Introduction

Abdul Rahim

July 19, 2024

Basic Info

- small incremental changes that add new features, make enhancement, fix bugs; are called **patches**
- new release in 2+ months
- new development and current release integration cycles run in parallel

Kernel release process

- lets call general updates made available to the public as launches
  - using this defination there are 2 types of launches
  - there is a **2 week merge window**, in which new features are accepted
- A major launch with **new features and fixes** is called a release
  - this is called **rc1**
- minor launches with **bug fixes only** called release candidates
  - once a launch is out in the public, it unravels bugs and security issues
  - these bugs/vulnerabilities are fixed with patches, and these paches are accumulated for a week and launched as minor launches; which is named like **rc2, rc3, rc4 . . .**

- usually the process of sending and including patches goes on upto **rc7, rc8**. Subject to weather **the code is good enough**?
- After that a 3 week quiet period starts. In this time, the maintainers prepare their trees, to send a pull request to linus
- After this 3 week period, the next merge window starts, in this merge window developers send their trees to linux via **signed pull**

Kernel release process

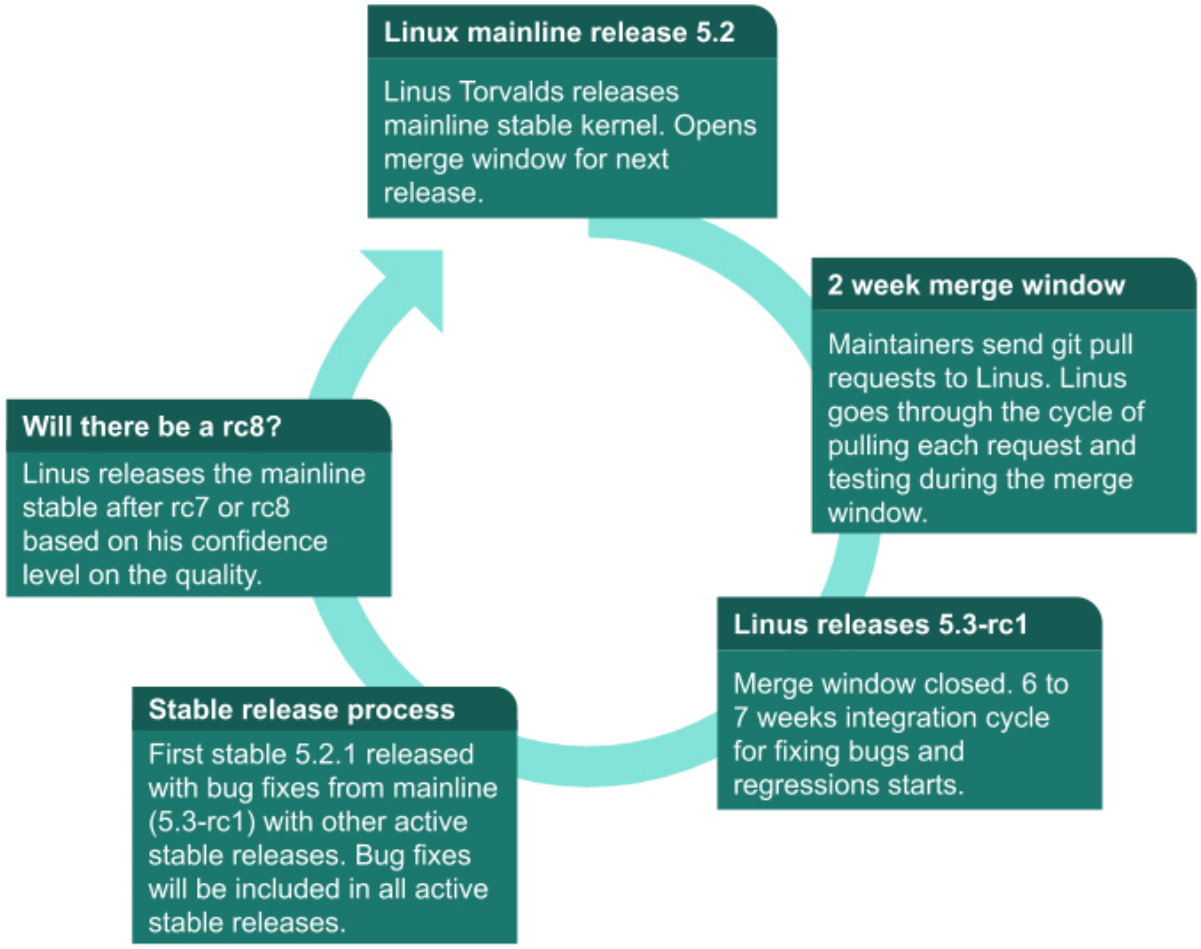


Figure 1: Release Process in the linux kernel

definations

- **release candidate/RC** releases are mainline kernel pre-releases that are used for testing new features in the mainline. These releases must be compiled from source. Kernel developers test these releases for bugs and regressions
- **stable releases** are bug fix only releases.
  - After a mainline kernel is released it moves into stable mode
  - bug fixes to stable kernel are backported *from mainline to stable* by designated stable kernel release maintainer
  - stable kernel release updates are released on average once a week(rc story)
- **long-term** releases are stable releases selected for long term maintenance
  - provide critical bug fixes to older kernel trees
- docs

kernel trees

- the kernel code is organised in several **main** and **subsystem repositories** called trees
- 1. **mainline kernel tree**: This is where linux releases mainline kernels and RC releases
  2. **stable tree**: this tree is maintained by greg kroah hartman. The tree consists of stable release branches, stable releases are based on this tree
  3. **linux-next**: this is the **integration** tree. code from large number of **subsystems trees** get pulled into this tree periodically, and then released for integration testing. This process of pulling changes from various trees catches merge conflicts between trees

- each subsysystem has its own tree and a maintainer(s), can find them *here*
- each subsystem has a mailing *list*
- development happens over mailing list:
  - contributors send patches to mailing list through emails and then they are discussed in mails
  - Once applied; a message is recieved “Thanks Applied”
  - *doc*

