General troubleshooting

This article explains some methods for general troubleshooting. For application specific issues, please reference the particular wiki page for that program.

Related articles

Reporting bug guidelines

Step-by-step debugging guide

Debugging/Getting traces

1 General procedures

It is crucial to *always* read any error messages that appear. Sometimes it may be hard, e.g. with graphical applications, to get a proper error message.

1. Run the application in a terminal so it is possible to inspect the output.
    1. Increase the verbosity (usually `--verbose` / `-v` / `-V` or `--debug` / `-d`) if there is still not enough information to debug.
    2. Sometimes there is no such parameter and it needs to be specified as a directive in the applications' configuration file.
    3. An application may also use log files, which are usually located in `/var/log` , `$HOME/.cache` or `$HOME/.local`
    4. If there is no way to increase the verbosity, it is always possible to run **strace** and similar.
2. Check the **journal**. It is possible that an error may also leave traces in the journal, especially if it depends on other applications.
    1. *dmesg* reads from the kernel ring buffer. This is useful if the disk is for some reason inaccessible but this may also result in incomplete logs because the kernel ring buffer is not infinite in size. Use *journalctl* if possible.
    2. *journalctl* has more **filtering options** than *dmesg* and uses human-readable timestamps by default.
3. It is always recommended to check the relevant issue trackers to see if there are known issues with already existing solutions.
    1. Depending on upstreams' choices, there is usually an issue tracker and sometimes also a forum or even e.g. an IRC channel.
    2. There is the **Arch Linux bug tracker (https://gitlab.archlinux.org/groups/archlinux/packaging/-/issues)**, which should be primarily used for packaging bugs.

1.1 Additional support

If you require any additional support, you may ask on **the forums (https://bbs.archlinux.org)** or on **IRC**.

Note

Support is provided for Arch Linux ONLY and not **Arch-based distributions**.

When asking for support post the **complete** output/logs, not just what you think are the significant sections. Sources of information include:

- Full output of any command involved - do not just select what you think is relevant.
- systemd's **journal**.
  - For more extensive output, use the `systemd.log_level=debug` boot parameter. This will produce a tremendous amount of output, so only enable it if it is really needed.
  - Do not use the `-x` parameter because this needlessly clutters the output and makes it harder to read.
  - Use `-b` unless you need logs from a previous boot. Not specifying this may lead to extremely large pastes that may even be too big for any pastebins.
- Relevant configuration files
- Drivers involved
- Versions of packages involved
- Kernel: `journalctl -k` or `dmesg` (both with root privileges).
- **Xorg**: depending on the setup the **display manager** in use is relevant here, too.
  - `Xorg.log` may be located in one of several places: the system journal, `/var/log/` or `$HOME/.local/share/xorg/`.
  - Some display managers like **LightDM** may also place the `Xorg.log` in its own log directory.
- **Pacman**: If a recent upgrade broke something, look in `/var/log/pacman.log`.
  - It may be useful to use *pacman*'s `--debug` parameter.

One of the better ways to post this information is to use a **pastebin**.

A link will then be output that you can paste to the forum or IRC.

Additionally, you may wish to review **how to properly report issues (http://jdebp.info/FGA/problem-report-standard-litany.html)** before asking.

2 Boot problems

When diagnosing boot problems, it is very important to know in which stage the boot fails.

1. Firmware (UEFI or BIOS)
   1. Usually only has very basic tools for debugging.
   2. Make sure **Secure Boot** is disabled.
2. **Boot loader**
   1. One of the most common things done here is the changing of kernel parameters.
   2. Common boot issue during the boot loader stage could be caused by **ACPI**.
3. **initramfs**
   1. Usually provides an emergency shell.
   2. Depending on the hooks chosen, either the dmesg or the journal is available within it.
4. The actual system
   1. Depending on how badly it is broken, a simple invocation of the **debug shell** may suffice here.

If the debugging tools provided by any stage are not enough to fix the broken component, try using a e.g. **USB stick with the latest Arch Linux ISO** on it.

