kernel (/tag/kernel) debugging (/tag/debugging)

Linux Kernel Testing and Debugging

by Shuah Khan (/users/shuah-khan) on July 10, 2014

Linux Kernel Testing Philosophy

Testing is an integral and important part of any software development cycle, open or closed, and Linux kernel is no exception to that. Developer testing, integration testing, regression, and stress testing have different individual goals, however from 1000 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 the new features work as designed.

Ensuring software is stable without regressions before the release, helps avoid debugging and fixing customer and user found bugs after the release. It costs more in time and effort to debug and fix a customer found problem. Hence, testing is very important in the case of any software, not just the Linux kernel. Unlike closed and proprietary operating systems, the development process is open and is not locked down. This process is its strength as well as weakness. With several developers continuing to add new features, and fixing bugs, continuous integration and testing is vital to ensure the kernel continues to work on existing hardware as new hardware support and features get added. In the open source development, developers and users share the testing responsibility. This is another important difference between a closed development model and an open one.

Almost all Linux kernel developers, if not all, are very active Linux users themselves. There is no requirement that testers should be developers, however, users and developers that are not familiar with the new code could be more effective at testing a new piece of code than the original author of that code. In other words, developer testing serves as an important step in verifying the functionality, however, developer testing alone is not sufficient to find interactions with other code, features, and unintended regressions on configurations and/or hardware, developer didn't anticipate and didn't have the opportunity and resources to test. Hence, users play a very important role in the Linux Kernel development process.

So now that we understand the importance of continuous integration testing, we will go into the details of testing itself. Before we talk about testing, I would like to walk through the development process itself to help understand how it works and how the changes funnel into the mainline kernel.

3000+ kernel developers from around the world contribute to the Linux kernel. It is a 24hours, seven days a week, and 365 days of continuous development process that results in a new release once every 2+ months and several stable and extended stable releases. New development and current release integration cycles run in parallel.

For further reading on the development process, please refer to Greg Kroah-Hartman's presentation on the Linux Kernel Development (http://events.linuxfoundation.org/images/stories/pdf/als2012_gregkh.pdf).

It is my intent that this guide should be useful to a beginner as well as an experienced contributor and/or individuals interested in getting involved in the Linux kernel development. Experienced developers can chose to skip sections that go over basic

Zack Brown (/user/801501)

(/content/working-around-intel-hardware-flaws)

Working around Intel Hardware Flaws (/content/working-around-intel-hardware-flaws)

Zack Brown (/user/801501)

(/content/userspace-networking-dpdk)

Userspace Networking with DPDK (/content/userspace networking-dpdk)

Rami Rosen (/users/rami-rosen)

(/content/diff-u-speeding-un-speed-able)

diff -u: Speeding Up the Un-Speed-Up-able (/content/diff-u-speeding-un-speed-able)

Zack Brown (/user/801501)

testing and debugging.

This paper will discuss how to test and debug Linux kernel, tools, scripts and debug mechanisms that aid in regression and integration testing. In addition, this paper will go into details on how to use git bisect to isolate a patch that introduced a bug, and what to test before sending patches to the Linux Kernel Mailing List. I will use Linux PM as an example target area for the testing and debugging discussion. Even though this paper is Linux Kernel testing focused, the importance of testing is applicable to any software project.

Configuring Development and Test System

Let's get started. First order of business is finding a development system that suits your needs. x86-64 systems are a good choice for a basic development system, unless there is a need for a specific architecture and/or configuration.

The second step is to install distribution of your preference. I prefer Ubuntu, hence this document will have the details on how to configure a kernel development system running Ubuntu distribution. Please follow How to Ubuntu (http://howtoubuntu.org) for installing the Ubuntu release of your choice.

On development and test systems, it is a good idea to ensure there is ample space for kernels in the boot partition. Choosing whole disk install or setting aside 3 GB disk space for the boot partition is recommended.