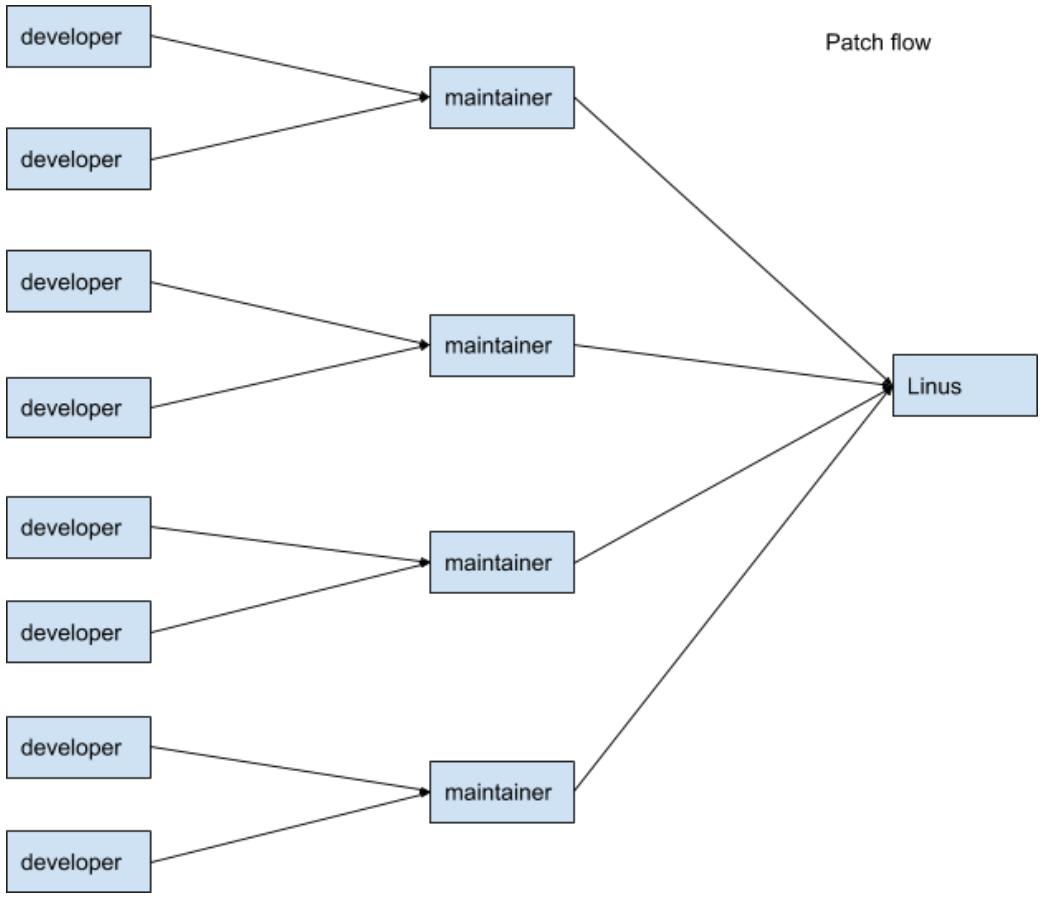


Figure 2: flow of patches

patches

abdul rahim

20 july 2024

basic survival skills

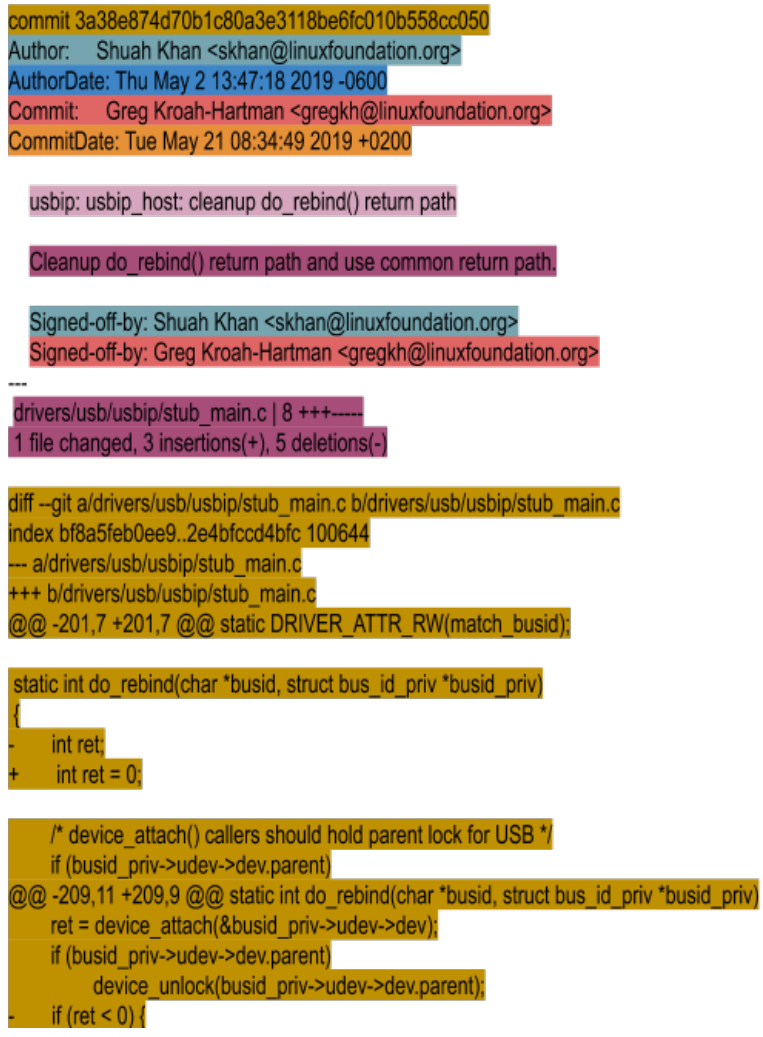
- these are basic git skills for survival
  - get commit history
  - working with remotes
  - creating branches
  - creating tags
  - rebasing tagged version
- provided in first 3 chapters of *git book*

- developers send changes, **through email as patches**, so patches are essentially emails,
- patches are **small incremental changes** made to the kernel
- a patch cannot break the kernel
- requiring each patch to do one thing helps isolate regression
- reverting a problem patch becomes easier as well
- complex changes are thus split into smaller chunks

whats in a ptch

- command to generate patch

```
1  git format-patch -1 --pretty=full 3a38e87...
```



```
commit 3a38e874d70b1c80a3e3118be6fc010b558cc050
Author:     Shuah Khan <skhan@linuxfoundation.org>
AuthorDate: Thu May 2 13:47:18 2019 -0600
Commit:     Greg Kroah-Hartman <gregkh@linuxfoundation.org>
CommitDate: Tue May 21 08:34:49 2019 +0200

    usbip: usbip_host: cleanup do_rebind() return path

    Cleanup do_rebind() return path and use common return path.

    Signed-off-by: Shuah Khan <skhan@linuxfoundation.org>
    Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
---
 drivers/usb/usbip/stub_main.c | 8 ++++-----
 1 file changed, 3 insertions(+), 5 deletions(-)

diff --git a/drivers/usb/usbip/stub_main.c b/drivers/usb/usbip/stub_main.c
index bf8a5feb0ee9..2e4bfccd4bfc 100644
--- a/drivers/usb/usbip/stub_main.c
+++ b/drivers/usb/usbip/stub_main.c
@@ -201,7 +201,7 @@ static DRIVER_ATTR_RW(match_busid);

 static int do_rebind(char *busid, struct bus_id_priv *busid_priv)
 {
-       int ret;
+       int ret = 0;

        /* device_attach() callers should hold parent lock for USB */
        if (busid_priv->udev->dev.parent)
@@ -209,11 +209,9 @@ static int do_rebind(char *busid, struct bus_id_priv *busid_priv)
        ret = device_attach(&busid_priv->udev->dev);
        if (busid_priv->udev->dev.parent)
                device_unlock(busid_priv->udev->dev.parent);
-       if (ret < 0) {
```

Figure 1: patch screenshot

patch components

1. commit id: hash of {commmit date, commiter name, commiter's email etc.}
2. **commit header**: follows the format:

```
major_subsystem:minor_area:
    short_description_of_what_is_being_changed
```

There is another varient, using (/) instead of (:), convention is set by maintainer

3. **commit log**: is a **detailed description** of the change and **why it is required**, and if **testing is done**
4. author: can be specified in `git commit` or `.gitconfig` file

5. Commiter's name and email:
   - The committer is a maintainer or developer that applies the patch to a git repository.
   - This patch was picked up by the USB maintainer Greg Kroah-Hartman and committed to the usb tree and tested prior to being included in a pull request from Greg KH to Linus Torvalds.
   - Since it was pulled, you won't see Linus as the committer for this patch. If you look at the git log, you will see several merge commits from Linus for when he pulls subsystem trees from maintainers.
6. sign-off: just a line you add; here's the *doc*

and others

Tags

1. Acked-by: used by maintainer
2. Reviewed-by: patch is reviewed by [name]
3. Reported-by: person who reported the bug
4. Tested-by: the person who tested the patch
5. Suggested-by: the person who suggested the idea
6. Fixes: indicates that this patch fixes issue in commit <commit_id>

subject of patch

the subject line in a patch can be

- [PATCH]: specifies that email contains a patch
- [PATCH RFC]: Request for comment c[patch
- [PATCH v4]: 4th revised version of patch

- patch version history:
  - When sending **reworked patch**, include version history
  - “what changed” between current and previous version of patch is added between −−− and start of diff in patch file
  - **any text added here (between - - - and start of diff) is not included in commit and will not be added to kernel when patch is applied**
  - include information helpful for reviewers here
  - an *example patch*
- do not send new version of a patch as a reply to previous version, start a new thread for each version of the patch

THANK YOU

configuring development environment

abdul rahim

20 july 2024

intro

- you need 3 GB of disk space in `boot` partition, to boot the kernel
- you can use Gpartition for that

emails

- generate patch with `git format-patch` command
- send patch with `git send-email my_patch.patch`

> Thats all you do

- configure your email client to be able send responses, to review comments, and other communication

email conventions

- bottom post: post your reply at bottom, Top posting is writing a message above the original text while sending a response to an email. Add your response at the bottom of the original text.
- delete or strip parts of message you are not replying to
- *doc*