2.1 Console messages

After the boot process, the screen is cleared and the login prompt appears, leaving users unable to read init output and error messages. This default behavior may be modified using methods outlined in the sections below.

Note that regardless of the chosen option, kernel messages can be displayed for inspection after booting by using `journalctl -k` or `dmesg`. To display all logs from the current boot use `journalctl -b`.

2.1.1 Flow control

This is basic management that applies to most terminal emulators, including virtual consoles (VC):

- Press `Ctrl+s` to pause the output.
- And `Ctrl+q` to resume it.

This pauses not only the output, but also programs which try to print to the terminal, as they will block on the `write()` calls for as long as the output is paused. If your *init* appears frozen, make sure the system console is not paused.

To see error messages which are already displayed, see **Getty#Have boot messages stay on tty1**.

2.1.2 Printing more kernel messages

Most kernel messages are hidden during boot. You can see more of these messages by adding different kernel parameters. The simplest ones are:

- `debug`, which has the following effects:
  - The kernel will raise its console **logging level (https://docs.kernel.org/core-api/printk-basics.html)** such that all messages in the kernel log buffer will be printed to the console. **[1] (https://docs.kernel.org/admin-guide/kernel-parameters.html)**
  - **systemd** will raise its log level such that it will log debug messages that otherwise would not be produced anywhere. **[2] (https://github.com/systemd/systemd/blob/v251/src/basic/log.c#L1125-L1126)**
- `ignore_loglevel`, which has the same effect on the kernel as `debug` or `loglevel=8` (since debug messages are at `7`), but prevents the log level from being raised later in the boot.

Other parameters you can add that might be useful in certain situations are:

- `earlyprintk=vga,keep` prints kernel messages very early in the boot process, in case the kernel would crash before output is shown. You must change `vga` to `efi` for **EFI** systems.
- `log_buf_len=16M` allocates a larger (16 MiB) kernel message buffer, to ensure that debug output is not overwritten.

2.1.3 Producing debug kernel messages

#Printing more kernel messages indicates how to print of the kernel log buffer to the console, but that buffer itself won't contain any messages it didn't already (aside from the debug systemd output). This heading discusses methods for getting more detailed information out of the kernel log.

2.1.3.1 Dynamic debugging

Messages printed with pr_debug (https://docs.kernel.org/core-api/printk-basics.html#c.pr_debug) or related functions such as `dev_dbg()`, `drm_dbg()`, and `bt_dev_dbg()` will not be produced unless you either:

- Modify the kernel source to define `DEBUG` where desired.
- Utilize the kernel's dynamic debug (https://docs.kernel.org/admin-guide/dynamic-debug-howto.html) feature to enable debugging messages.

This section will discuss how to use dynamic debug, which is useful if you have already looked at your kernel log with everything up to informational logs, and would like even more debugging information from a particular location.

Firstly, you must be running a kernel that was compiled with the `CONFIG_DYNAMIC_DEBUG` kernel configuration option set. This is already the case for `linux` (https://archlinux.org/packages/?name=linux), so no action is required if you are using that kernel.

Then, you need to know where you want to see debug messages from. A couple of options are:

- Going with the kernel module name, if the issue seems to be isolated to a module. For example, to troubleshoot Intel graphics, you might concern yourself with the `i915` DRM kernel module.
- Going with a directory in the kernel that corresponds with functionality you are interested in. You will want to check out (or navigate online) the kernel source code to understand the structure. As an example, to inspect debug messages for all DRM kernel modules, you could go with the path drivers/gpu/drm (https://github.com/torvalds/linux/tree/v5.19/drivers/gpu/drm).

Using that "source" of messages, you have to come up with a dynamic debug query that indicates which debug messages to enable, of the format:

```
match_type match_parameter flags
```

Where:

- *match_type* is the type of match to make. Corresponding to the two options given earlier, this could be `module` or `file`.
- *match_parameter* is the module or file path to watch. In the latter case, using asterisks for wildcards is permissible.
- *flags* dictates what to do with the match. This could be `+p` to start printing its messages, or `-p` to undo that.

