

Coding Challenge Summary: Configure, Install, and Test the Most Stable Kernel Version

TOC :

1. Configuring the kernel
2. Build and Install the kernel
3. Displaying a custom message on boot
4. Running Kselftests and Kunit tests
5. Additional step: Using KernelCI for testing cross-architecture compatibility

Step 1: Configuring the Kernel

1.Previous Kernel Version:6.11.2



```
krrish@kali: /  
$ uname -r  
6.11.2-arm64  
$ neofetch  
krrish@kali  
OS: Kali GNU/Linux Rolling aarch64  
Host: VMware20.1.1  
Kernel: 6.11.2-arm64  
Uptime: 1 hour, 42 mins  
Packages: 2773 (dpkg)  
Shell: zsh 5.9  
Resolution: 2560x1600  
DE: Xfce 4.18  
WM: Xfwm4  
WM Theme: Kali-Dark-xHIDPI  
Theme: Kali-Dark [GTK2], adw-gtk3-dark [GTK3]  
Icons: Flat-Remix-Blue-Dark [GTK2/3]  
Terminal: qterminal  
Terminal Font: FiraCode 10  
CPU: (4)  
GPU: 00:0f.0 VMware Device 0406  
Memory: 691MiB / 3917MiB
```

2. Most recent kernel version as of January 19, 2024, is 6.12.10

3. Downloading and Extracting the Kernel Tar File.



```
krrish@kali: [~/Downloads]  
$ cd linux-6.12.10  
$ ls  
arch  build.log  COPYING  crypto  drivers  include  io_uring  Kbuild  kernel  LICENSES  Makefile  net  rust  scripts  sound  usr  
block  certs  CREDITS  Documentation  fs  init  ipc  Kconfig  lib  MAINTAINERS  mm  README  samples  security  tools  virt
```

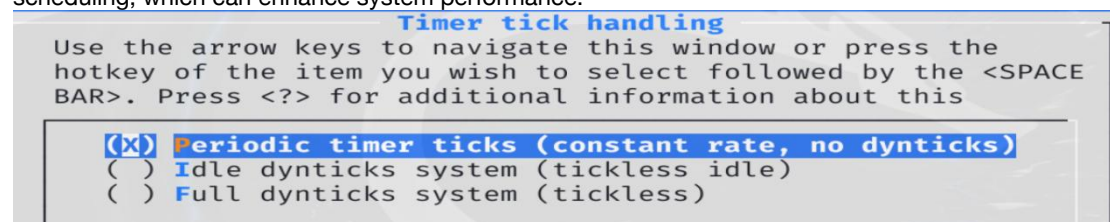
4. Configuring the Kernel to Meet My Requirements

- > cp /boot/config-\$(uname -r) .config (was used to copy the current kernel configuration to .config)
- > make olddefconfig (was used to apply default values for new kernel options in version 6.12.10 , faster setup compared to `make oldconfig`)
- > make menuconfig (was used to modify the kernel configuration according to my specific requirements)

5. Key Changes Made:

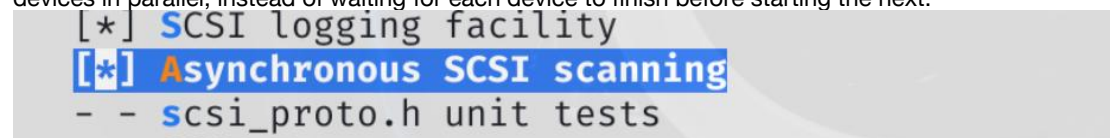
Processor Features:

-> Enabled periodic timer tick that improves system responsiveness by reducing variability in task scheduling, which can enhance system performance.



Device Drivers:

-> Enabled async SCSI scanning as normally, when the kernel scans for SCSI devices, it performs this operation sequentially. Asynchronous SCSI scanning allows the kernel to start scanning multiple devices in parallel, instead of waiting for each device to finish before starting the next.