Once the distribution is installed and the system is ready for development packages, enable root account and also enable sudo for your user account. The system might already have the build-essential package which is what you need to build Linux kernels on an x86_64 system. If build-essential is not already installed, run the following command to install it:

```
sudo apt-get install build-essential
```

At this point, you may install the following packages as well, so the system is ready for cross-compiling Linux kernels. Note that ncurses-dev is a required package to run make menuconfig.

```
sudo apt-get install binutils-multiarch
```

```
sudo apt-get install ncurses-dev
```

```
sudo apt-get install alien
```

Now let's install a few tools every Linux kernel developer need in his/her tool chest.

```
sudo apt-get install git
```

```
sudo apt-get install cscope
```

```
sudo apt-get install meld
```

```
sudo apt-get install gitk
```

If you would like to get the system configured for cross-compiling other supported architectures non-natively on your x86-64 system, please follow: Cross-compiling Linux kernel on x86 64 (http://linuxdriverproject.org/mediawiki/index.php/Cross-compiling_Linux_kernel_on_x86_64).

The Stable Kernel

Start by cloning the stable kernel git, building and installing the latest stable kernel. You can find information on the latest stable and mainline releases at The Linux Kernel Archive (https://www.kernel.org/).

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

The above step will create a new directory named linux-stable and populate it with the sources.

You can also download just the Linux Kernel source tar-ball instead of cloning the git, and then unpack the tar-ball.

```
tar xvf linux-3.x.y.tar.xz
```

Compiling and Installing Stable Kernel

If you cloned the stable git:

```
cd linux-stable
```

```
git checkout linux-3.x.y
```

or if you are using the tar-ball:

```
cd linux-3.x.y
```

Starting out with the distribution configuration file is the safest approach for the very first kernel install on any system. You can do so by copying the configuration for your current kernel from /boot.

```
cp /boot/config-3.x.y-z-generic .config
```

Run the following command to generate kernel configuration file based on the current configuration. You will be prompted to tune the configuration to enable new features and drivers that have been added since the Ubuntu snapshot the kernel from the mainline.

```
make oldconfig
```

Once this step is complete, it is time to compile the kernel:

```
make all
```

Once the kernel compilation is complete, install the new kernel:

```
sudo "make modules_install install"
```

The above command will install the new kernel and run update-grub to add the new kernel to the grub menu. Now it is time to reboot the system to boot the newly installed kernel. Before we do that, let's save logs from the current kernel to compare and look for regressions and new errors, if any:

```
dmesg -t > dmesg_current
```

```
dmesg -t -k > dmesg_kernel
```

```
dmesg -t -l emerg > dmesg_current_emerg
```

```
dmesg -t -l alert > dmesg_current_alert
```

```
dmesg -t -l crit > dmesg_current_alert
```

```
dmesg -t -l err > dmesg_current_err
```

```
dmesg -t -l warn > dmesg_current_warn
```

In general, dmesg should be clean with no emerg, alert, crit, and err level messages. If you see any of these, it might indicate some hardware and/or kernel problem.

A couple more important steps before trying out the newly installed kernel. There is no guarantee that the new kernel will boot. As a safe guard, please ensure that there is at least one good kernel installed. Change the default grub configuration file /etc/default/grub:

Enable printing early boot messages to vga using earlyprink=vga kernel boot option:
GRUB_CMDLINE_LINUX="earlyprink=vga"
Increase the GRUB_TIMEOUT value to 10 - 15 seconds, so grub pauses in menu allowing time to choose kernel to be boot:
Uncomment GRUB_TIMEOUT and set it to 10: GRUB_TIMEOUT=10
Comment out GRUB_HIDDEN_TIMEOUT and GRUB_HIDDEN_TIMEOUT_QUIET
Run update-grub to update the grun configuration in /boot
sudo update-grub

Now restart the system. Once the new kernel comes up, compare the saved dmesg from the old kernel with the new one and see if there are any regressions. If the newly installed kernel fails to boot, you will have to boot a good kernel and then investigate why the new kernel failed to boot.