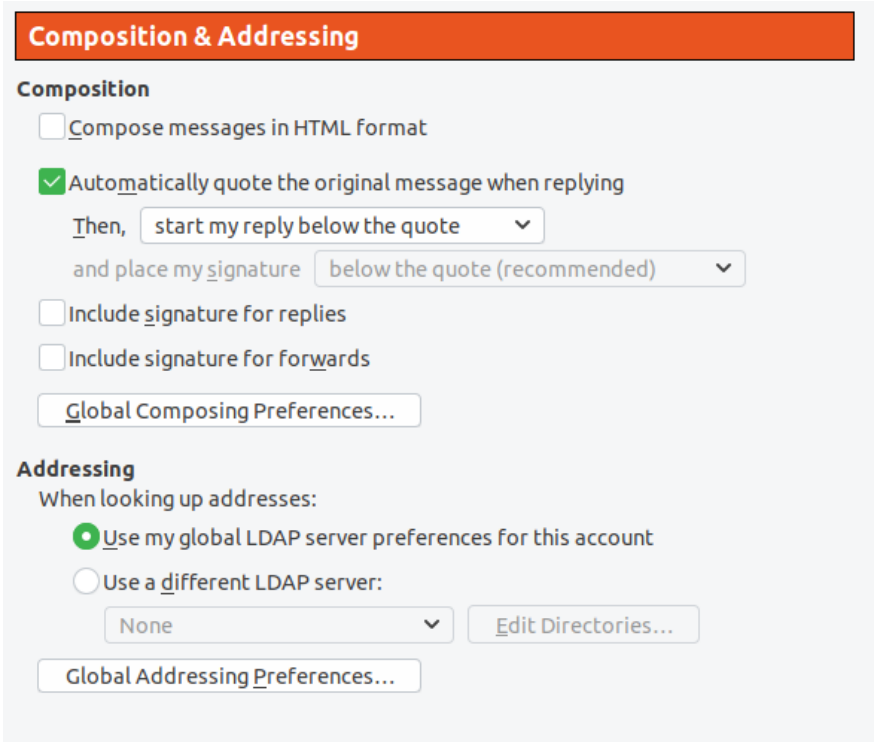


Figure 1: example configuration with thunderbird

Cool stuff now

- be in /linux_work
- clone the mainline

```
1  git clone git://git.kernel.org/pub/scm/linux/
       kernel/git/torvalds/linux.git
       linux_mainline
```

- note that changing the protocol makes clone much slower
- you can use *cregit* to view kernel sources

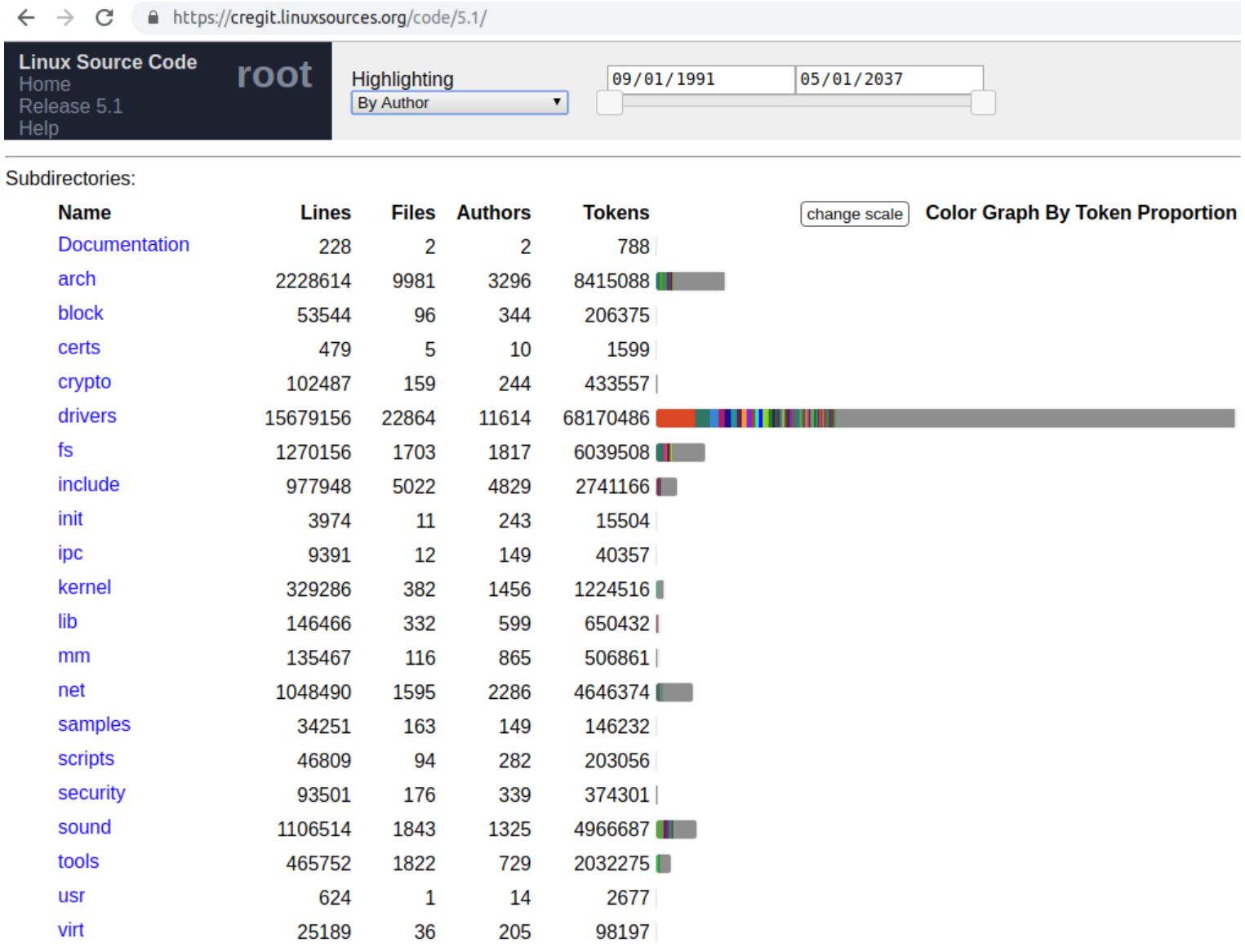


Figure 2: *cregit*

- Take a look at the Makefile and MAINTAINERS files in the main directory and scripts/get_maintainer.pl and scripts/checkpatch.pl. You will be using them in your everyday kernel development life.

```
# Get back to your root of the workspace
cd /linux_work

git clone git://git.kernel.org/pub/scm/linux/kernel/git/shuah/linux-kselftest.git
linux_kselftest

git branch -a
  devel
* fixes
  master
  next
  remotes/linux/master
  remotes/origin/HEAD -> origin/master
  remotes/origin/devel
  remotes/origin/fixes
  remotes/origin/khdr_fix
  remotes/origin/ksft-tap-refactor
  remotes/origin/master
  remotes/origin/next

git checkout next
Switched to branch 'next'
Your branch is up to date with 'origin/next'.
```

building and installing the kernel

abdul rahim

21 july 2024

> Disclaimer: Info here is from: LDF103

clone the stable release

```
1 git clone git://git.kernel.org/pub/scm/linux/
     kernel/git/stable/linux-stable.git
     linux_stable
2 cd linux_stable
```

switch to latest stable branch

```
git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/linux-2.6.11.y
  ...
  remotes/origin/linux-6.7.y
  remotes/origin/linux-6.8.y
  remotes/origin/linux-6.9.y
  remotes/origin/linux-rolling-lts
  remotes/origin/linux-rolling-stable
  remotes/origin/master

git switch remotes/origin/linux-6.9.y
```

generating .config

- copy the current configuration, into `linux_stable` directory

```
1 ls /boot # look for config-...-generic
2 cp /boot/config-...-generic /path/to/linux\
      _stable
```

- genereate `.config` based on current configuration

```
1  make oldconfig
```

- keep pressing enter to set default options, or choose what features you need

install dependencies

- these are the dependencies that I had to install; yours might be different based on configuration

```
1  sudo apt-get install build-essential vim git
       cscope libncurses-dev libssl-dev bison flex
        libelf-dev
```

compile the kernel

- run the command

```
make -jN
```

- where `N` is number of threads on your CPU

```
make -j$(nproc)
```

- if it gives an error like <xyz.h> missing, then install the missing dependencies
- after successfull compilation, `vmlinux` binary would be generated

install the kernel

```
1  make modules_install
2  sudo make install
```

```
sudo su
[sudo] password for kaku:
make modules_install install
  SYMLINK /lib/modules/6.9.10/build
  INSTALL /lib/modules/6.9.10/modules.order
    ...other drivers
  INSTALL /lib/modules/6.9.10/kernel/net/qrtr/
     qrtr-mhi.ko
  SIGN    /lib/modules/6.9.10/kernel/net/qrtr/
     qrtr-mhi.ko
  DEPMOD  /lib/modules/6.9.10
  INSTALL /boot
```