-> Disabled: RAID and LVM support, as RAID or LVM are not useful unless you plan to manage multiple disks for data redundancy, scaling, or advanced disk management, and on my local PC, I

don't plan to do it. Since I'm working in a virtualized environment with VMware, the storage management is handled by the hypervisor, making these features unnecessary.

```
<M> Serial ATA and Parallel ATA drivers (libata) —→
[*] Multiple devices driver support (RAID and LVM) —
<M> Generic Target Core Mod (TCM) and ConfigFS Infrastructure —→
```

Kernel Hacking:

1. Enabled Kunit support for running kunit tests once I boot into the new kernel.

```
— KUnit - Enable support for unit tests
[*] KUnit - Enable /sys/kernel/debug/kunit debugfs representation
[*] Enable KUnit tests which print BUG stacktraces (NEW)
<M> KUnit test for KUnit
<M> Example test for KUnit
<M> All KUnit tests with satisfied dependencies
[*] Default value of kunit.enable (NEW)
```

2. Enabled KCOV as its useful in kunit tests to monitor and analyze which parts of the kernel code are executed during a test.

```
[*] Code coverage for fuzzing
[*] Enable comparison operands collection by KCOV
[*] Instrument all code by default (NEW)
(0x/0000) Size of interrupt coverage collection area in words (NEW)
```

Step 2: Build and Install the Kernel

-Ran make -j\${nproc} to compile the kernel.

-Since my system arch is ARM64, I didn't see the usual success message for x86 architecture(Kernel: arch/x86/boot/bzImage ready), but the kernel image for ARM64 was successfully built.

-Proof: The "Image" file (kernel image for ARM64) containing the compressed kernel binary code was created successfully.

Build successful:

```
KSYSMS .tmp_vmlinux0.kallsyms.S
AS .tmp_vmlinux0.kallsyms.o
LD .tmp_vmlinux1
NM .tmp_vmlinux1.syms
KSYSMS .tmp_vmlinux1.kallsyms.S
AS .tmp_vmlinux1.kallsyms.o
LD .tmp_vmlinux2
NM .tmp_vmlinux2.syms
KSYSMS .tmp_vmlinux2.kallsyms.S
AS .tmp_vmlinux2.kallsyms.o
LD vmlinux
NM System.map
SORTTAB vmlinux
OBJCOPY arch/arm64/boot/Image

LD [M] net/psample/psample.ko
LD [M] net/ife/ife.ko
LD [M] net/openvswitch/openvswitch.ko
LD [M] net/openvswitch/vport-vxlan.ko
LD [M] net/openvswitch/vport-geneve.ko
LD [M] net/openvswitch/vport-gre.ko
LD [M] net/vmw_vsock/vsock.ko
LD [M] net/vmw_vsock/vsock_diag.ko
LD [M] net/vmw_vsock/vmw_vsock_virtio_transport.ko
LD [M] net/vmw_vsock/vmw_vsock_virtio_transport_common.ko
LD [M] net/vmw_vsock/hv_vsock.ko
LD [M] net/vmw_vsock/vsock_loopback.ko
LD [M] net/nsh/nsh.ko
LD [M] net/hsr/hsr.ko
LD [M] net/qrtr/qrtr.ko
LD [M] net/qrtr/qrtr-smc.ko
LD [M] net/qrtr/qrtr-mhi.ko

(krrish@kali)~/Downloads/Linux-6.12.10]
(krrish@kali)~/Downloads/linux-6.12.10]
$ ls arch/arm64/boot/
dts Image Image.gz install.sh Makefile
(krrish@kali)~/Downloads/linux-6.12.10]
$ find . -name "Image" -print
./arch/arm64/boot/Image
```