Some examples of queries are:

- `module i915 +p` to print debug messages from the `i915` kernel module.
- `file drivers/gpu/drm/* +p` to print debug messages from DRM drivers.

- `file * +p` to print debug messages.

Finally, to actually enact the query, you can either:

- Do so during runtime, by running:

```
# echo "query" > /sys/kernel/debug/dynamic_debug/control
```

  This assumes that **debugfs** is mounted at `/sys/kernel/debug/`, which you can verify using `mount`. **[3] (https://stackoverflow.com/a/63682160)**

- Do so at boot, by adding the `dyndbg="query"` **kernel parameter**

This is a greatly simplified overview of dynamic debug's capabilities; see **the documentation (https://docs.kernel.org/admin-guide/dynamic-debug-howto.html#command-language-reference)** for further details.

2.1.3.2 Subsystem-specific debugging

There are also a number of separate debug parameters for enabling debugging in specific subsystems e.g. `bootmem_debug`, `sched_debug`. Also, `initcall_debug` can be useful to investigate boot freezes. (Look for calls that did not return.) Check the **kernel parameter documentation (https://docs.kernel.org/admin-guide/kernel-parameters.html)** for specific information.

2.1.4 netconsole

netconsole (https://docs.kernel.org/networking/netconsole.html) is a kernel module that sends all kernel log messages (i.e. dmesg) over the network to another computer, without involving user space (e.g. syslogd). Name "netconsole" is a misnomer because it is not really a "console", more like a remote logging service.

It can be used either built-in or as a module. Built-in *netconsole* initializes immediately after NIC cards and will bring up the specified interface as soon as possible. The module is mainly used for capturing kernel panic output from a headless machine, or in other situations where the user space is no more functional.

2.2 Recovery shells

Getting an interactive shell at some stage in the boot process can help you pinpoint exactly where and why something is failing. There are several kernel parameters for doing so, but they all launch a normal shell which you can `exit` to let the kernel resume what it was doing:

- `rescue` launches a shell shortly after the root file system is remounted read/write
- `emergency` launches a shell even earlier, before most file systems are mounted
- `init=/bin/sh` (as a last resort) changes the init program to a root shell. `rescue` and `emergency` both rely on **systemd**, but this should work even if *systemd* is broken.

Another option is systemd's debug-shell which adds a root shell on `tty9` (accessible with `Ctrl+Alt+F9`). It can be enabled by either adding `systemd.debug_shell` to the **kernel parameters**, or by **enabling** `debug-shell.service`.

Warning

Remember to disable the service when done to avoid the security risk of leaving a root shell open on every boot.

2.3 Debugging kernel modules

See Kernel modules#Obtaining information.

2.4 Debugging hardware

- You can display extra debugging information about your hardware by following udev#Debug output.
- Ensure that Microcode updates are applied on your system.
- To test the RAM, see Stress testing#Memtest86+.
- To see if your system is overheating, use lm_sensors.
- To check your storage health, see S.M.A.R.T.

3 Debugging freezes

Unfortunately, freezes are usually hard to debug and some of them take a lot of time to reproduce. There are some types of freezes which are easier to debug than others:

- Is sound still playing? If so, just the display may be frozen. This may be a problem with the video driver.
- Is the machine still responding? Try SSH if switching to another TTY does not work.
- Is the disk activity LED (if present) indicating that a lot is being written to disk? Heavy swapping may temporarily freeze the system. See this StackExchange answer (https://unix.stackexchange.com/a/340567) for information about freezes on large writes.

If nothing else helps, try a **clean** shutdown. Pressing the power button *once* may unfreeze the system and show the classic "shutdown screen" which displays all the units that are getting stopped. Alternatively, using the magic SysRq keys may also help to achieve a clean shutdown. This is very important because the journal may contain hints why the machine froze. The journal may not be written to disk on an unclean shutdown. Hard freezes in which the whole machine is unresponsible are harder to debug since logs can not be written to disk in time.

