A Guide to Hacking the Linux Kernel

Abhishek Safui

Linux Kernel Hacking

In this document, we discuss the steps to get yourself ready for exploring the Linux kernel. We will be compiling the Linux kernel on a development host (laptop/desktop) and running the kernel on a virtual machine (same architecture). We are using an x86_64 (64-bit x86 Intel/AMD) laptop running Ubuntu 22.04, for development and testing purposes. Additionally, we shall use *libvirt*, a virtualization library that manages the interface to the QEMU CPU emulator and KVM virtual machine. Specifically, it exposes APIs to launch virtual machines. The *virt-manager* is a graphical user interface for configuring *libvirt* virtual machines. The v*irsh* command is the command line counterpart of *virt-manager*.

Getting your system ready

The first step towards Linux kernel hacking is to prepare your system for kernel compilation and testing.

For compiling the Linux kernel, we need to install the following dependencies:

```
sudo apt update
```

```
sudo apt install git fakeroot build-essential ncurses-dev xz-utils
libssl-dev bc flex libelf-dev bison
```

To launch virtual machines using *libvirt*, we need to install the following software packages:

```
sudo apt install cpu-checker qemu qemu-kvm libvirt-daemon libvirt-
clients bridge-utils virt-manager
```

Add the user to *libvirt* group.

```
sudo usermod -G libvirt -a $USER
```

Check that the *libvirtd* daemon is running:

```
sudo systemctl status libvirtd
```

Check that KVM is working in your development system using the following command:

```
sudo kvm-ok
```

If everything is OK, the output should be:

```
INFO: /dev/kvm exists
```

```
KVM acceleration can be used
```

If KVM does not work, you may need to check the BIOS configuration to enable virtualization technologies (Intel VT-X or AMD-V).

The steps that we will typically follow are:

- (Cross) Compile the kernel
- Run the compiled kernel in a virtual machine
- Attach GDB to the kernel running on the virtual machine.

Compile the Linux Kernel

Fetch the Linux kernel code:

```
git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
cd linux
git checkout v6.1.6
```

- The same instructions should work for any 6.x kernel.

After fetching the Linux kernel source code, we need to configure the features to be compiled using a .config file that can be generated using the *make menuconfig* command. A default configuration will get generated if we run *make menuconfig* and do not modify any configuration. We may not be interested in all the features provided by the Linux kernel's default configuration. Disabling unnecessary drivers and other Linux kernel features will reduce the kernel compilation time and size.

For the first build, we may go ahead with the default .config file generated by running *make menuconfig*. Later on we can replace the .config file with a minimal set of features that we need for our development and testing purposes.

Now, to build the kernel, just execute the *make* command with a single argument: the number of cores available in your system.

```
make -j <num_cores>
```

If you encounter errors related to system certificates (required for verification of signed kernel modules for security purposes), disable them to avoid the additional complexity. It is important to keep matters simple at this stage. Run the following commands.

```
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS
```

Once the kernel is compiled, it will be available at *arch/x86_64/boot/bzImage*.

Directly Booting the Linux Kernel on QEMU

To boot the x86_64 Linux kernel on a virtual machine, we need a *rootfs* prepared for the x86_64 architecture. We may use a pre-built *rootfs* available for download over the internet, or we may build a custom *rootfs* of our own.

In this step, we shall build a minimal root filesystem and create a .config file for configuring our Linux kernel compilation features. The aim is to reduce the size of the Linux kernel binary and the time required to build the kernel.

Building a custom rootfs for our custom kernel (Optional if using a prebuilt rootfs)

To reduce the compilation and testing time, we can use a minimal configuration available from embedded Linux platform tools such as **Buildroot** (https://buildroot.org/) or The **Yocto Project** (https://www.yoctoproject.org/). Both these tools are extensively used in the software industry to package software for deployment on a VM or embedded hardware. These tools allow us to create custom Linux operating system images, which include a *rootfs* with desired packages, libraries and a package management system, and a custom kernel. Expertise in these tools is a much-desired skill particularly for embedded Linux kernel developers.

In this document, we shall explain how to get started with the Yocto Project. The *rootfs* and the kernel .config file created in this step will be used to boot a VM with our custom kernel using *virt-manager*. Please make sure that you have 60 GB of disk space available. The *build* process will take a lot of time to build everything from scratch, but subsequent modifications can be processed very quickly.

