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

TOC :

Step 1: Configuring the Kernel

1.Previous Kernel Version:6.11.2

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

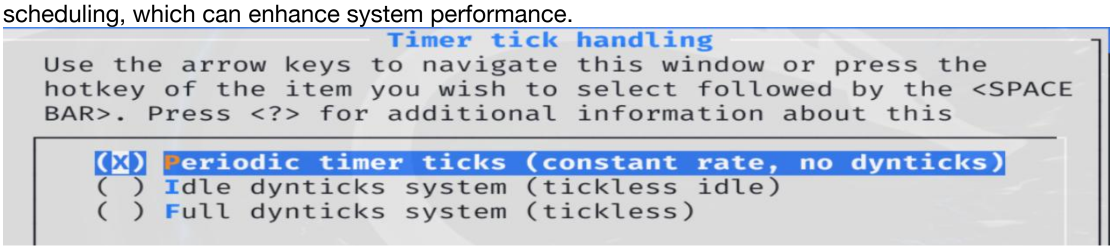
3. Downloading and Extracting the Kernel Tar File.

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:

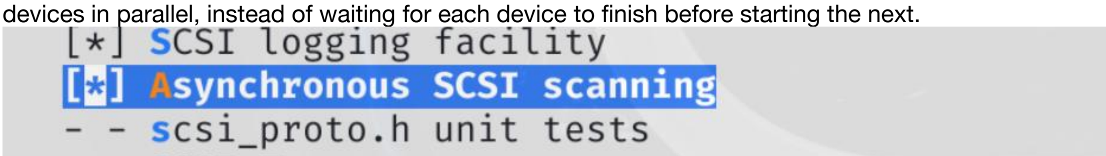
Processor Features:

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

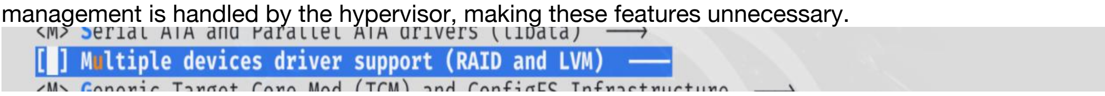


Device Drivers:

-> 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.

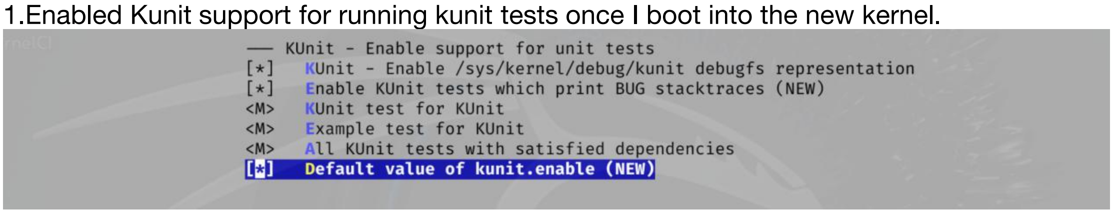


-> Disabled: RAID and LVM support, as RAID or LVM are not useful unless you plan to manage multiple disks for data redundancy, scaling, or advanced disk management, and on my local PC, I don’t plan to do it. Since I'm working in a virtualized environment with VMware, the storage management is handled by the hypervisor, making these features unnecessary.

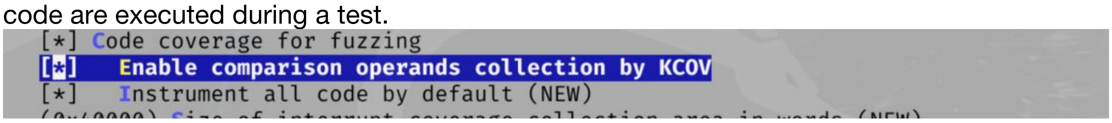


Kernel Hacking:

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



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



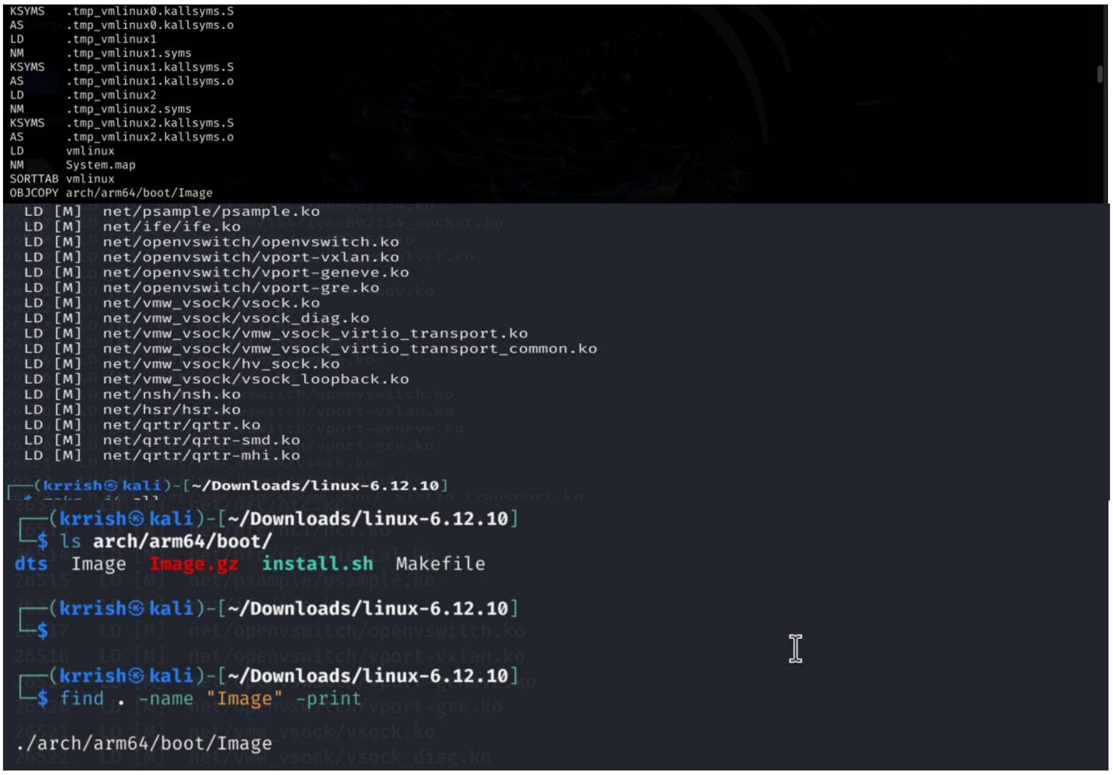
Step 2: Build and Install the Kernel

-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:



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-smd.ko
  SIGN    /lib/modules/6.12.10/kernel/net/qrtr/qrtr-smd.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/vmlinux-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/vmlinux-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/vmlinux-6.12.10
Found initrd image: /boot/initrd.img-6.12.10
Found linux image: /boot/vmlinux-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
```

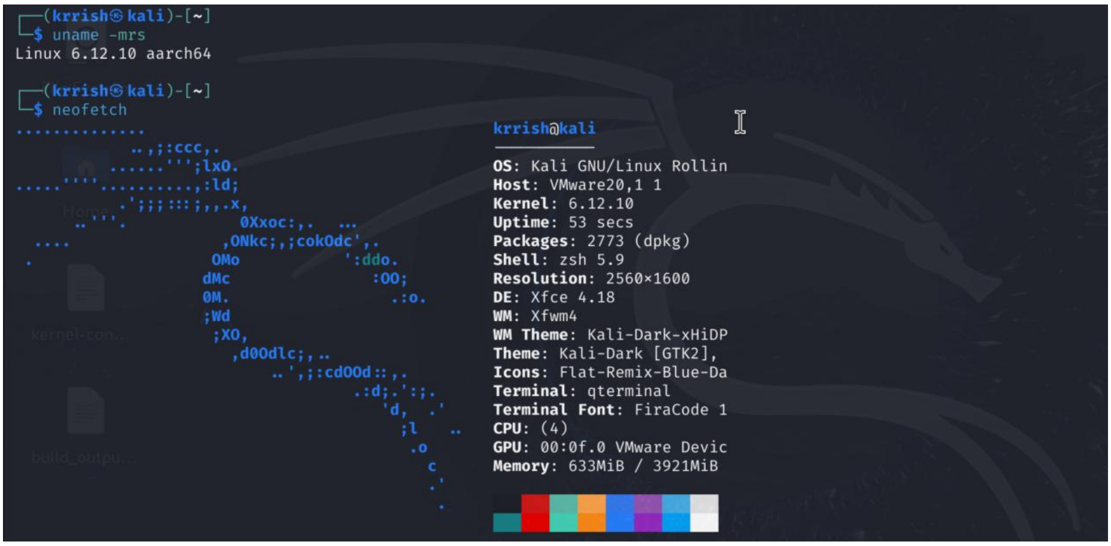
Step3: Display Custom Message and Reboot to New Version

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 tftpboot 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.

Succefully booted to the new kernel version:

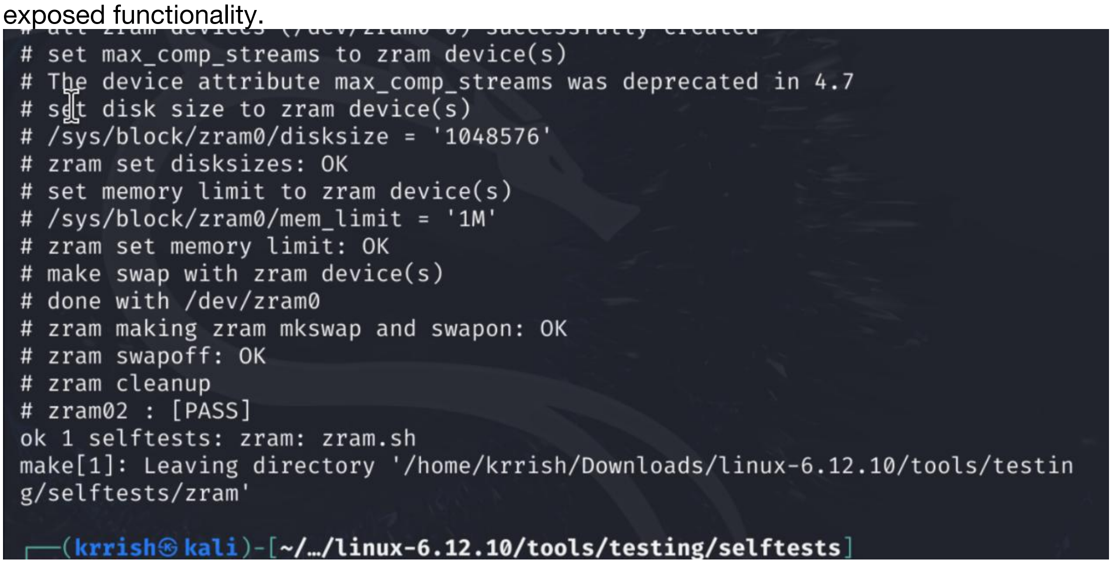


Step4: Testing:

1. Kselftests:
I used this becuase its 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).

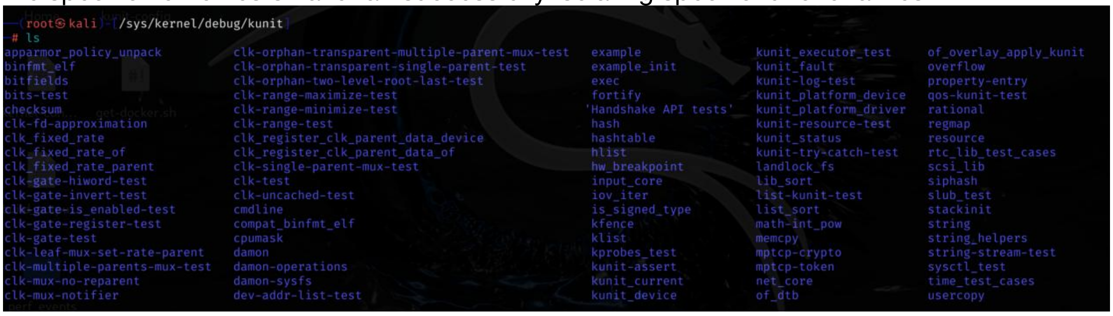
(Also submitted the full output of the kselftests in /Tests/kself-tests.txt)

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



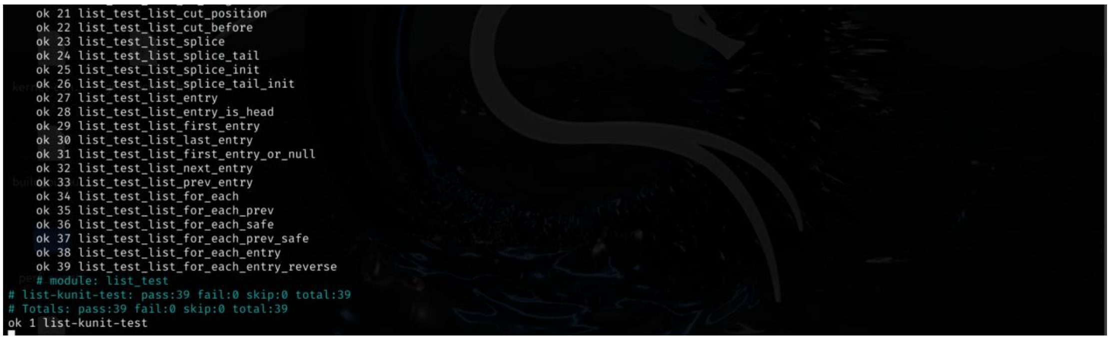