check your new kernel in /boot

```
{4:04}/boot ➪ ll
total 599M
drwx------. 4 root root 4.0K Dec 31  1969 efi
-rw-r--r--. 1 root root  160 Apr 10 20:00 .vmlinuz-6.8.5-301.fc40.x86_64.hmac
-rwxr-xr-x. 1 root root  15M Apr 10 20:00 vmlinuz-6.8.5-301.fc40.x86_64
-rw-r--r--. 1 root root 8.5M Apr 10 20:00 System.map-6.8.5-301.fc40.x86_64
-rw-r--r--. 1 root root 266K Apr 10 20:00 config-6.8.5-301.fc40.x86_64
lrwxrwxrwx. 1 root root   45 Apr 14 19:01 symvers-6.8.5-301.fc40.x86_64.xz -> /lib/modules/6.8.5-301.fc40.x86_64/symvers.xz
-rw-r--r--. 1 root root  161 May 16 20:00 .vmlinuz-6.8.10-300.fc40.x86_64.hmac
-rwxr-xr-x. 1 root root  15M May 16 20:00 vmlinuz-6.8.10-300.fc40.x86_64
-rw-r--r--. 1 root root 8.6M May 16 20:00 System.map-6.8.10-300.fc40.x86_64
-rw-r--r--. 1 root root 266K May 16 20:00 config-6.8.10-300.fc40.x86_64
drwx------. 2 root root  16K May 18 11:31 lost+found
drwxr-xr-x. 3 root root 4.0K May 18 11:36 loader
-rwxr-xr-x. 1 root root  15M May 18 11:37 vmlinuz-0-rescue-4f68f95714c84ebc8d232ac71f8406bc
-rw-------. 1 root root 155M May 18 11:38 initramfs-0-rescue-4f68f95714c84ebc8d232ac71f8406bc.img
-rw-------. 1 root root  58M May 18 11:39 initramfs-6.8.5-301.fc40.x86_64.img
-rw-r--r--. 1 root root  14M Jul 21 19:02 vmlinuz-6.9.10
dr-xr-xr-x. 1 root root  158 Jul 21 20:03 ..
-rw-------. 1 root root  47M Jul 21 20:25 initramfs-6.8.10-300.fc40.x86_64.img
lrwxrwxrwx. 1 root root   46 Jul 21 20:25 symvers-6.8.10-300.fc40.x86_64.xz -> /lib/modules/6.8.10-300.fc40.x86_64/symvers.xz
drwx------. 3 root root 4.0K Jul 22 03:47 grub2
dr-xr-xr-x. 6 root root 4.0K Jul 22 03:58 .
-rw-------. 1 root root 267M Jul 22 03:58 initramfs-6.9.10.img
{4:04}/boot ➪
```

Figure 1: your kernel must appear in **/boot**

update grub configuration

- there is no gurantee that newly installed kernel will boot
- need to ensure that there is atleast one **good kernel** to boot from
- increase the `GRUB\_TIMOUT` to make grub allow us enough time to be able to select kernel to boot
- in `/etc/default/grub`

enable early messages

- if the new kernel fails to boot; we should be able to see **early messages** to debug why it failed to boot
- enable early messages by changing `GRUB_CMDLINE_LINUX` to `earlyprintk-=vga`

```
1  GRUB_CMDLINE_LINUX="earlyprintk=vga"
```

- in `/etc/default/grub`
- run `sudo update-grub` to update grub configuration

restart the system

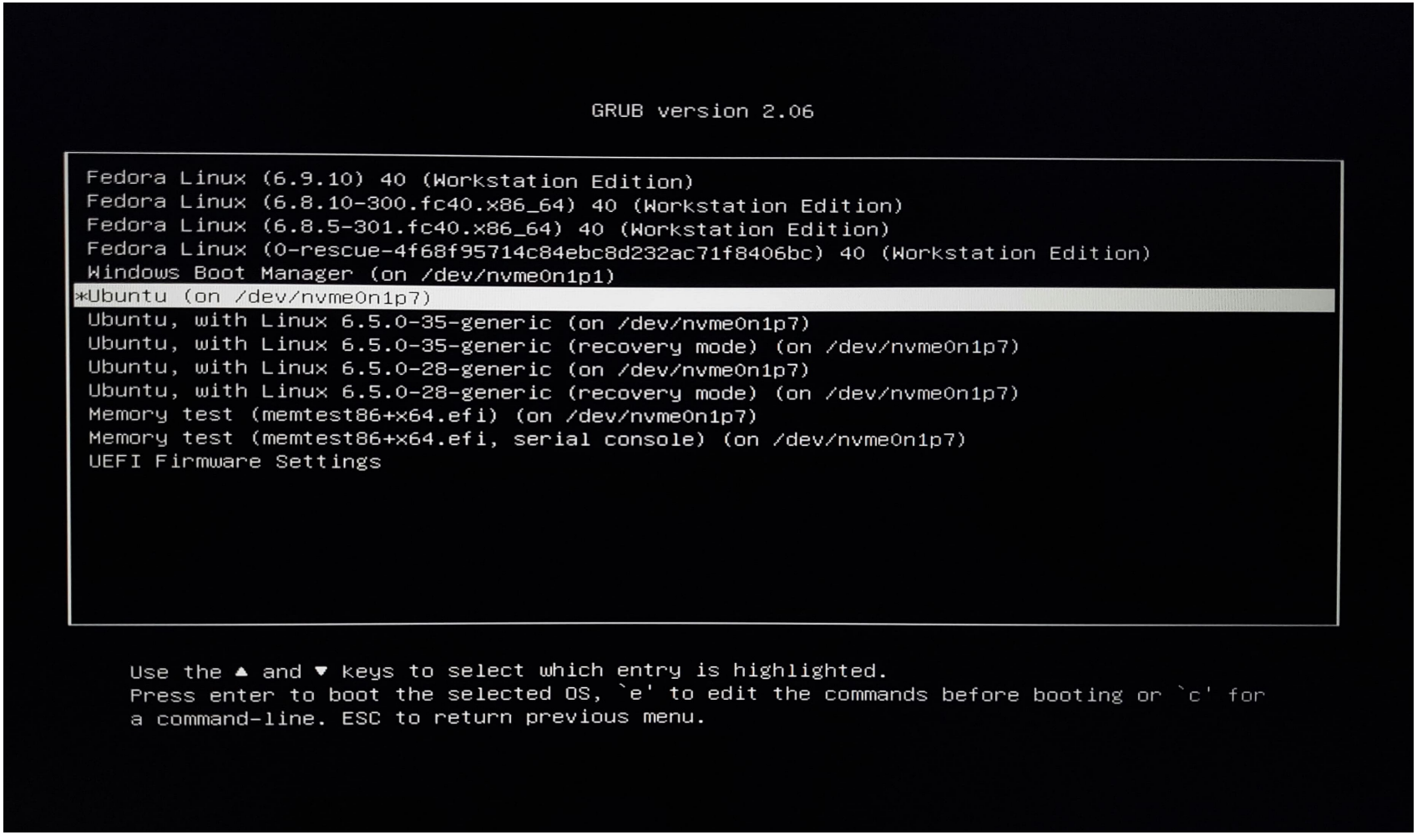


Figure 2: At the boot menu you see your new kernel; yours might look different

- boot into the new kernel
- check kernel verison to confirm

```
$ uname -r
6.9.10
```

new kernel fails to boot

- if kernel fails to boot; compare dmesg of your old kernel and newly build kernel to check for regressions (Checking whether changes to software have broken functionality that used to work).

Thank You

writing first patch

abdul rahim

21 july 2024

linux kernel configuration

- drivers configuration options
  1. Disabled: “n”
  2. Built into the kernel (vmlinux image) to be loaded at boot time: “y”
  3. Built as a module to be loaded as needed using `modprobe`: “m”

modules **(.ko)** files can be loaded when kernel detects hardware that matches the driver

- new releases often introduce new configuration variables or rename configuration symbols (in which case `make oldconfig` will not produce a working kernel)
- run `make listnewconfig` to see list of new configuration symbols

refer *this-doc* for kconfig and `make config`

and *this-doc* to learn about kernel build process

writing patch

- add remote; so you can fetch changes and choose a tag to rebase from

```
1  git remote add linux git://git.kernel.org/pub/
       scm/linux/kernel/git/torvalds/linux.git
2  git fetch linux
```

- create new branch in `linux_mainline`

```
1  git checkout -b test_patch
```

make change to a driver

- run `lsmod` to list modules loaded on your system and
- we will make a change to `uvcvideo` driver
- now you have to find `.c`, `.h` files; in linux kernel repository

searching Makefile can lead you there; `git grep` will get you there faster; since it ignores generated `.o`, `.ko`, ... files and `.git` folder

> uvcvideo is a USB Video Class (UVC) media driver for video input devices, such as webcams. It supports webcams on laptops.