Living in The Fast Lane

If you like driving in the fast lane and have the need for speed, clone the mainline kernel git or better yet the linux-next git. Booting and testing mainline and linux-next helps find and fix problems before the kernel is released.

Mainline:

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

linux-next:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git
```

Compiling and installing mainline and linux-next kernels is exactly same as the stable kernel. Please follow the instructions from previous sections.

Applying Patches

Linux kernel patch files are text files that contain the differences from the original source to the new. 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:

```
patch -p1 < file.patch
```

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

Either one will work, however, when a patch adds a new file and if it is applied using the patch command, git doesn't know about the new files and they will be treated as untracked files. "git diff" will not show the files in its output and "git status" will show the files as untracked.

For the most part, there are no issues with building and installing kernels, however, "git reset --hard" will not remove the newly added files and a subsequent git pull will fail. A couple of ways to tell git about the new files and have it track them, there by avoiding the above issues:

Option 1:
When a patch that adds new files is applied using the patch command, run "git clean" to remove untracked files, before running "git reset --hard". For example, git clean -dfx will force remove untracked directories and files, ignoring any standard ignore rules specified in the .gitignore file. You could include -q option to run git clean in quiet mode, if you don't care to know which files are removed.
Option 2:
An alternate approach is to tell git to track the newly added files by running "git apply --index file.patch". This will result in git applying the patch and adding the result to the index. Once this is done, git diff will show the newly added files in its output and git status will report the status correctly tagging these files as newly created files.

Basic Testing

Once a new kernel is installed, the next step is try to boot it and see what happens. Once the new kernel is up and running, check dmesg for any regressions. Run a few usage tests:

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

Examine Kernel Logs

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. lease note that new warn messages aren't as bad. New code at times 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

The following script runs the above dmesg commands and saves the output for comparing with older release dmesg files. It then runs diff commands against the older release dmesg files. Old release is a required input parameter. If one is not supplied, it will simply generate dmesg files and exit. Regressions indicate newly introduced bugs and/or bugs that escaped patch testing and integration testing in linux git trees prior to including the patch in a release. Are there any stack traces resulting from WARN_ON in the dmesg? These are serious problems that require further investigation.

- dmesg regression check script (http://linuxdriverproject.org/mediawiki/index.php/Dmesg_regression_check_script)

Stress Testing

Running 3 to 4 kernel compiles in parallel is a good overall stress test. Download a few Linux kernel gits, stable, linux-next etc.. Run timed compiles in parallel. Compare times with old runs of this test for regressions in performance. Longer compile times could be indicators of performance regression in one of the kernel modules. Performance problems are hard to debug. 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, file-systems, dma, and drivers.

```
time make all
```

Kernel Testing Tools

There are several tests under tools/testing that are included in the Linux kernel git. There is a good mix of automated and functional tests.

ktest suite

ktest is an automated test suite that can test builds, installs, and kernel boots. It can also run cross-compile tests provided the system has cross-compilers installed. ktest depends on flex and bison tools. Please consult the ktest documentation in tools/testing/ktest for details on how to run ktest. It is left to the reader as a self-study. A few resources that go into detail on how to run ktest:

- ktest-eLinux.org (http://elinux.org/Ktest#Git_Bisect_type)

tools/testing/selftests

Let's start with selftests. Kernel sources include a set of self-tests which test various sub-systems. As of this writing, breakpoints, cpu-hotplug, efivarfs, ipc, kcmp, memory-hotplug, mqueue, net, powerpc, ptrace, rcutorture, timers, and vm sub-systems have self-tests. In addition to these, user memory self-tests test user memory to kernel memory copies via test_user_copy module. The following is on how to run these self-tests:

Compile tests:
make -C tools/testing/selftests
Run all tests: (running some tests needs root access, login as root and run)
make -C tools/testing/selftests run_tests
Run only tests targeted for a single sub-system:

```
make -C tools/testing/selftests TARGETS=vm run_tests
```

tools/testing/fault-injection

Another test suite under tools/testing is fault-injection. failcmd.sh script runs a command to inject slab and page allocation failures. This type of testing helps validate how well kernel can recover from faults. This test should be run as root. The following is a quick summary of currently implemented fault injection capabilities. The list keeps growing as new fault injection capabilities get added. Please refer to the Documentation/fault-injection/fault-injection.txt for the latest.

failslab (default option)
injects slab allocation failures. kmalloc(), kmem_cache_alloc(), ...
fail_page_alloc
injects page allocation failures. alloc_pages(), get_free_pages(), ...
fail_make_request
injects disk IO errors on devices permitted by setting, /sys/block//make-it-fail or /sys/block///make-it-fail. (generic_make_request())
fail_mmc_request
injects MMC data errors on devices permitted by setting debugfs entries under /sys/kernel/debug/mmc0/fail_mmc_request

The capabilities and behavior of fault-injection can be configured. fault-inject-debugfs kernel module provides some debugfs entries for runtime. Ability to specify the error probability rate for faults, the interval between fault injection are just a couple of examples of the configuration choices fault-injection test supports. Please refer to the Documentation/fault-injection/fault-injection.txt for details. Boot options can be used to inject faults during early boot before debugfs becomes available. The following boot options are supported:

- failslab=
- fail_page_alloc=
- fail_make_request=
- mmc_core.fail_request=[interval],[probability],[space],[times]

The fault-injection infrastructure provides interfaces to add new fault-injection capabilities. The following is a brief outline of the steps involved in adding a new capability. Please refer to the above mentioned document for details:

define the fault attributes using DECLARE_FAULT_INJECTION(name);
Please see the definition of struct fault_attr in fault-inject.h for details.
add a boot option to configure fault attributes
This can be done using helper function setup_fault_attr(attr, str); Adding a boot option is necessary to enable the fault injection capability during early boot time.
add debugfs entries

Use the helper function fault_create_debugfs_attr(name, parent, attr); to add new debugfs entries for this new capability.

add module parameters

Adding module parameters to configure the fault attributes is a good option, when the scope of the new fault capability is limited to a single kernel module.

add a hook to insert failures

should_fail(attr, size); Upon should_fail() returning true, client code should inject a failure.

Applications using this fault-injection infrastructure can target a specific kernel module to inject slab and page allocation failures to limit the testing scope if need be.

Auto Testing Tools

There are several automated testing tools and test infrastructures that you can chose from based on your specific testing needs. This section is intended to be a brief overview and not a detailed guide on how to use each of these.

AuToTest (http://autotest.github.io)

Autotest is a framework for fully automated testing. It is designed primarily to test the Linux kernel, though it is useful for many other functions such as qualifying new hardware. It is an open source project under the GPL. Autotest works in server-client mode. Autotest server can be configured to initiate, run, and monitor tests on several target systems running the autotest client. Autotest client can be run manually on a target system or via the server. Using this framework, new test cases can be added. Please Autotest White Paper (https://github.com/autotest/autotest/wiki/WhitePaper) for more information.

Linaro Automated Validation Architecture (https://wiki.linaro.org/QA/AutomatedTestingFramework#Running_Tests)

LAVA-Test Automated Testing Framework is a framework to help with automated installation and executions of tests. For example, running LTP in LAVA framework can be accomplished with a few commands. Running lava-test tool to install LTP will automatically install any dependencies, download the source for the recent release of LTP, compile it, and install the binaries in a self-contained area so that they can be removed easily when user runs uninstall. At this point running lava-test run with ltp test option will execute LTP tests and save results with an unique id that includes the test name, time/date stamp of the test execution. These results are saved for future reference. This is a good feature to find regressions, if any, between test runs. Summary of commands to run as an example:

Show a list of tests supported by lava-test:
lava-test list-tests
Install a new test:
lava-test install ltp
Run the test:
lava-test run ltp
Check results:
lava-test results show ltp-timestamp.0
Remove tests:
lava-test uninstall ltp

Kernel Debug Features

Linux kernel includes several debugging features such as kmemcheck and kmemleak.

kmemcheck
kmemcheck is a dynamic checking tool that detects and warns about some uses of uninitialized memory. It serves the same function as Valgrind's memcheck which is a userspace memory checker, where as kmemcheck checks kernel memory. CONFIG_KMEMCHECK kernel configuration option enables the kmemcheck debugging feature. Please read the Documentation/kmemcheck.txt for information on how to configure and use this feature, and how to interpret the reported results.

kmemleak
kmemleak can be used to detect possible kernel memory leaks in a way similar to a tracing garbage collector. The difference between the tracing garbage collector and kmemleak is that the latter doesn't free orphan objects, instead it reports them in /sys/kernel/debug/kmemleak. A similar method of reporting and not freeing is used by the Valgrind's memcheck --leak-check to detect memory leaks in user-space applications. CONFIG_DEBUG_KMEMLEAK kernel configuration option enables the kmemleak debugging feature. Please read the Documentation/kmemleak.txt for information on how to configure and use this feature, and how to interpret the reported results.

Kernel Debug Interfaces

Linux kernel has support for static and dynamic debugging via configuration options, debug APIs, interfaces, and frameworks. Let's learn more about each of these starting with the static options.

Debug Configuration Options - Static

Linux kernel core and several Linux kernel modules, if not all, include kernel configuration options to debug. Several of these static debug options can be enabled at compile time. Debug messages are logged in dmesg buffer.

Debug APIs

An example of Debug APIs is DMA-debug which is desiged for debugging driver dma api usage errors. When enabled, it keeps track of dma mappings per device, detects unmap attempts on addresses that aren't mapped, and missing mapping error checks in driver code after dma map attempts. CONFIG_HAVE_DMA_API_DEBUG and CONFIG_DMA_API_DEBUG kernel configuration options enable this feature on architectures that provide the support. With the CONFIG_DMA_API_DEBUG option enabled, the Debug-dma interfaces are called from DMA API. For example, when a driver calls dma_map_page() to map a dma buffer, dma_map_page() will call debug_dma_map_page() to start tracking the buffer until it gets released via dma_unmap_page() at a later time. For further reading on Detecting silent data corruptions and memory leaks using DMA Debug API (http://events.linuxfoundation.org/sites/events/files/slides/Shuah_Khan_dma_map_error.pdf)

Dynamic Debug

Dynamic debug feature allows dynamically enabling/disabling pr_debug(), dev_dbg(), print_hex_dump_debug(), print_hex_dump_bytes() per-callsite. What this means is, a specific debug message can be enabled at run-time to learn more about a problem that is observed. This is great because, there is no need to re-compile the kernel with debug options enabled, then install the new kernel, only to find that the problem is no longer reproduciable. Once CONFIG_DYNAMIC_DEBUG is enabled in the kernel, dynamic debug feature enables a fine grain enable/disable of debug messages. /sys/kernel/debug/dynamic_debug/control is used to specify which pr_* messages are enabled. A quick summary of how to enable dynamic debug per call level, per module level is as follows:

Enable pr_debug() in kernel/power/suspend.c at line 340:

```
echo 'file suspend.c line 340 +p' > /sys/kernel/debug/dynamic_deb
ug/control
```

Enable dynamic debug feature in a module at module load time
Pass in dyndbg="plmft" to modprobe at the time module is being loaded.
Enable dynamic debug feature in a module to persist across reboots
create or change modname.conf file in /etc/modprobe.d/ to add dyndbg="plmft" option. However for drivers that get loaded from initramfs, changing modname.conf is insufficient for the dynamic debug feature to persist across reboot. For such drivers,

change grub to pass in module.dyndbg="+plmft" as a module option as a kernel boot parameter.

dynamic_debug.verbose=1 kernel boot option increases the verbosity of dynamic debug messages. Please consult the Documentation/dynamic-debug-howto.txt for more information on this feature.

Tracepoints

So far we talked about various static and dynamic debug features. Both static debug options and debug hooks such as the DMA Debug API are either enabled or disabled at compile time. Both of these options require a new kernel to be compiled and installed. The dynamic debug feature eliminates the need for a recompile, however the debug code is compiled in with a conditional variable that controls whether or not the debug message gets printed. It helps that the messages can be enabled at run-time, however, the conditional code is executed at run-time to determine if the message needs to be printed. Tracepoint code on the otherhand can be triggered to be included at run-time only when the tracepoint is enabled. In other words, tracepoint code is different in that, it is inactive unless it is enabled. When it is enabled, code is modified to include the tracepoint code. It doesn't add any conditional logic overhead to determine whether or not to generate a trace message.

Please read Tips on how to implement good tracepoint code (http://www.linuxjournal.com/content/july-2013-linux-kernel-news) for more insight into how tracing works.

Tracepoint mechanism

The tracepoints use jump-labels which is a code modification of a branch.

When it is disabled, the code path looks like:

```
[ code ]
nop
back:
[ code ]
return;
tracepoint:
[ tracepoint code ]
jmp back;
```

When it is enabled, the code path looks like: (notice how the tracepoint code appears in the code path below)

```
[ code ]
jmp tracepoint
back:
[ code ]
return;
tracepoint:
[ tracepoint code ]
jmp back;
```

Linux PM Sub-system Testing

Using debug, dynamic debug, and tracing, let's run a few suspend to disk PM tests. When system is suspended, kernel creates hibernation image on disk, suspends and uses the image to restore the systerm state at resume time.

Enable logging time it takes to suspend and resume each device

```
echo 1 > /sys/power/pm_print_times
```

Run suspend to disk test in reboot mode

```
echo reboot > /sys/power/disk
echo disk > /sys/power/state
```

Run suspend to disk test in shutdown mode - same as reboot, except requires powering on to resume

```
echo shutdown > /sys/power/disk
echo disk > /sys/power/state
```

Run suspend to disk test in platform mode - more extensive and tests BIOS suspend and resume paths e.g: ACPI methods will be invoked. This is the recommended mode for hibernation so BIOS is informed and aware of suspend/resume action.

```
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

Linux PM Sub-system Testing in Simulation Mode

The Linux PM sub-system provides five PM test modes to test hibernation in a simulated mode. These modes allow exercising the hibernation code in various layers of the kernel without actually suspending the system. This is useful when there is a concern that suspend might not work on a specific platform and help detect errors in a simulation similar to simulating flying a plane, so to speak.

freezer - test the freezing of processes

```
echo freezer > /sys/power/pm_test
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

devices - test the freezing of processes and suspending of devices

```
echo devices > /sys/power/pm_test
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

platform - test the freezing of processes, suspending of devices and platform global control methods(*)

```
echo platform > /sys/power/pm_test
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

processors - test the freezing of processes, suspending of devices, platform global control methods(*) and the disabling of non-boot CPUs

```
echo processors > /sys/power/pm_test
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

core - test the freezing of processes, suspending of devices, platform global control methods, the disabling of non-boot CPUs and suspending of platform/system devices. Note: this mode is tested on ACPI systems.

```
echo core > /sys/power/pm_test
echo platform > /sys/power/disk
echo disk > /sys/power/state
```

Linux PM Sub-system Trace Events

PM sub-system supports several tracepoints and trace events that can be enabled to trigger during run-time. I will give an overview on how to enable couple of these trace events and where to find the trace information they generate:

Enabling PM events at run-time:

```
cd /sys/kernel/debug/tracing/events/power
echo 1 > cpu_frequency/enable
cat /sys/kernel/debug/tracing/set_event
less /sys/kernel/debug/tracing/trace
```

Enabling events at boot-time kernel trace parameter with a kernel boot option:

```
trace_event=cpu_frequency
```

For more information on Linux PM testing, please consult the Documentation/power/basic-pm-debugging.txt and other documents in the same directory.

git bisect

git bisect is an invaluable and powerful tool to isolate an offending commit. I will go over very basic git bisect steps.