Remote logging may help if the freeze does not permit writing anything to disk. A crude remote logging solution, which needs to be invoked from another device, can be used for basic debugging:

```
$ ssh freezing_host journalctl -f
```

Many fatal freezes in which the whole system does not respond anymore and require a forced shutdown may be related to buggy firmware, drivers or hardware. Trying a different kernel (see Kernel#Debugging regressions) or even a different Linux distribution or operating system, updating the firmware and running hardware diagnostics may help finding the problem.

Tip

It is recommended to try to update the firmware of the device, since these updates may fix strange issues.

If a freeze does not permit gathering any kind of logs or other information required for debugging, try reproducing the freeze in a live environment. If a graphical environment is required to reproduce the freeze or if the freeze can be reproduced on the archiso, use the live environment of a different distribution, which is preferably not based on Arch Linux to eliminate the possibility that the freeze is related to the version or patches of the kernel. Should the freeze still happen in a live environment, chances are that it *may* be hardware-related. If it does not happen anymore, it is necessary to be aware of the differences of both systems. Different configurations, differences in versions and kernel parameters and other, similar changes may have fixed the freeze.

However, a blinking caps lock LED may indicate a **kernel panic**. Some setups may not show the TTY when a kernel panic occurred, which may be confusing and can be interpreted as another kind of freeze.

4 Debugging regressions

Warning

This tends to cause **partial upgrades**, which are a necessary evil in this specific case. Proceed with caution and prepare a **method to recover your system**, just in case this scenario prevents a normal boot.

If an update causes an issue but **downgrading** the specific package fixes it, it is likely a **regression**. If this happened after a normal full system upgrade, check your `pacman.log` to determine which package(s) may have caused the issue. The most important part of debugging regressions is checking if the issue was already fixed, as this can save much time. To do so, first ensure the application is fully updated (e.g. ensure the application is the same version as in the **official repositories**). If it already is or if updating it does not fix the issue, try using the actual latest version, usually a **-git** version, which may already be packaged in the **AUR**. If this fixes the issue and the version with the fixes is not yet in the official repositories, wait until the new version arrives in them and then switch back to it.

If the issue still persists, debug the issue and/or **bisect** the application and report the bug on the upstream bug tracker so it can be fixed.

Note

The kernel needs a **slightly different approach** when debugging regressions.

5 Cannot use some peripherals after kernel upgrade

This will manifest commonly (but probably not only) as:

- newly plugged USB devices showing up with *dmesg* but not in `/dev/`,
- file systems unable to be mounted if they were not already used before the kernel update,

- the inability to use a wired/wireless connection on a laptop if it was not already used before the kernel update,
- `FATAL: Module module not found in directory /lib/module/kernelversion` when using *modprobe* to load a module that was not already used before the kernel package update.

As partially covered in System maintenance#Restart or reboot after upgrades, the kernel is not updated when you update the package but only once you reboot afterwards. Meanwhile, the kernel modules, located in `/usr/lib/modules/kernelversion/` are removed by pacman when installing the new kernel. As explained in FS#16702 (https://bugs.archlinux.org/task/16702), this approach avoids leaving files on the system not handled by the package manager but leads to the aforementioned symptoms. To fix them, reboot systematically after updating the kernel. The long-term evolution, yet to be implemented, will be to use versioned kernel packages : the main blocker being how to handle the removal of the previous kernel versions once they are not needed anymore.

Another solution is available as kernel-modules-hook (https://archlinux.org/packages/?name=kernel-modules-hook), where two pacman hooks use rsync to keep the kernel modules on the file system after the kernel update and `linux-modules-cleanup.service` that marks the old modules for removal four weeks after once enabled.

6 Package management

See Pacman#Troubleshooting for general topics, and pacman/Package signing#Troubleshooting for issues with PGP keys.