Let’s check the source files for this driver.

```
1  ls drivers/media/usb/uvc/
```

Let's make a small change to the **probe function** of the uvcvideo driver. A probe function is called when the driver is loaded. Let's edit `uvc_driver.c`:

- find the `uvc\_probe` function; add `pr\_info("your message");` to it;
- A pr_info() function writes a message to the kernel log buffer, and we can see it using dmesg.

```
vim drivers/media/usb/uvc/uvc_driver.c
```

```
static int uvc_probe(struct usb_interface *
    intf,
                         const struct
                            usb_device_id *id)
{
        struct usb_device *udev =
            interface_to_usbdev(intf);
        ...
        int ret;

        pr_info("I changed uvcvideo driver in
            the Linux Kernel\n"); //Add this

        if (id->idVendor && id->idProduct)
        ...
}
```

- To make `uvcvideo` compiled as module; change `CONFIG_USB_VIDEO_CLASS=m`; in the Makefile
- Or you can built it into the kernel using “y” option

time to test your change

Recompile your kernel and install. Then try unloading and loading the module and you should see your message

- Unload module
  - using: `sudo rmmod uvcvideo`
  - Check `dmesg` for any messages about the uvcvideo module removal.
  - Run `lsmod | grep uvcvideo`, Do you see the module?
- Load module:
  - using: `sudo modprobe uvcvideo`
  - Run `lsmod | grep uvcvideo`. Do you see the module?
  - Run `dmesg | less` and search for “your message”. Do you see the message?

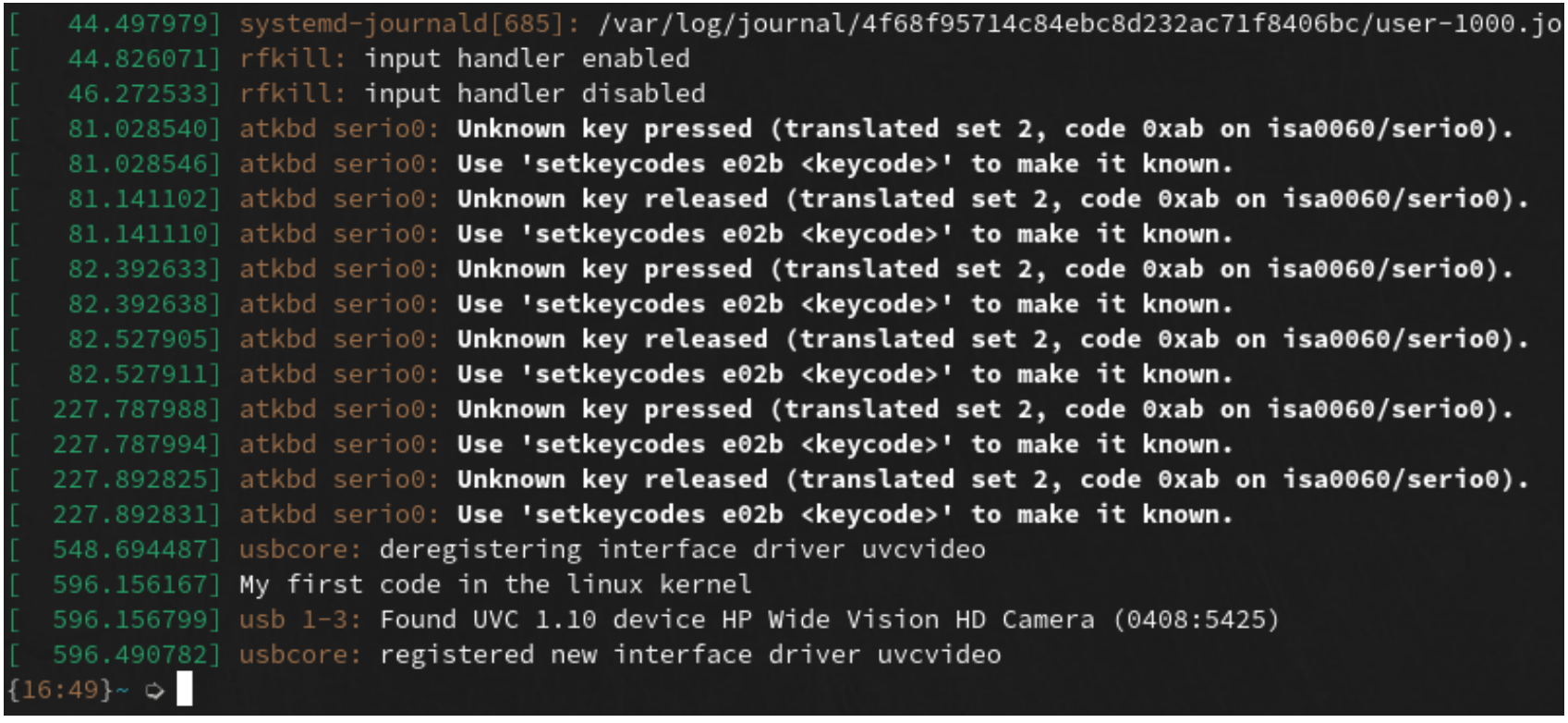


Figure 1: screenshot of dmesg; 3rd last line

preparing and sending patches

Abdul Rahim

23 July 2024

preparing patches

- ensure that your code conforms **linux kernel coding style**
- a script `checkpatch.pl` is provided; which you can run off of `diff`

```
1 git diff /path/to/changed/file  /tmp/df; \
2     scripts/checkpatch.pl /tmp/df
```

- run checkpatch.pl **before testing or commmiting** changes
- make necessary changes and fix warning/errors generated by the script

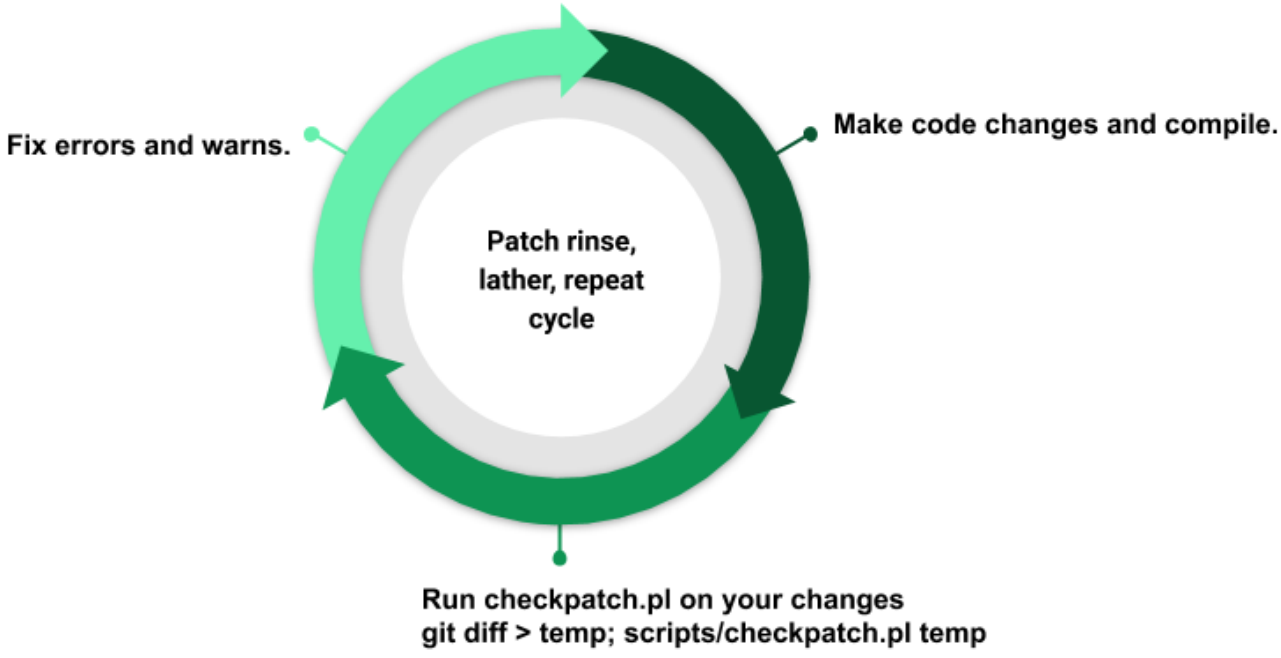


Figure 1: patch workflow

once checkpatch is happy; test your changes and commit your change

```
1  git commit -a # to commit all changes
2  git commit <filenames> # in case you want to
       send changes as seprate patches
```