Getting Started with the YOCTO Project

Download

```
git clone -b mickledore git://git.yoctoproject.org/poky.git
```

Configure

```
cd poky
```

```
source oe-init-build-env
```

This will create a `build/` directory.

Edit *build/conf/local.conf* to modify the following variables according to your requirements:

PACKAGE_CLASSES ?= "package_deb"

SDKMACHINE ?= "x86_64"

EXTRA_IMAGE_FEATURES ?= "debug-tweaks tools-sdk tools-debug tools-profile"

IMAGE_FSTYPES = "live wic.qcow2"

CONF_VERSION = "2"

INHERIT += "rm_work"

BB_NUMBER_THREADS = "10"

PARALLEL_MAKE = "-j 10"

IMAGE_INSTALL:append = "procps vim"

The above configuration is self-explanatory, except the EXTRA_IMAGE_FEATURES. Each feature is well documented in this link https://docs.yoctoproject.org/3.2.3/ref-manual/ref-features.html#image-features

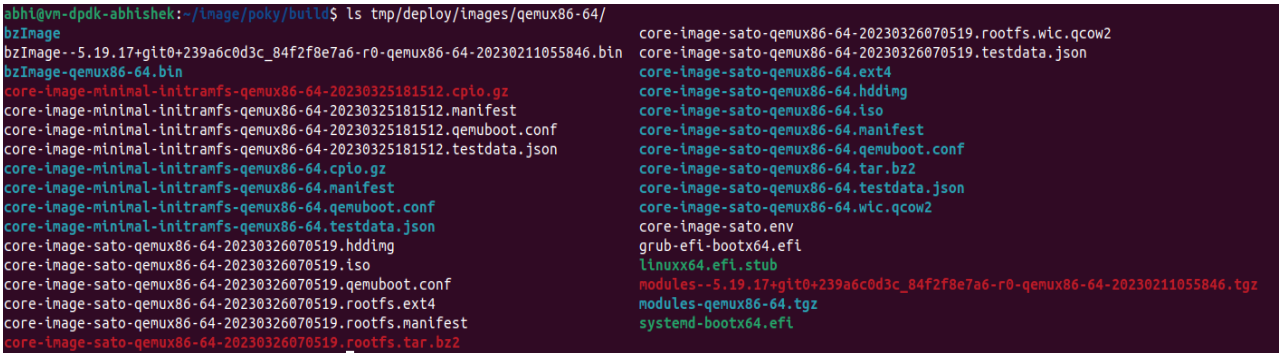
Build

Execute the following command to build an image

```
bitbake -k core-image-sato
```

core-image-sato will build an image with a minimal GUI for navigation. *core-image-minimal* will generate a minimal image with no GUI.

The image will be stored at poky/build/tmp/deploy/images/qemux86-64/



The Linux kernel compiled by the Yocto project will store its .config at

poky/build/tmp/work/qemux86_64-poky-linux/linux-yocto/5.19.17+gitAUTOINC+239a6c0d3c_84f2f8e7a6-r0/linux-qemux86_64-standard-build/.config

We will not need this kernel binary, but we will be using the generated .config file to compile our kernel.

Preparing the SDK for Compiling User Space Applications (optional)

It is also good to have an SDK, which we can use to compile software meant for running inside the VM booted with our custom *rootfs*. We can use the following command:

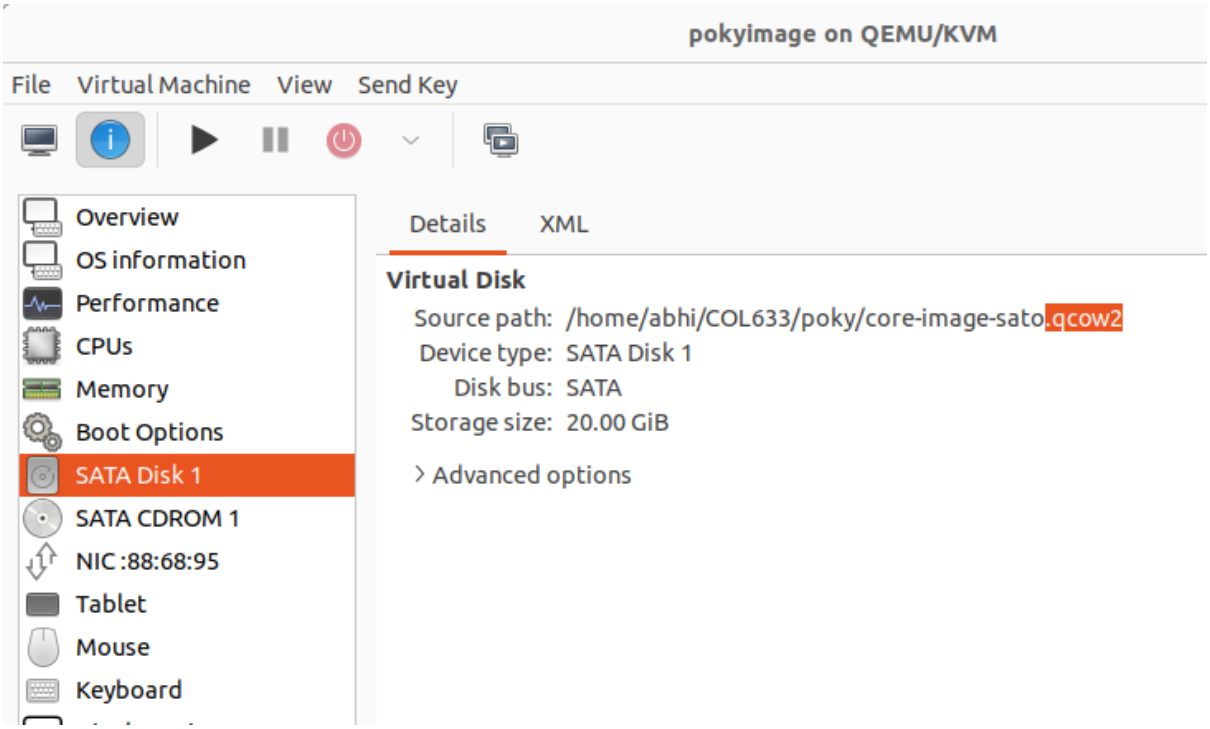
```
bitbake -c populate_sdk core-image-sato
```

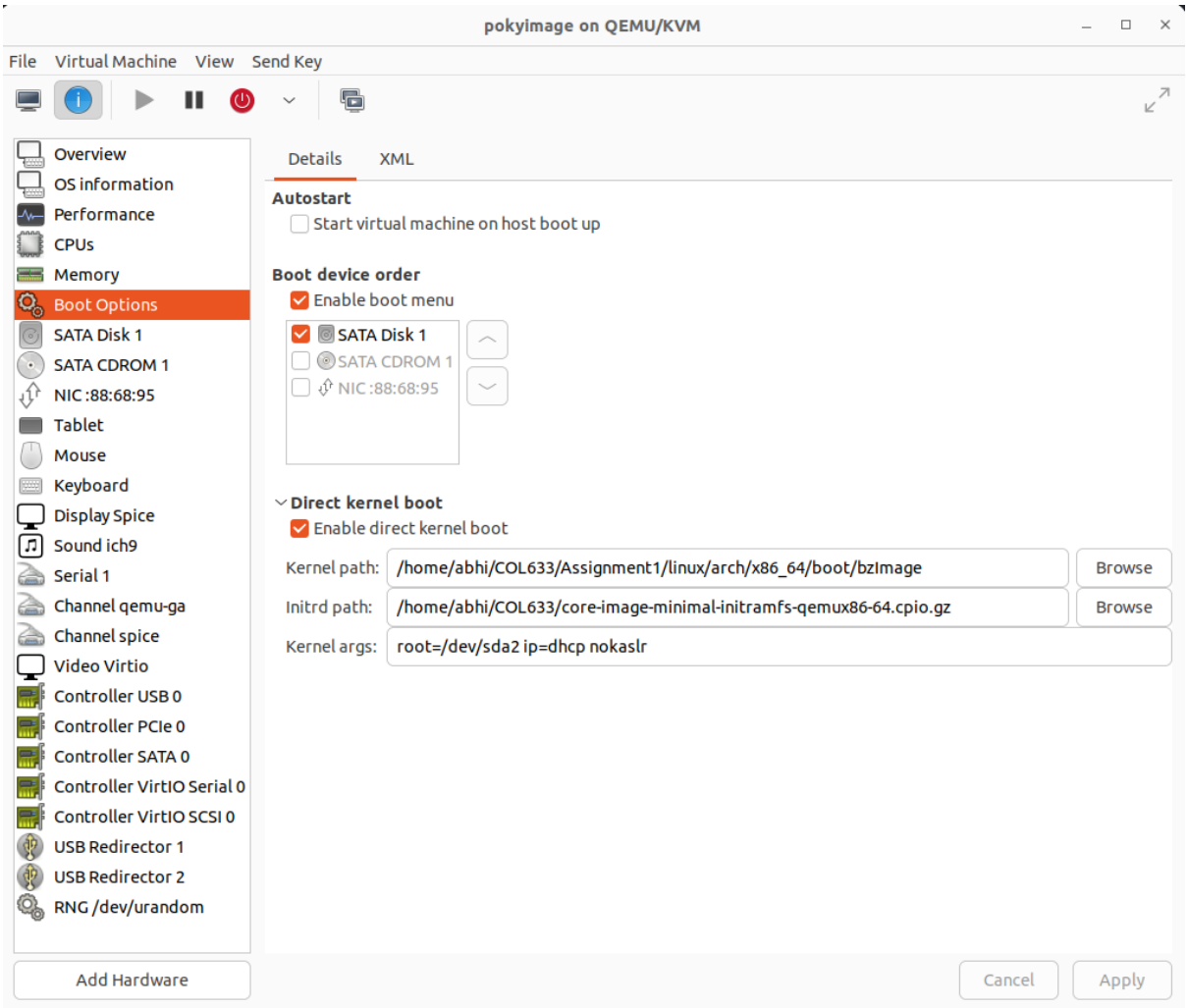
To understand in detail how a *rootfs* is built from scratch, one may consult this reference. It describes the process in great detail: https://www.linuxfromscratch.org/