2. KunitTests:
So I used, Kunit tests, which are essential to cover testing the missing aspects of kselftests and other core internal features of the kernel.

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.

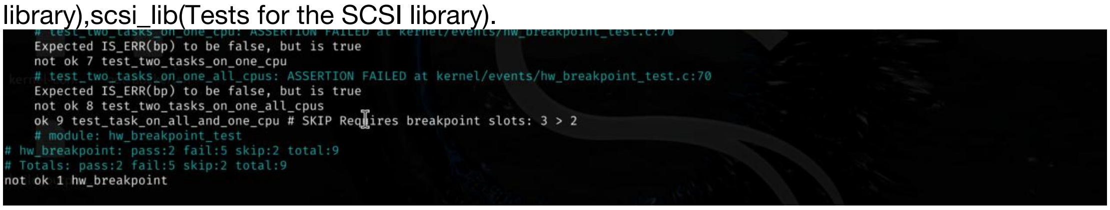


Screenshots of some of the main unit tests with their significance is provided below along with the test files being attached in the root folder submitted (/Tests/kunit-tests).

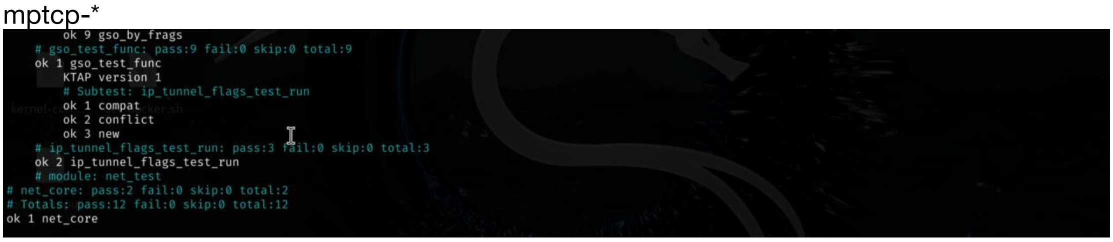
1. Internal data structures tests[ALL PASSED]: list-kunit-test,list_sort,klist (Test on operations on kernel data strucutres like linked lists etc)



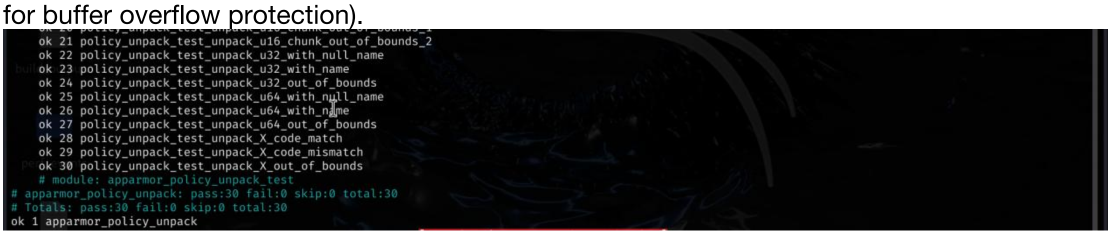
2. Driver Tests[ALL PASSED]:
hw_breakpoint: (Tests for hardware breakpoints),rtc_lib_test_cases(tests real time clock library),scsi_lib(Tests for the SCSI library).



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



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



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
    KTAP version 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.