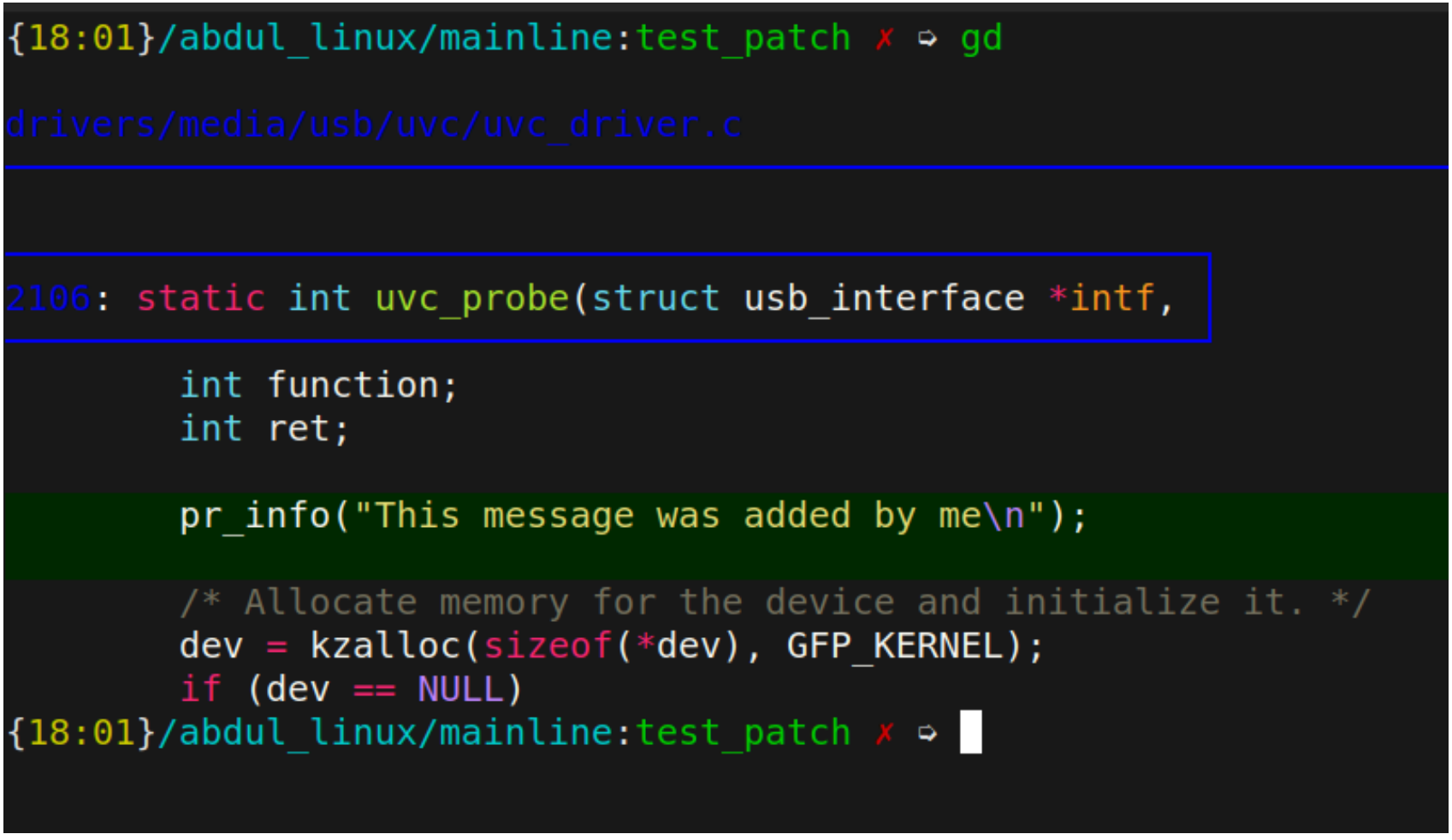


Figure 2: diff output

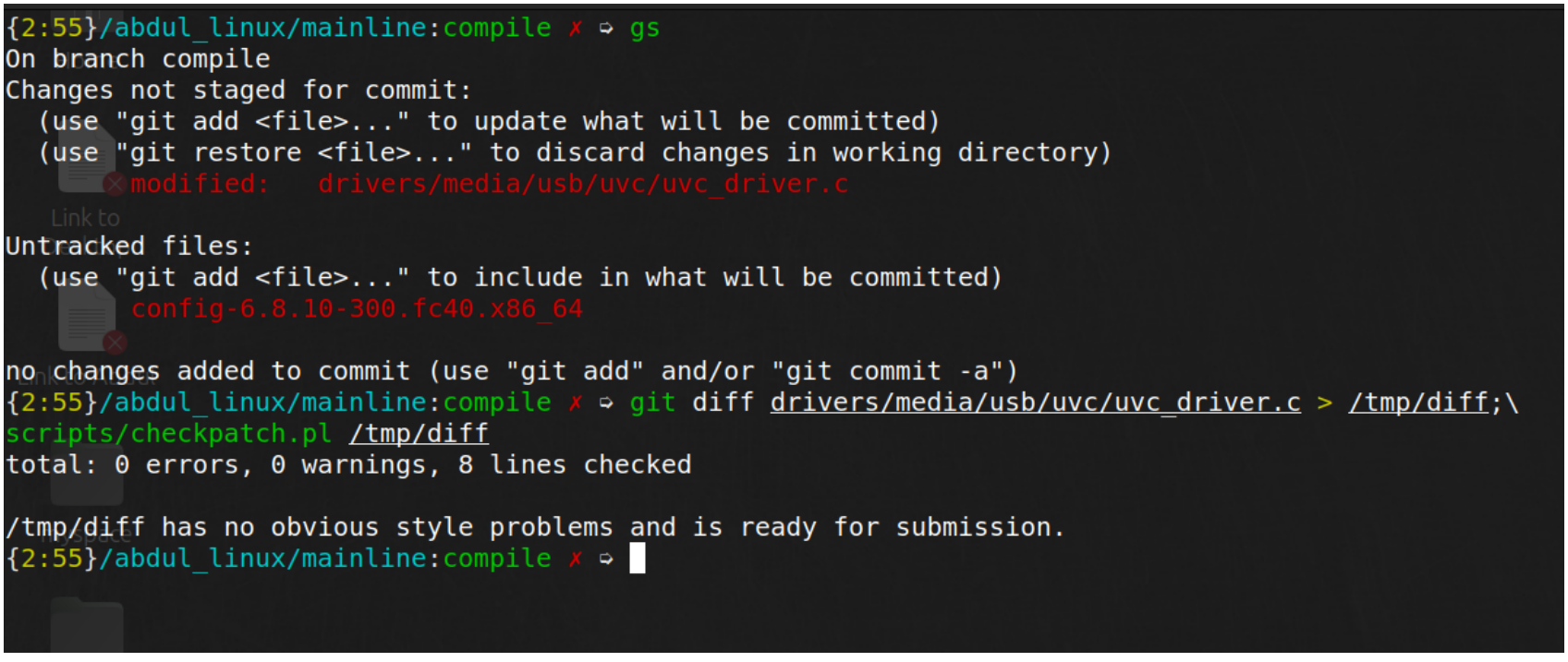


Figure 3: checkpatch script gives no warning

commit conventions

- commit message has a short subject; and a description/longer commit message;
- Explain: **why the code change is needed**, instead of **what code does**. Checkout *how to write a git commit*
- begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description.
- The **text up to the first blank line** in a commit message is treated as the commit **title**, and that title is used throughout Git. For example, **Git-format-patch(1) turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.**

generate patch

- After committing the change, generate the patch running the following command:

```
1  git format-patch -1 <commit ID>
```

- do not use the dont use –pretty option. Because then `git send-email` doesn't work.

sending your patch

send your patch using the command:

```
1  git send-email --to=email@gmail.com /path/to/
       patch
```

how and where to send patch

Abdul Rahim

23 July 2024

- the `get_maintainer.pl` tells you whome to send the patch to
- You should send the patch to *maintainers, commit signers, supporters, and all the mailing lists* shown in the `get\_maintainer.pl`'s output.
- Mailing lists are on the “cc” and the rest are on the “To” list when a patch is sent

```
scripts/get_maintainer.pl drivers/media/usb/uvc/uvc_driver.c
Laurent Pinchart <laurent.pinchart@ideasonboard.com> (maintainer:USB VIDEO CLASS)
Mauro Carvalho Chehab <mchehab@kernel.org> (maintainer:MEDIA INPUT INFRASTRUCTURE (V4L/DVB))
linux-media@vger.kernel.org (open list:USB VIDEO CLASS)
linux-kernel@vger.kernel.org (open list)
```

- Then you can run:

```
1  git format-patch -1 <commit ID> --to=laurent.
       pinchart@ideasonboard.com --to=
       mchehab@kernel.org --cc=linux-media@vger.
       kernel.org --cc=linux-kernel@vger.kernel.
       org
```

- This will generate a patch that you can send using:

```
1  git send-email <patch_file>
```

patch review

- Your patch will get comments from reviewers with suggestions. wait for a minimum of one week before requesting a response(might take longer).
- When you send a new version of your patch, add version history describing the changes made in the new version. The **right place for the version history is after the "—" below the Signed-off-by tag and the start of the changed file list**, as shown in the *screenshot*.
- Everything between the Signed-off-by and the diff is just for the reviewers, and will not be included in the commit. Please don't include version history in the commit log.