Running the Kernel

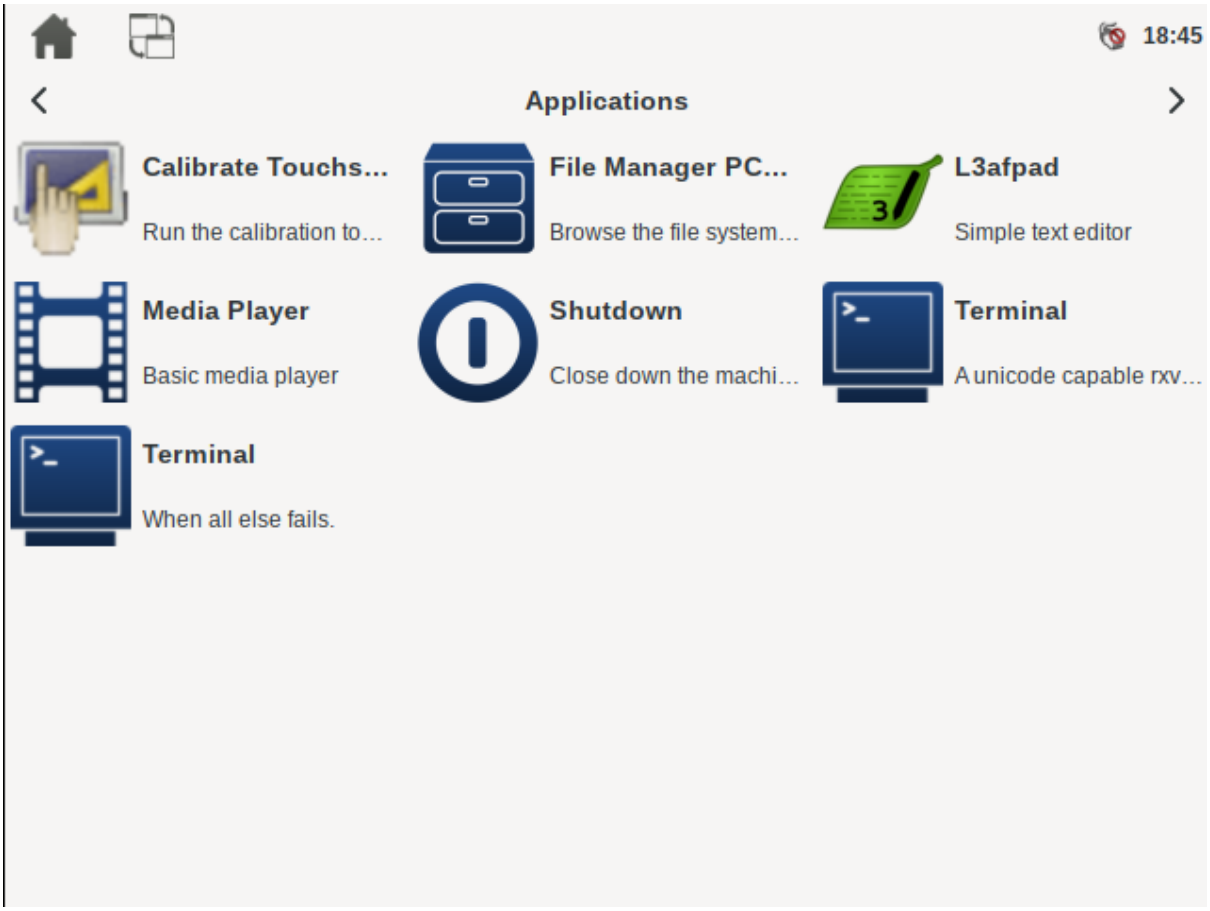
Now that we have a kernel (the Yocto kernel) and a *rootfs*, we can try booting up a VM. Later when we build our own kernel, we will replace the kernel keeping the *rootfs* the same.