Module installation successful:

```

SIGN      /lib/modules/6.12.10/kernel/net/vmw_vsock/vmw_vsock_virtio_transport.ko
INSTALL   /lib/modules/6.12.10/kernel/net/vmw_vsock/vmw_vsock_virtio_transport_common.ko
SIGN      /lib/modules/6.12.10/kernel/net/vmw_vsock/vmw_vsock_virtio_transport_common.ko
INSTALL   /lib/modules/6.12.10/kernel/net/vmw_vsock/hv_sock.ko
SIGN      /lib/modules/6.12.10/kernel/net/vmw_vsock/hv_sock.ko
INSTALL   /lib/modules/6.12.10/kernel/net/vmw_vsock/vsock_loopback.ko
SIGN      /lib/modules/6.12.10/kernel/net/vmw_vsock/vsock_loopback.ko
INSTALL   /lib/modules/6.12.10/kernel/net/nsh/nsh.ko
SIGN      /lib/modules/6.12.10/kernel/net/nsh/nsh.ko
INSTALL   /lib/modules/6.12.10/kernel/net/hsr/hsr.ko
SIGN      /lib/modules/6.12.10/kernel/net/hsr/hsr.ko
INSTALL   /lib/modules/6.12.10/kernel/net/qrtr/qrtr.ko
SIGN      /lib/modules/6.12.10/kernel/net/qrtr/qrtr.ko
INSTALL   /lib/modules/6.12.10/kernel/net/qrtr/qrtr-sm-d.ko
SIGN      /lib/modules/6.12.10/kernel/net/qrtr/qrtr-sm-d.ko
INSTALL   /lib/modules/6.12.10/kernel/net/qrtr/qrtr-mhi.ko
SIGN      /lib/modules/6.12.10/kernel/net/qrtr/qrtr-mhi.ko
DEPMOD    /lib/modules/6.12.10

```

—(krrish@kali)~[~/Downloads/linux-6.12.10]

Kernel Installation successful:

```

—(krrish@kali)~[~/Downloads/linux-6.12.10]
$ sudo make install
INSTALL /boot
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 6.12.10 /boot/vmlinuz-6.12.10
update-initramfs: Generating /boot/initrd.img-6.12.10
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 6.12.10 /boot/vmlinuz-6.12.10
Generating grub configuration file ...
Found theme: /boot/grub/themes/kali/theme.txt
Found background image: /usr/share/images/desktop-base/desktop-grub.png
Found linux image: /boot/vmlinuz-6.11.2-arm64
Found initrd image: /boot/initrd.img-6.11.2-arm64
Found linux image: /boot/vmlinuz-6.12.10
Found initrd image: /boot/initrd.img-6.12.10
Found linux image: /boot/vmlinuz-6.12.10.old
Found initrd image: /boot/initrd.img-6.12.10
Warning: os-prober will not be executed to detect other bootable partitions.
Systems on them will not be added to the GRUB boot configuration.
Check GRUB_DISABLE_OS_PROBER documentation entry.
Adding boot menu entry for UEFI Firmware Settings ...
done

```

Step3: Display Custom Message and Reboot to New Version

To display a custom boot message, I modified the /etc/grub.d/40_custom file that allowed me to add a custom option in grub menu that displayed the custom message while booting up, since I'm using local storage to access the files and resources and not a PXE or network boot, the mentioned instruction to add the message in tftpbboot folder didn't work.

Custom Message:

"Welcome to your custom Linux kernel! Built by Krrish Sehgal on Sun Jan 19 12:07:35 PM EST 2025 for ARM64 architecture with kernel modifications: 1. Periodic timer ticks: On, 2. Async SCSI: On, 3. RAID and LVM support: Off, 4. Kunit support: On"

I appended changes in my message to easily track my modifications, making it simpler to compare configurations in the future.