Best Practices for Sending Patches

- Document your change and include relevant testing details and results of that testing.
- Signed-off-by should be the last tag.
- Copy mailing lists and maintainers/developers suggested by `scripts/get_maintainer.pl`
- Always thank the reviewers for their feedback and address them.

- When working on a patch based on a suggested idea, make sure to give credit using the Suggested-by tag. Other tags used for giving credit are Tested-by, Reported-by.
- In general, getting response and comments is a good sign that the community likes the patch and wants to see it improved. Silence is what you want to be concerned about. If you don't hear any response back from the maintainer after a week, feel free to either send the patch again, or send a gentle "ping" - something like "Hi, I know you are busy, but have you found time to look at my patch?"
- Expect to receive comments and feedback at any time during the review process.

- Stay engaged and be ready to fix problems, if any, **after the patch gets accepted into linux-next for integration into the mainline**.
- Kernel build and Continuous Integration (CI) bots and rings usually find problems.
- When a patch gets accepted, you will either see an email from the maintainer or an automated patch **accepted email** with information on **which tree it has been applied to**, and some estimate on when you can expect to see it in the mainline kernel.

- Not all maintainers might send an email when the patch gets merged. The patch could stay in linux-next for integration until the next merge window, before it gets into Linus's tree. Unless the patch is an **actual fix to a bug in Linus's tree**, in which case, it may go directly into his tree.

Sometimes you need to send multiple related patches. This is useful for grouping, say, to **group driver clean up patches for one particular driver into a set**, or grouping patches that are part of a **new feature** into one set.

```
git format-patch -2 -s --cover-letter --thread
    --subject-prefix="PATCH v3" --to= "name"
    --cc="name"
```

will create a threaded patch series that includes the top two commits and generated cover letter template. It is a good practice to send a cover letter when sending a patch series.

- Including patch series version history in the cover letter will help reviewers get a quick snapshot of changes from version to version. When a maintainer accepts a patch, the maintainer assumes maintenance responsibility for that patch. As a result, maintainers have decision power on how to manage patch flow into their individual sub-system(s) and they also have individual preferences. Be prepared for maintainer-to-maintainer differences in commit log content and sub-system specific coding styles.
- *sample*

git hooks

- Git includes some hooks for **scripts** that can be run **before or after specific git commands are executed**.
- Checking the patch for compliance and errors can be automated using **git pre-commit** and **post-commit hooks**. The post-commit hook is run after you make a git commit with the git commit command.

- If you don't already have `/usr/share/codespell/dictionary.txt`, do:

```
1  sudo apt-get install codespell
```

- If you already have a `.git/hooks/post-commit` file, move it to `.git/hooks/post-commit.sample`. git will not execute files with the .sample extension.
- Then, edit the `.git/hooks/post-commit` file to contain only the following two lines:

```
1  #!bash
2  #!/bin/sh
3  exec git show --format=email HEAD | ./scripts/
       checkpatch.pl --strict --codespell
4  # Make sure the file is executable:
5  chmod a+x .git/hooks/post-commit
```

- After you make the commit, this hook will output any checkpatch errors or warnings that your patch creates
- if you want to avoid running this hook use:

```
1 git add .
2 git commit --amend
```

driver building

Abdul Rahim

23 July 2024

after making changes to a driver; it is convenient to compile the driver only instead of instead of a complete kernel build

after building the module, it can be **reloaded**.

targated compilation

to compile a single source file

```
1  make /path/to/object/file.o
```

to compile a directory

```
1  make path/to/directory
```

compile as a module

example of compiling vimrc module

```
1  make M=drivers/media/test-drivers/vimc
```

you need to figure out dependencies for a module/driver/configuration option, until all dependencies are enabled, The driver you are looking ot enable will not be enabled.

Lets figure out dependencies of `vimc`: vimc can be enabled by changing the `CONFIG_VIDEO_VIMC` option. Also it is a tristate driver, which means it has options y,n,m for kconfig

It depends on `CONFIG_VIDEO_DEV`, `CONFIG_VIDEO_V4L2`, and `CONFIG_VIDEO_V4L2_SUBDEV_API` to be enabled. It will also autoselect `CONFIG_VIDEOBUF2_VMALLOC` and `CONFIG_VIDEO_V4L2_TPG`.

some options are boolean; which means they can either be enabled or disabled.

use `make menuconfig`. to tweak config

testing

Abdul Rahim

23 July 2024

Developer testing, integration testing, regression, and stress testing have different individual goals. However, from 1,000 feet up, the end goal is the same: to ensure the software continues to work as it did before adding a new body of code, and that new features work as designed. It is time to look at testing.

Automated build bots and continous integration (CI) test rings run tests on various kernel repositories. The 0-day build bot can pull in patches and run build tests on several configurations and architectures. It is helpful in finding compile errors and warnings that might show up on other architectures that developers might not have access to.

Continuous Integration (CI) rings are test farms that host several platforms and run boot tests and Kernel Selftests on stable, linux-next, and mainline trees, these tests run in real hardware as well as qrmu environments. covering a wide range of archectures and configurations.

Applying patches

- Each Linux patch is a self-contained change to the code that stands on its own, unless explicitly made part of a patch series. New patches are applied as follows:

```
1  git apply --index file.patch
```

- If your use `patch` command; The problem with this is that: git reset –hard command will not remove the newly created files and a subsequent `git pull` will fail; hence it is not recommended.

basic testing

- check `dmesg`
- Is networking (wifi or wired) functional?
- Does ssh work?
- Run rsync of a large file over ssh
- Run git clone and git pull
- Start a web browser
- Read email
- Download files: ftp, wget, etc.
- Play audio/video files
- Connect new USB devices - mouse, USB stick, etc.

dmesg

Checking for regressions in `dmesg` is a good way to identify problems, if any, introduced by the new code. As a general rule, **there should be no new crit, alert, and emerg level messages in dmesg**. There should be no **new err level messages**. Pay close attention to any new warn level messages as well. Please note that **new warn messages are not as bad**. At times, new code adds new warning messages which are just warnings.

```
dmesg -t -l emerg
dmesg -t -l crit
dmesg -t -l alert
dmesg -t -l err
dmesg -t -l warn
dmesg -t -k
dmesg -t
```

Are there any stack traces resulting from WARN_ON in the dmesg? These are serious problems that require further investigation.

stress test

Performance problems are hard to debug. The first step is to detect them. Running several compiles in parallel is a good overall stress test that could be used as a performance regression test and overall kernel regression test, as it exercises various kernel modules like memory, filesystems, dma, and drivers.

```
1  time make all
```

debug options for testing your code

As you are making changes to drivers and other areas in the kernel, there are a few things to watch out for. There are several configuration options to test for locking imbalances and deadlocks. It is important to remember to enable the Lock Debugging and CONFIG_KASAN for memory leak detection. We do not intend to cover debugging in depth in this chapter, but we want you to start thinking about debugging and configuration options that facilitate debugging.

Enabling the following configuration option is recommended for testing your changes to the kernel:

- CONFIG_KASAN
- CONFIG_KMSAN
- CONFIG_UBSAN
- CONFIG_LOCKDEP
- CONFIG_PROVE_LOCKING
- CONFIG_LOCKUP_DETECTOR

Running `git grep -r DEBUG | grep Kconfig` can find all supported debug configuration options.

Checkout *LF mentorship series*

You can find resources on static and dynamic program analysis, fuzz testing, and Kernel testing using KUnit and Kselftest.

debugging

Abdul Rahim

24 July 2024

Asking the following questions can help to understand and identify the nature of the problem and how best to solve it:

- Is the problem easily reproducible?
- Is there a reproducer or test that can trigger the bug consistently?
- Are there any panic, or error, or debug messages in the dmesg when the bug is triggered?
- Is reproducing the problem time-sensitive?

Easily reproducible bugs with a test to trigger make it easier to debug, identify the problem, fix it and verify the fix. Time-sensitive problems could be a result of race conditions, and these are harder to debug and fix.

kernel panic

- kernel panic
  - A kernel panic is an action taken by an operating system upon detecting an internal fatal error from which it cannot safely recover and force the system to do a **controlled system hang / reboot** due to a detected **run-time system malfunction** (not necessarily an OOPS). The operation of the panic kernel feature may be controllable via **run-time sysconfig settings*** such as **hung task handling**.
- oops
  - oops are due to kernel exception handler getting executed including maros such as BUG ( ) which is defined as a invalid instruction. each exception has a unique number. Many oops result in kernel panic.

types of panics

panics are classified as:

1. hard panics: Aiee
2. soft panics: Oops

oops

when a kernel oops is encountered in a runnning kernel an OOPS message like:

```
1  [67.994624] Internal error: Oops: 5 [#1]
       XXXXXXXXXXXi
```

is displayed in the screen. An oops message contains:

- values of CPU registers
- address of the dunction that invoked the failure i.e.
  - PC(Program Counter)
  - stack
  - name of current process executing

how to debug oops

in linux `System.map` is a symbol table used by the kernel. System.map is required when symbol of an address or address of a symbol is required. The kernel does address to name translation when CONFIG_KALLSYMS is enabled so that tools like ksymoops are not required.