We first need to configure a VM in *virt-manager* as shown in the images below.




pokyimage on QEMU/KVM
File
Virtual Machine
View
Send Key
Overview
OS information
Performance
CPUs
Memory
Boot Options
SATA Disk 1
SATA CDROM 1
NIC :88:68:95
Tablet
Mouse
Keyboard
Display Spice
Sound ich9
Serial 1
Channel qemu-ga
Channel spice
Video Virtio
Controller USB 0
Controller PCIe 0
Controller SATA 0
Controller VirtIO Serial 0
Controller VirtIO SCSI 0
USB Redirector 1
USB Redirector 2
RNG /dev/urandom
Add Hardware
Details
XML
Autostart
Start virtual machine on host boot up
Boot device order
Enable boot menu
SATA Disk 1
SATA CDROM 1
NIC :88:68:95
Direct kernel boot
Enable direct kernel boot
Kernel path:
/home/abhi/COL633/Assignment1/linux/arch/x86_64/boot/bzImage
Browse
Initrd path:
/home/abhi/COL633/core-image-minimal-initramfs-qemux86-64.cpio.gz
Browse
Kernel args:
root=/dev/sda2 ip=dhcp nokaslr
Cancel
Apply

It should boot with a GUI (like the image shown below) for *core-image-sato*.



Now that we have booted up an image, we note down the free space available in the image.

Realizing that we do not have much space left in the booted hdd, we need to expand the hdd and then modify the partition table. We can increase the disk size as follows:

```
qemu-img resize disk.qcow2 +10G
```

To change the partition table, we need to boot up the VM with a *gparted* ISO image and expand the partition of the hdd and then reboot the system without the cdrom.

Replace the Yocto Kernel with our Custom Kernel

We have already compiled an x86_64 kernel, which we can boot using QEMU: **qemu-system-x86_64**.

We may now replace the kernel and its initialization (*initrd* (initial RAM disk)) configuration in the *virt-manager* configuration section "Boot Options" and boot with the generated *bzImage* located at arch/x86_64/boot/bzImage. Before booting, we also need to install the kernel modules and header files in the *rootfs*. For this, we need to mount the image (explained in the next section). After mounting the disk image containing the *rootfs*, and installing the kernel headers and the kernel modules, we may proceed to start the virtual machine.

In case, the kernel complains about an unknown filesystem, we need to create an *initrd* for the kernel or reinclude all the required drivers and rebuild the kernel.

Mounting the rootfs to Install the Kernel Modules and Headers

While the VM is shut down, we perform the following steps to mount the *hdd* image and perform modifications as needed.

This is a quick guide to mounting *qcow2* disk images on your host. This is useful to reset passwords, edit files or recover something without the virtual machine actually running.

Step 1 - Enable NBD (network block device module) on the host

```
modprobe nbd max_part=8
```

Step 2 - Connect the QCOW2 image as the network block device

```
qemu-nbd --connect=/dev/nbd0 /var/lib/vz/images/100/vm-100-disk-1.qcow2
```

Step 3 - Find the virtual machine partitions

```
fdisk /dev/nbd0 -l
```

Step 4 - Mount the partition from the VM

```
mount /dev/nbd0p1 /mnt/somepoint/
```

Step 5 - After you are done, unmount and disconnect

```
umount /mnt/somepoint/
```

```
qemu-nbd --disconnect /dev/nbd0
```

```
rmmod nbd
```

After mounting, we install headers as follows:

```
sudo make headers_install INSTALL_HDR_PATH= <mount-point>/usr/include
```

To install modules:

```
export INSTALL_PATH=<mount-point>/lib/modules/<kernel-version>/
make modules_install
```

Debugging the Linux Kernel with KDB and KGDB

This section discusses the steps to connect to the kernel debugger for debugging the kernel code. There are two ways of connecting to the kernel debugger: kdb and kgdb. The Linux Kernel has a debug core that is common to both kdb and kgdb.