Succesfully booted to the new kernel version:

```

(krrish@kali)-[~]
$ uname -mrs
Linux 6.12.10 aarch64

(krrish@kali)-[~]
$ neofetch

               krrish@kali
-----
OS: Kali GNU/Linux Rollin
Host: VMware20,1 1
Kernel: 6.12.10
Uptime: 53 secs
Packages: 2773 (dpkg)
Shell: zsh 5.9
Resolution: 2560x1600
DE: Xfce 4.18
WM: Xfwm4
WM Theme: Kali-Dark-xHiDP
Theme: Kali-Dark [GTK2],
Icons: Flat-Remix-Blue-Da
Terminal: qterminal
Terminal Font: FiraCode 1
CPU: (4)
GPU: 00:0f.0 VMware Devic
Memory: 633MiB / 3921MiB

```

Step4: Testing:

1. Kselftests:

I used this because it's a comprehensive suite of tests that can help to evaluate kernel functionality from a userspace perspective (syscalls, device drivers, filesystems, etc). The successful execution of majority of kselftests indicated that the configured and built kernel is functioning as expected within the scope of the tested features. It signifies correctness in key areas such as memory management, file I/O, debugging interfaces, and some security mechanisms.

Some test suites under kselftests resulted in 'not ok', like breakpoints failed because on vmware, low-power suspend state isn't supported, Cgroups suite because Cgroups were not frozen during migration tests, which is not really important as this mainly affects workloads using the freezer subsystem (related to suspending/resuming a group of processes).

(Also submitted the full output of the kselftests in /Tests/kself-tests.txt)

However, kselftest does not cover every aspect of the kernel and primarily focuses on exposed functionality.

```

# set max_comp_streams to zram device(s)
# The device attribute max_comp_streams was deprecated in 4.7
# set disk size to zram device(s)
# /sys/block/zram0/disksize = '1048576'
# zram set disksize: OK
# set memory limit to zram device(s)
# /sys/block/zram0/mem_limit = '1M'
# zram set memory limit: OK
# make swap with zram device(s)
# done with /dev/zram0
# zram making zram mkswap and swapon: OK
# zram swapoff: OK
# zram cleanup
# zram02 : [PASS]
ok 1 selftests: zram: zram.sh
make[1]: Leaving directory '/home/krrish/Downloads/linux-6.12.10/tools/testing/selftests/zram'

(krrish@kali)-[~/.../linux-6.12.10/tools/testing/selftests]

```

2. KunitTests:

So I used, Kunit tests, which are essential to cover testing the missing aspects of kselftests and other core internal features of the kernel.

Since kunit_tool is written for um architecture instead of modifying the scripts I used this docs “ https://docs.kernel.org/dev-tools/kunit/run_manual.html ” that helped me get a work around, and since I enabled the kunit tests in the configuration menu, so it ran automatically in the background AFTER boot and every directory listed below had a results file in which the specific kernel tests have ran successfully isolating specific functionalities.

```

root@kali:~/sys/kernel/debug/kunit# ls
apparmor_policy_unpack      clk-orphan-transparent-multiple-parent-mux-test  example      kunit_executor_test      of_overlay_apply_kunit
binfmt_elf                 clk-orphan-transparent-single-parent-test        example_init  kunit_fault              overflow
bitfields                 clk-orphan-two-level-root-last-test             exec         kunit-log-test          property-entry
bits-test                 clk-range-maximize-test                         fortify       kunit_platform_device   qos-kunit-test
checksum                  clk-range-minimize-test                        'Handshake API tests'  kunit_platform_driver  rational
clk-fd-approximation      clk-range-test                                  hashtable    kunit-resource-test     regmap
clk_fixed_rate            clk_register_clk_parent_data_device             hlist        kunit-status            resource
clk_fixed_rate_of        clk_register_clk_parent_data_of                 hw_breakpoint kunit-try-catch-test    rtc_lib_test_cases
clk_fixed_rate_parent    clk-single-parent-mux-test                      input_core   landlock_fs             scsi_lib
clk-gate-hiword-test     clk-uncached-test                               iov_iter     list-kunit-test         siphash
clk-gate-invert-test     cmdline                                          is_signed_type list_sort               slub_test
clk-gate-is_enabled-test  compat_binfmt_elf                               kfence       list_sort               stackinit
clk-gate-register-test   cpumask                                         klist        math-int_pow            string
clk-gate-test            daemon                                           klist        mptcp-crypto            string_helpers
clk-leaf-mux-set-rate-parent  daemon-operations                             kprobes_test mptcp-token             string-stream-test
clk-multiple-parents-mux-test  daemon-sysfs                                   kunit-assert mptcp-token             sysctl_test
clk-mux-no-reparent       dev-addr-list-test                             kunit_current net_core                 time_test_cases
clk-mux-notifier          dev-addr-list-test                             kunit_device of_dtb                   usercopy

