will the docs you mentioned, but if you know! thanks! – humanityANDpeace Apr 2, 2014 at 6:07

1 I'm surprised this question and answer does not have more upvotes. – user56041 Mar 14, 2019 at 13:11

@gilles-so-stop-being-evil - Would it be possible for existing user processes to still be running after the device_shutdown step? It might explain an issue we have with a running process not being able to access a mount to an external system. Our current reasoning is that the device is unmounted but the user process using the mount is not yet terminated. – Lieven Keersmaekers Mar 17 at 9:53

1 @LievenKeersmaekers If things haven't changed since I wrote this answer, when `device_shutdown` runs, the process data structures still exist, but the processes won't get any CPU time: no processes can *be running* after that point. I don't see how this could be related to a problem with *running* processes not being able to access a mount. – Gilles 'SO- stop being evil' Mar 17 at 11:48

1 @LievenKeersmaekers No. Killing a process closes all its open files and doesn't give it CPU time.

2

This is only a partial answer and I for sure invite other answer, which might be more exhaustive and clear.

The content of this answer is taken from the 3.13 linux kernel's `kernel/reboot.c` file (which might be not the first guess as the name is not shutdown.c but reboot.c)

Anyways in there we have basically three functions that sketch the process of shutting down the system

- `void kernel_halt(void)` // which ends with a system in halt state
- `void kernel_power_off(void)` // which ends with a system powered off
- `void kernel_restart(char *cmd)` // which ends the system to yet restart it

Those functions are very brief and can be hence pasted here in complete. Their code best shows what steps are taken on the way to shutdown in the kernel. (the comments are by me and might not be 100% ideal and correct, check yourself for being sure. It is simple a try.

`void kernel_halt(void)`

```
void kernel_halt(void)
{
    // 1st step does:
    // a) call functions/callback registered to run at reboot/shutdown
    // b) set system_sate to SYSTEM_HALT
    // c) stop the userspacetool interaction
    // d) call device_shutdown() function
    kernel_shutdown_prepare(SYSTEM_HALT);

    // 2nd step: I think this is mostly a necessity for multi-cpu systems
    migrate_to_reboot_cpu();

    // 3rd step:
    // syscore_shutdown - Execute all the registered system core shutdown callbacks
    syscore_shutdown();

    // 4th messages
    pr_emerg("System halted\n");
    kmsg_dump(KMSG_DUMP_HALT);

    // 5th call arch specific cpu-halt-code
    machine_halt();
}
```

the whole thing is initiated with the `sys_reboot` system call which, given that it does not only reboot but also shutdown, not the direct thing to connect with the shutdown process anyhow.

Share Improve this answer Follow

answered Apr 1, 2014 at 13:23

humanityANDpeace

12.9k 11 58 98