This is how the process works:

```
git bisect start
git bisect bad   # Current version is bad
git bisect good v3.14-rc6       # last good version
```

Once, one bad and one good version are specified, git bisect will start bisecting by pulling in commits between the good version and the bad. Once a set of commits are pulled in, compile the kernel, install, test, and tag the version good or bad. This process repeats until the selected commits are tested and tagged as good or bad. There can be several kernel versions to test. When the last version is tested, git bisect will flag a commit that is bad. The following useful git-bisect command can aid in using git-bisect process:

See step by step bisect progress

```
git bisect log
```

Reset git bisect can be used in case of mistakes in tagging, save git log output and replay prior to reset

```
git bisect reset
```

Replay a git-bisect log

```
git bisect replay git_log_output
```

git bisect can be run on a section of kernel source tree if the problem is clearly in that area. For example, when debugging a problem in radeon driver, running git bisect on drivers/drm/radeon will limit the scope of bisect to just the commits to drivers/drm/radeon driver.

Start git bisect on a section of a kernel tree

```
git bisect start drivers/drm/radeon
```

Linux Kernel Patch Testing

Are you try your hands on writing a kernel patch? This section will go over how to test a new patch before sending it to the Linux mailing list. Further more, we will also talk about how to send it.

Once the code is ready, compile it. Save the make output to a file to see if the new code introduced any new warnings. Address warnings, if any. Once the code compiles cleanly, install the compiled kernel and boot test. If it boots successfully, make sure there are no new errors in the dmesg, comparing it with the previous kernel dmesg. Run a few usage and stress tests. Please refer to the testing content we discussed earlier in this paper. If the patch is for fixing a specific bug, make sure the patch indeed fixes the bug. If the patch fixes the problem, make sure, other module regression tests pass. Identify regression tests for the patched module and run them. When a patch touches other architectures, cross-compile build testing is recommended. Please check the following in the source git as a reference to identify tests.

- linux_git/Documentation
- linux_git/tools/testing
- Cross-compiling reference: Cross-compiling Linux Kernels on x86_64: A tutorial on How to Get Started (http://events.linuxfoundation.org/sites/events/files/slides/Shuah_Khan_cross_compile_linux.pdf)

Once you are satisfied with the patch testing, it is time to commit the change and generate the patch. Make sure the commit message describes the change made very clearly. It is important that the maintainer and other developers can understand what this change is all about. Once patch is ready, run scripts/checkpatch.pl on the

generated patch. Address checkpatch errors and/or warnings, if any. Regenerate and repeat until the patch passes the checkpatch test. Unless the checkpatch errors are minor whitespace type, re-test the patch. Apply the patch to another instance of the kernel git to make sure patch applies cleanly.

Now you are ready to send the patch. Please run the scripts/get_maintainer.pl to identify whom the patch should be sent to. Please remember that the patch needs to be sent as a plain text, not as an attachment. Please make sure your email client can send plain text messages. Email the patch to yourself to test your client settings. Run checkpatch and apply the received patch. If these two steps pass, then you are ready to send the patch to the Linux Kernel Mailing List. git send-email is the safest way to send patches to avoid email client complications. Please make sure your .gitconfig includes sendemail with a valid smtpserver. Please consult git manpage for details.

Please refer to the following documentation in the kernel sources for rules and guidelines on sending patches:

- linux_git/Documentation/applying-patches.txt
- linux_git/Documentation/SubmitChecklist
- linux_git/Documentation/SubmittingDrivers
- linux_git/Documentation/SubmittingPatches
- linux_git/Documentation/stable_kernel_rules.txt
- linux_git/Documentation/stable_api_nonsense.txt

The following is a list of additional testing guides and resources:

- USB Testing on Linux (http://www.linux-usb.org/usbtest/)
- Linux Kernel Tester's Guide Chapter2 (http://kernelnewbies.org/Linux_Kernel_Tester%27s_Guide_Chapter2)
- Linux Kernel Tester's Guide (http://www.kerneltravel.net/downloads/tester_guide.pdf)
- Testing resources at eLinux.org (http://elinux.org/Test_Systems)
- eLinux Debugging Portal (http://elinux.org/Debugging_Portal)

Kernel test suites and projects

In addition to the testing resources we discussed so far, there are projects both open source and initiated by hardware vendors that are worth a mention. Each of these projects focus on specific areas of the kernel and in some cases a specific space such as, embedded or enterprise where the kernel is used. We will look at a few in this section.