```

Screenshots of some of the main unit tests with their significance is provided below along with the test files being attached in the root folder submitted (/Tests/kunit-tests).

1. Internal data structures tests[ALL PASSED]: list-kunit-test,list_sort,klist (Test on operations on kernel data strucutres like linked lists etc)

```

ok 21 list_test_list_cut_position
ok 22 list_test_list_cut_before
ok 23 list_test_list_splice
ok 24 list_test_list_splice_tail
ok 25 list_test_list_splice_init
ok 26 list_test_list_splice_tail_init
ok 27 list_test_list_entry
ok 28 list_test_list_entry_is_head
ok 29 list_test_list_first_entry
ok 30 list_test_list_last_entry
ok 31 list_test_list_first_entry_or_null
ok 32 list_test_list_next_entry
ok 33 list_test_list_prev_entry
ok 34 list_test_list_for_each
ok 35 list_test_list_for_each_prev
ok 36 list_test_list_for_each_safe
ok 37 list_test_list_for_each_prev_safe
ok 38 list_test_list_for_each_entry
ok 39 list_test_list_for_each_entry_reverse
# module: list_test
# list-kunit-test: pass:39 fail:0 skip:0 total:39
# Totals: pass:39 fail:0 skip:0 total:39
ok 1 list-kunit-test

```

2. Driver Tests[ALL PASSED]:

hw_breakpoint: (Tests for hardware breakpoints),rtc_lib_test_cases(tests real time clock library),scsi_lib(Tests for the SCSI library).

```

Expected IS_ERR(bp) to be false, but is true
not ok 7 test_two_tasks_on_one_cpu
# test_two_tasks_on_one_all_cpus: ASSERTION FAILED at kernel/events/hw_breakpoint_test.c:70
Expected IS_ERR(bp) to be false, but is true
not ok 8 test_two_tasks_on_one_all_cpus
ok 9 test_task_on_all_and_one_cpu # SKIP Requires breakpoint slots: 3 > 2
# module: hw_breakpoint_test
# hw_breakpoint: pass:2 fail:5 skip:2 total:9
# Totals: pass:2 fail:5 skip:2 total:9
not ok 1 hw_breakpoint

```

3. Networking Tests[ALL PASSED]: net_core(Tests for the networking core subsystem), mptcp-*

```

ok 9 gso_by_frags
# gso_test_func: pass:9 fail:0 skip:0 total:9
ok 1 gso_test_func
KTAP version 1
# Subtest: ip_tunnel_flags_test_run
ok 1 compat
ok 2 conflict
ok 3 new
# ip_tunnel_flags_test_run: pass:3 fail:0 skip:0 total:3
ok 2 ip_tunnel_flags_test_run
# module: net_test
# net_core: pass:2 fail:0 skip:0 total:2
# Totals: pass:12 fail:0 skip:0 total:12
ok 1 net_core