KDB vs KGDB

kdb is a debugging tool that is not source code aware, but it provides a shell to perform kernel debugging, like dumping the kernel log buffer using *dmesg* (messages written by the kernel).

kgdb(gdb), on the other hand, is a source code-aware tool that lets us peek into the kernel data structures, like what we can do with `*gdb*` for user space applications (with some limitations). But there are limitations of using kgdb on a complex system like the kernel since the entire system is halted, and interrupts and other time-critical events are delayed if we try to single-step through each line of the kernel code. Typically, we use kdb for simple tasks like dumping all available debugging information and move on to kgdb if required.

Some of the tasks that can be achieved with the *kdb* shell are:

- Dump register or memory contents
- Change memory contents
- Dump dmesg logs
- List all processes
- Backtrace any process
- Dump the ftrace buffer(s)

Using KDB

To debug the Linux Kernel with the kdb shell, we need to compile the Linux Kernel with some flags enabled, which we explain in this section. After compilation, we boot the kernel in a qemu virtual machine and launch the kdb shell.

Compiling the Linux Kernel with debug flags

To debug the kernel using kdb, we need to compile the kernel with following flags:. The mandatory configuration options for kdb are highlighted. The **CONFIG_KGDB** flag enables the Linux kernel debugger and the **CONFIG_KGDB_KDB** flag enables the kdb frontend to the kernel debugger. Other flags enable alternate ways of invoking the kdb shell.

- **CONFIG_KGDB=y**
- **CONFIG_KGDB_KDB=y**
- CONFIG_FRAME_POINTER=y # For accurate stack back traces
- CONFIG_KGDB_SERIAL_CONSOLE=y
- CONFIG_KDB_KEYBOARD=y #Applicable for KDB only, with PS/2 style keyboard as input
- CONFIG_MAGIC_SYSRQ=y # To enter kdb using MAGIC SYSRQ
- CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE=0x1
- CONFIG_MAGIC_SYSRQ_SERIAL=y
- CONFIG_MAGIC_SYSRQ_SERIAL_SEQUENCE=""

To enable CONFIG_DEBUG_INFO, go to *Kernel hacking > Compile-time checks and compiler options > Debug information* and select *Generate DWARF Version 4 debuginfo.*

To enable *CONFIG_FRAME_POINTER* go to *Kernel hacking > x86 Debugging > Frame pointer unwinder* and select *Frame pointer unwinder*.

To enable *CONFIG_KGDB* go to *Kernel hacking > Kernel debugging* and select *KGDB: kernel debugger*

To search the exact location for any config option in *make menuconfig*, press `/` and type in the configuration option that you are looking for.

Booting the Linux Kernel and Invoking the *kdb* Shell

Once the Linux Kernel is compiled with the required debug flags, boot the kernel in a *qemu* virtual machine as already explained in a previous section. As the *root* user, invoke the following command:

echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc

The above command may be executed from any terminal (*ssh* login terminal for example). But the *kdb* command will only work from *ttyS0,* which is the serial console. So, the next step is to log in to the serial console. Use the following command to enter the console.