Linux Test Project (http://ltp.sourceforge.net/documentation/how-to/ltp.php) (LTP) test suite is a collection of tools to test reliability, robustness, and stability of Linux kernel and related features. This test suite can be customized by adding new tests and the LTP project welcomes contributions. runltp script tests the following sub-systems by default:

- filesystem stress tests
- disk I/O tests
- memory management stress tests
- ipc stress
- scheduler tests
- commands functional verification tests
- system call functional verification tests

LTP-DDT (http://processors.wiki.ti.com/index.php/LTP-DDT) is an LTP based test application wth a reduced focus to test embedded device drivers.

Linux Driver Verification (http://linuxtesting.org/project/ldv) project's goals are to improve the quality of Linux device drivers, develop an integrated platform for device drivers verification, and adopt latest research outcome to enhance quality of verification tools.

Compliance Testing

If you ever had to port applications from one Unix variant to another, you would understand the importance of the Linux Standard Base (LSB) (http://www.linuxfoundation.org/collaborate/workgroups/lsb) and LSB compliance test suite. The LSB is a Linux Foundation workgroup created to reduce the costs of supporting Linux platform, by reducing the differences between various Linux distributions and ensuring application portability between distributions. If anything, divergence in the Unix world taught us that it is vital to avoid it in the Linux world. This is exactly the reason why you can take an rpm convert it to deb and install and run it, and how sweet is that.

Static Analysis and Tools

Static analysis tools analyze the code without executing it, hence the name static analysis. There are a couple of static analysis tools that are sepcifically written for analyzing the Linux kernel code base. Sparse is a static type-checking program written specifically for the Linux kernel, by Linus Torvalds. Sparse is a semantic parser. It

creates a sematic prase tree to validate C semantics. It performs lazy type evaluation. Kernel build system has support for sparse and provides a make option to compile the kernel with sparse checking enabled.

Run sparse on all kernel C files that would get re-compiled:

```
make C=1 allmodconfig
```

Run sparse on all kernel C files even when they don't need a re-compile:

```
make C=2 allmodconfig
```

Sparse resources:

- Sparse Archive (http://codemonkey.org.uk/projects/git-snapshots/sparse/)
- Sparse How To (http://tree.celinuxforum.org/CelfPubWiki/Sparse)

Smatch analyzes source to detect programming logic errors. It can detect logic errors such as, attempts to unlock already unlocked spinlock. It is actively used to detect logic errors in the Linux kernel sources.

Run smatch on Linux kernel:

```
make CHECK="~/path/to/smatch/smatch -p=kernel" C=1 bzImage module
s | tee warns.txt
```

Please follow instructions on how to get smatch from smatch git repo and compile. Smatch is work in progress, instructions keep changing.

- Smatch (http://smatch.sourceforge.net/)

So what do we do about all of these semantic and logic problems found by Sparse and Smatch? Some of these problems are isolated to a routine and/or a module which can be fixed easily. However, some of these semantic issues are global in nature due to cut and paste of code. In some cases when interfaces get obsoleted or changed slightly, a mass change to update several source files becomes necessary. This is where Coccinelle comes in to rescue. Coccinelle is a program matching and transformation engine which provides the language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code. Coccinelle was initially targeted towards performing collateral evolutions in Linux.
For example, foo(int) interfaces changes to foo(int, char *) with an optional second input parameter which can be a null. All usages of foo() will need to be updated to the new convention, which will be a very laborious task. Using Cocinelle, this task becomes easier with a script that looks for all instances of foo(parameter1) and replacing them with foo(parameter1, NULL). Once this task is done, all instances of foo() can be examined to see if passing in NULL value for parameter2 is a good assumption. For