```

4. Security Tests[ALL PASSED]: apparmor_policy_unpack(tests for apparmour that lays out policies for access control for individual apps running on linux), landlock_fs, fortify(Checks for buffer overflow protection).

```

ok 21 policy_unpack_test_unpack_u16_chunk_out_of_bounds_2
ok 22 policy_unpack_test_unpack_u32_with_null_name
ok 23 policy_unpack_test_unpack_u32_with_name
ok 24 policy_unpack_test_unpack_u32_out_of_bounds
ok 25 policy_unpack_test_unpack_u64_with_null_name
ok 26 policy_unpack_test_unpack_u64_with_name
ok 27 policy_unpack_test_unpack_u64_out_of_bounds
ok 28 policy_unpack_test_unpack_X_code_match
ok 29 policy_unpack_test_unpack_X_code_mismatch
ok 30 policy_unpack_test_unpack_X_out_of_bounds
# module: apparmor_policy_unpack_test
# apparmor_policy_unpack: pass:30 fail:0 skip:0 total:30
# Totals: pass:30 fail:0 skip:0 total:30
ok 1 apparmor_policy_unpack

```

5. Clock Management Tests[ALL PASSED]: All clk-* tests, as they cover core kernel clock APIs(recall, this also validates my personal kernel configuration of using periodic timer ticks).

```
KTAP version 1
1..1
  # Subtest: clk-test
  1..4
  ok 1 clk_test_get_rate
  ok 2 clk_test_set_get_rate
  ok 3 clk_test_set_set_get_rate
  ok 4 clk_test_round_set_get_rate
  # module: clk-test
# clk-test: pass:4 fail:0 skip:0 total:4
# Totals: pass:4 fail:0 skip:0 total:4
ok 1 clk-test
```

6. Low-level kernel operations tests[ALL PASSED]: property-entry(validate that kernel interacts properly with hardware configuration data), overflow(Overflow tests protect against critical bugs that could compromise the entire system).

```
ok 17 overflow_allocation_test
# overflow_size_helpers_test: 43 overflow size helper tests finished
ok 18 overflow_size_helpers_test
# overflows_type_test: 658 overflows_type() tests finished
ok 19 overflows_type_test
# same_type_test: 0 __same_type() tests finished
ok 20 same_type_test
# castable_to_type_test: 103 castable_to_type() tests finished
ok 21 castable_to_type_test
ok 22 DEFINE_FLEX_test
# module: overflow_kunit
# overflow: pass:22 fail:0 skip:0 total:22
# Totals: pass:22 fail:0 skip:0 total:22
ok 1 overflow
```

Additional Step: Setting up Kernel-CI architecture locally and testing the built Image:

Since KernelCI could be integrated into my build process to perform checks on my custom build for different architectures by uploading my custom-built Image binaries to it, I attempted to set up the KernelCI architecture locally. For this, I used three official KernelCI repositories.

1. kernelci-frontend
2. kernelci-backend
3. kernelci-docker

All of these repositories had deprecated dependencies and outdated Dockerfiles, so I submitted some pull requests to update the dependencies and Dockerfiles to ensure the services under KernelCI Docker run seamlessly:

1. <https://github.com/kernelci/kernelci-docker/pull/29>
2. <https://github.com/kernelci/kernelci-frontend/pull/155>
3. <https://github.com/kernelci/kernelci-backend/pull/306>

Ongoing Issues:

While I resolved some major issues in my PRs. Some services, such as the proxy under kernelci-docker, still require further work before they can run seamlessly. I have also opened issues to streamline the setup process:

<https://github.com/kernelci-docker/issues/28>.

Next Steps:

- Address pending issues related to kernelci-docker to achieve a smoother setup.
- Set up and integrate LAVA for automated hardware-based boot testing. I plan to contribute, if I face any challenges in kernelci/lava-docker to enhance its compatibility and utility.

After setting this up, it will help me automate boot testing on a variety of supported platforms (virtualized and physical hardware). This will ensure my kernel image is not only built successfully but also boots correctly across diverse environments.

I believe this work will help the community and also prepare me for working effectively on kernel development and continuous integration projects, as part of this mentorship and beyond.