Note in the kernel backtraces in the logs, the kernel finds the nearest symbol to the address being analysed. Not all function symbols are available because of inlining, static, and optimisation so sometimes the reported function name is not the location of the failure.

using kernel stacktrace and objdump

1. check kernel stack trace, figure out the function address where the problem is occuring
2. run `objdump` on vmlinux and find out the instruction near the function. See the instructions near; verify the addresses match and find out the cause

using gdb

1. identify the line of code from panic/oops message;
2. run gdb list command; which will tell you the exact line where the error occured

```
(gdb) list *(function+0xoffset)
```

references

1. *panics*
2. *creating kernel oops*

decode stacktrace script

Panic messages can be decoded using the `decode_stacktrace.sh` tool.

```
Usage:
        scripts/decode_stacktrace.sh -r <release
            > | <vmlinux> [base path] [modules
            path]
```

Save (cut and paste) the panic trace in the dmesg between the two following lines of text into a .txt file.

```
------------[ cut here ]------------
------------[ end trace ]------------
```

Run this tool in your kernel repo. You will have to supply the [base path], which is the root of the git repo where the vmlinux resides if it is different from the location the tool is run from. If the panic is in a dynamically loaded kernel module, you will have to pass in the [modules path] where the modules reside.

```
scripts/decode_stacktrace.sh ./vmlinux <
   panic_trace.txt
```

It goes without saying that reading code and understanding the call trace leading up to the failure is an essential first step to debugging and finding a suitable fix.

final

Abdul Rahim

24 July 2024

some useful scripts to install:

stable_rc_checkout.sh

```
1        #!/bin/bash
2        ## SPDX-License-Identifier: GPL-2.0
3        # Copyright(c) Shuah Khan <
             skhan@linuxfoundation.org>
4       #
5        # LICENSE: GPLv2
6        # Example usage: stable_rc_checkout.sh <
             stable-rc e.g 5.2>
7        mkdir -p stable_rc
8        cd stable_rc
9        git clone git://git.kernel.org/pub/scm/
             linux/kernel/git/stable/linux-stable-
             rc.git
```

linux-$1.y

```
1       cd linux-$1.y
2       #cp /boot/<currentconfig> .config #
            update script
3       make -j2 all
4       rc=$?; if [[ $rc !=0 ]]; then exit $rc;
            fi
5       su -c "make modules_install install"
6       echo Ready for reboot test of Linux-$1
```

Download and applying stable release patches

Alternately, you can download and apply the patch. The following is my workflow for getting the repository ready, applying the patch, compiling, and installing. Run the `stable_checkout.sh` script once to set up your stable repository. After that, run `pre_compile_setup.sh` to get the patch file and apply whenever a stable release patch is released. I apply patches and use the same repository to be able to detect regressions. I save dmesg for the current rc to compare with the next rc. Please feel free to make changes to suit your needs. Also, make sure to pass in the correct release information from the stable release emails as arguments to this script.

stable_checkout.sh

```
#!/bin/bash
## SPDX-License-Identifier: GPL-2.0
# Copyright(c) Shuah Khan <skhan@linuxfoundation.org>
#
# License: GPLv2
# Example usage: stable_checkout.sh <stable-release-version e.g 5.2>
mkdir -p stable
cd stable
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git linux_$1_stable
cd linux_$1_stable
git checkout linux-$1.y
#cp /boot/ .config # update script
```

pre_compile_setup.sh

```
1  #!/bin/bash
2  ## SPDX-License-Identifier: GPL-2.0
3  # Copyright(c) Shuah Khan <
       skhan@linuxfoundation.org>
4  #
5  # License: GPLv2
6  # Example usage: pre_compile_setup.sh 5.2.11 1
       5
7  # Arg 1 is the stable release version which is
       typically 5.2.x
8  # Arg2 is the 1 for rc1 or 2 for rc2
9  # Arg3 is 4.x or 5.x used to call wget to get
       the patch file
```

```
1  echo Testing patch-$1-rc$2
2  wget https://www.kernel.org/pub/linux/kernel/
       v$3.x/stable-review/patch-$1-rc$2.gz
3  git reset --hard
4  make clean
5  git pull
6  gunzip patch-$1-rc$2.gz
7  git apply --index patch-$1-rc$2
8  echo "Patch-$1-rc$2 applied"
9  head Makefile
10 make -j2 all
11 rc=$?; if [[ $rc != 0 ]]; then exit $rc; fi
12 su -c "make modules_install install"
13 echo Ready for reboot test of Linux-$1-$2
```

dmesg_checks.shi

```
1  # !/bin/bash
2  #
3  #SPDX-License-Identifier: GPL-2.0
4  # Copyright(c) Shuah Khan <
       skhan@linuxfoundation.org>
5  #
6  # License: GPLv2
7
8        if [ "$1" == "" ]; then
9           echo "$0 " <old name -r>
10          exit -1
11 fi
```

```
release=`uname -r`
echo "Start dmesg regression check for
    $release" > dmesg_checks_results

echo "----------------------------" >>
    dmesg_checks_results

dmesg -t -l emerg > $release.dmesg_emerg
echo "dmesg emergency regressions" >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
diff $1.dmesg_emerg $release.dmesg_emerg >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
```

```
1  dmesg -t -l crit > $release.dmesg_crit
2  echo "dmesg critical regressions" >>
       dmesg_checks_results
3  echo "----------------------------" >>
       dmesg_checks_results
4  diff $1.dmesg_crit $release.dmesg_crit >>
       dmesg_checks_results
5  echo "----------------------------" >>
       dmesg_checks_results
6
7  dmesg -t -l alert > $release.dmesg_alert
8  echo "dmesg alert regressions" >>
       dmesg_checks_results
9  echo "----------------------------" >>
       dmesg_checks_results
10 diff $1.dmesg_alert $release.dmesg_alert >>
       dmesg_checks_results
11 echo "----------------------------" >>
       dmesg_checks_results
```

```
dmesg -t -l err > $release.dmesg_err
echo "dmesg err regressions" >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
diff $1.dmesg_err $release.dmesg_err >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results

dmesg -t -l warn > $release.dmesg_warn
echo "dmesg warn regressions" >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
diff $1.dmesg_warn $release.dmesg_warn >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
```

```
1  dmesg -t > $release.dmesg
2  echo "dmesg regressions" >>
       dmesg_checks_results
3  echo "----------------------------" >>
       dmesg_checks_results
4  diff $1.dmesg $release.dmesg >>
       dmesg_checks_results
5  echo "----------------------------" >>
       dmesg_checks_results
```

```
dmesg -t > $release.dmesg_kern
echo "dmesg_kern regressions" >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results
diff $1.dmesg_kern $release.dmesg_kern >>
    dmesg_checks_results
echo "----------------------------" >>
    dmesg_checks_results

echo "----------------------------" >>
    dmesg_checks_results

echo "End dmesg regression check for $release"
     >> dmesg_checks_results
```

keep contributing

Abdul Rahim

24 July 2024

ways to contribute to the kernel

- Subscribe to the Linux Kernel mailing list for the area of your interest.
- Follow the development activity reading the Linux Kernel Mailing List Archives.
- Find spelling errors in kernel messages.
- Static code analysis error fixing: Static code analysis is the process of detecting errors and flaws in the source code. The Linux kernel

- Makefile can be invoked with options to enable to run the Sparse source code checker on all source files, or only on the re-compiled files.
  - Compile the kernel with the source code checker enabled and find errors and fix as needed.
- Fix the Syzbot null pointer dereference and WARN bug reports which include the reproducer to analyze.
  - Run the reproducer to see if you can reproduce the problem.
  - Look at the crash report and walk through sources for a possible cause. You might be able to fix problems.

- Look for opportunities to add/update `.gitignore` files for tools and Kselftest. Build tools and Kselftest and run git status. **If there are binaries**, then it is time to add a new .gitignore file and/or an entry to an existing .gitignore file.
- Run mainline kernels built with the `CONFIG_KASAN`, Locking debug options mentioned earlier in the debugging section, and report problems if you see any. This gives you an opportunity to debug and fix problems. The community welcomes fixes and bug reports