virsh console <vm-name>

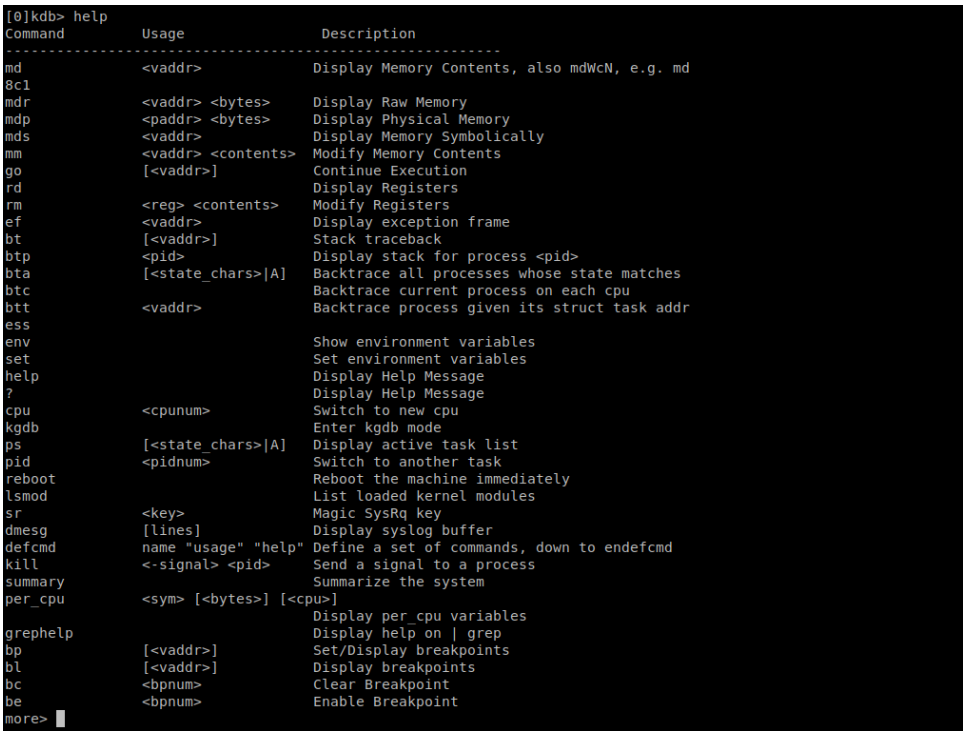
Press *Enter* to get the prompt. Login using root credentials. You may wait for a fault to happen or may enter the kdb shell using the following command:

echo g > /proc/sysrq-trigger

```
root@qemux86-64:~# tty
/dev/ttyS0
root@qemux86-64:~# echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
root@qemux86-64:~# echo g > /proc/sysrq-trigger

Entering kdb (current=0xffff88810e031f80, pid 729) on processor 0 due to NonMask
able Interrupt @ 0xffffffff8114cc74
[0]kdb> bt
Stack traceback for pid 729
0xffff88810e031f80      729      723  1    0   R  0xffff88810e032a40 *sh
CPU: 0 PID: 729 Comm: sh Not tainted 6.1.6-yocto-standard+ #128
Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS 1.15.0-1 04/01/2014
Call Trace:
 <TASK>
 dump_stack_lvl+0x38/0x4d
 dump_stack+0x10/0x16
 kdb_dump_stack_on_cpu.cold+0x5/0xa
 kdb_show_stack+0x82/0x90
 kdb_bt1+0xc1/0x130
 kdb_bt+0x341/0x3a0
 kdb_parse+0x2b9/0x6a0
 ? kprobe_free_init_mem+0x90/0xc0
 kdb_main_loop+0x4a3/0x990
 ? kprobe_free_init_mem+0x90/0xc0
 kdb_stub+0x1b7/0x400
 kgdb_cpu_enter+0x326/0x5c0
 kgdb_handle_exception+0xc0/0x110
 __kgdb_notify+0x34/0x90
 kgdb_ll_trap+0x46/0x60
 do_int3+0x30/0x90
more>
```

Use the *help* command to list all the commands available in the *kdb* shell.



In case you don’t get the kdb shell working, check the terminal using the *tty* command. It should be the same serial console as configured using the *kgdboc* parameter.

Using KGDB (*gdb)*

In this section, we shall describe the process of debugging the Linux Kernel using *kgdb*. First, we introduce *kgdb* and explore its relationship with our familiar debugging tool, *gdb.* Next, we describe how to compile the Linux Kernel to enable *kgdb.* Finally, we explain how to configure *qemu* to allow gdb running on a remote host to attach to the Linux Kernel running as a guest.

KGDB and the Linux Kernel Debug Core

kgdb is a stub that allows *gdb*, running on a second machine, to connect to the *kgdb* core of the target machine. *kgdb* has two components running inside the kernel:

- KGDB core: It performs functions like setting breakpoints and fetching the data in memory.
- KGDB I/O: It connects the *kgdb* core to various drivers like the serial console, keyboard, etc., and takes care of the transmission of the debug information.