6.1 Fixing a broken system

If a partial upgrade was performed, try updating your whole system. A reboot may be required.

```
# pacman -Syu
```

If you usually boot into a GUI and that is failing, perhaps you can press `Ctrl+Alt+F1` through `Ctrl+Alt+F6` and get to a working tty to run *pacman* through.

If the system is broken enough that you are unable to run *pacman*, boot using a monthly Arch ISO from a USB flash drive, an optical disc or a network with PXE. (Do not follow any of the rest of the installation guide.)

Mount your root file system:

```
# mount /dev/rootFileSystemDevice /mnt
```

Mount any other partitions that you created separately, adding the prefix `/mnt` to all of them, i.e.:

```
# mount /dev/bootDevice /mnt/boot
```

Try using your system's *pacman* while chrooted:

```
# arch-chroot /mnt
# pacman -Syu
```

If that fails, exit the *chroot*, and try:

```
# pacman -Syu --sysroot /mnt
```

If that fails, try:

```
# pacman -Syu --root /mnt --cachedir /mnt/var/cache/pacman/pkg
```

7 fuser

fuser is a command-line utility for identifying processes using resources such as files, file systems and TCP/UDP ports.

fuser is provided by the `psmisc (https://archlinux.org/packages/?name=psmisc)` package, which should be already installed as a dependency of the `base (https://archlinux.org/packages/?name=base)` **meta package**. See `fuser(1) (https://man.archlinux.org/man/fuser.1)` for details.

8 Session permissions

Note

You must be using **systemd** as your init system for local sessions to work.**[4] (https://archlinux.org/news/d-bus-now-launches-user-buses/)** It is required for polkit permissions and ACLs for various devices (see `/usr/lib/udev/rules.d/70-uaccess.rules` and **[5] (https://enotty.pipebreaker.pl/2012/05/23/linux-automatic-user-acl-management/)**)

First, make sure you have a valid local session within X:

```
$ loginctl show-session $XDG_SESSION_ID
```

This should contain `Remote=no` and `Active=yes` in the output. If it does not, make sure that X runs on the same tty where the login occurred. This is required in order to preserve the logind session.

Basic **polkit** actions do not require further set-up. Some polkit actions require further authentication, even with a local session. A polkit authentication agent needs to be running for this to work. See **polkit#Authentication agents** for more information on this.

9 Message: "error while loading shared libraries"

If, while using a program, you get an error similar to:

```
error while loading shared libraries: libusb-0.1.so.4: cannot open shared object file: No such file or direc
tory
```

Use **pacman** or **pkgfile** to search for the package that owns the missing library:

```
$ pacman -F libusb-0.1.so.4

extra/libusb-compat 0.1.5-1
    usr/lib/libusb-0.1.so.4
```

In this case, the **`libusb-compat (https://archlinux.org/packages/?name=libusb-compat)`** package needs to be **installed**. Alternatively, the program requesting the library may need to be rebuilt following a **soname bump**.

The error could also mean that the package that you used to install your program does not list the library as a dependency in its **PKGBUILD**: if it is an official package, **report a bug**; if it is an **AUR** package, report it to the maintainer using its page in the AUR website.

10 See also

- **A how-to in troubleshooting for newcomers (https://gist.github.com/Hashino/6d54ba515c7407ab9dd780d0326997da)**
- **List of Tools for UBCD (https://wiki.ultimatebootcd.com/index.php?title=Tools)** - Memtest-like tools to add to grub.cfg on UltimateBootCD.com
- **Wikipedia:BIOS Boot partition**
- **REISUB**
- **systemd documentation on Debug Logging to a Serial Console (https://systemd.io/DEBUGGING/#debug-logging-to-a-serial-console)**
- **How to Isolate Linux ACPI Issues (https://web.archive.org/web/20120217124742/http://www.lesswatts.org/projects/acpi/debug.php)** on Archive.org

Retrieved from "https://wiki.archlinux.org/index.php?title=General_troubleshooting&oldid=844290"