In the *kgdb* approach of debugging the Linux Kernel, we use *gdb* running on a second machine since it serves as the frontend client to connect to the *kgdb* core. There are multiple ways of connecting *gdb* to the *kgdb* core. The *kgdboc* kernel command line parameter is used to specify how we wish to connect to it. There are other related command line parameters:

- *kgdwait* tells the kernel to wait (during boot) until the debugger is attached
- *sysrq_always_enabled* enables the sysrq (Magic System Request Key)

FreeBSD has a tool named *kgdb* that is based on *gdb* and is used to connect to the KDGB core. But this tool is limited to FreeBSD.

Kernel Configuration to get kgdb to Work

Disable KASLR at compile time using the CONFIG_RANDOMIZE_BASE config option. This can also be done at runtime using the *nokaslr* kernel command line option. Also, set the following compile-time flags.

- **CONFIG_KGDB=y**
- **CONFIG_KGDB_SERIAL_CONSOLE=y**
- CONFIG_GDB_SCRIPTS=y
- CONFIG_DEBUG_KERNEL=y
- CONFIG_DEBUG_INFO=y # gdb will require symbols for proper debugging
- CONFIG_FRAME_POINTER=y # For accurate stack back traces

Directly Booting the Kernel in QEMU/KVM and Debugging using kgdb

In this step, we directly boot the kernel, as we did in a previous section that explained how to get started with booting a custom kernel with a custom *rootfs*. This time we will configure the QEMU VM with the "-s" option. This option asks QEMU to listen on TCP port 1234 for connections from k*gdb*. To configure the *libvirt* virtual machine with this option, we edit the VM configuration using the following command.

```
virsh edit <vm-name>
```

This will open the *libvirt* XML file corresponding to the virtual machine. We add the following configuration under the *domain* tag. Note the *xmlns* option in the *domain* tag.

```
<domain type='kvm'
xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>

...

...

  <qemu:commandline>

    <qemu:arg value='-s'/>

  </qemu:commandline>

</domain>
```

Now that we have booted the VM with the kernel that is compiled with all the required configuration options for debugging, we are ready to attach *gdb* to the *kgdb* core from the host machine as follows.

```
gdb vmlinux

(gdb) target remote :1234

(gdb) continue
```

```
(gdb) target remote :1234
Remote debugging using :1234
0xffffffff81d8c32c in default_idle () at arch/x86/kernel/process.c:731
731     }
(gdb) bt
#0  0xffffffff81d8c32c in default_idle () at arch/x86/kernel/process.c:731
#1  0xffffffff81034892 in arch_cpu_idle () at arch/x86/kernel/process.c:722
#2  0xffffffff81d8c5dd in default_idle_call () at kernel/sched/idle.c:109
#3  0xffffffff810cba1c in cpuidle_idle_call () at kernel/sched/idle.c:191
#4  do_idle () at kernel/sched/idle.c:303
#5  0xffffffff810cbc80 in cpu_startup_entry (state=state@entry=CPUHP_ONLINE) at kernel/sched/idle.c:400
#6  0xffffffff81d83074 in rest_init () at init/main.c:729
#7  0xffffffff8287adba in arch_call_rest_init () at init/main.c:890
#8  0xffffffff8287b465 in start_kernel () at init/main.c:1145
#9  0xffffffff8287a4a1 in x86_64_start_reservations (real_mode_data=real_mode_data@entry=0x13ca0 <exception_st
#10 0xffffffff8287a56c in x86_64_start_kernel (real_mode_data=0x13ca0 <exception_stacks+31904> <error: Cannot
#11 0xffffffff81000145 in secondary_startup_64 () at arch/x86/kernel/head_64.S:358
#12 0x0000000000000000 in ?? ()
(gdb) info thread
  Id   Target Id                    Frame
* 1    Thread 1.1 (CPU#0 [halted ]) 0xffffffff81d8c32c in default_idle () at arch/x86/kernel/process.c:731
  2    Thread 1.2 (CPU#1 [halted ]) 0xffffffff81d8c32c in default_idle () at arch/x86/kernel/process.c:731
(gdb)
```

Conclusion

In this section, we have described the process of attaching gdb to the kernel running in the guest QEMU/KVM virtual machine over the network from the host machine. We have also explained how to use the *kdb* shell to connect to the Linux Kernel debug core.

References

1. https://qemu-project.gitlab.io/qemu/system/gdb.html
2. https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html
3. Using kgdb, kdb and the kernel debugger internals — The Linux Kernel documentation