Goal

Get Ubuntu running in a VM with a custom kernel starting from scratch. Then, access the VM via SSH, and use GDB to debug the kernel of the VM.

NOTE: You will need sudo access to run some of these commands.

Overview

To test our kernel, we want to be able to run a QEMU command to start the VM, passing the `-kernel` option to specify our kernel build without having to build and install a full .rpm or .deb. Moreover, we would like to be able to run a full distro (in this case, Ubuntu), rather than the stripped down environment you get just after boot. Finally, we would like to be able to SSH into our VM to interact with it, rather than having to use the serial console or VNC.

Our setup will look like this:

- At boot, the system will start using an initramfs that we build ourselves using busybox. This is a stripped-down in-memory root filesystem that contains the bare minimum we need to initialize devices and then jump to a more comprehensive init system.
- Immediately after booting, the kernel will attempt to run whatever file is at /init in the initramfs. We will provide this file so that it does minimal setup, mounts a device with a full Ubuntu setup and then switches to running Ubuntu's init.
- We will use Ubuntu's cloud-images to build the root filesystem for the Ubuntu side of things.
- On the QEMU end, we will use virtio-blk and virtio-net devices as our storage and networking devices for the VM.
    - The storage devices will be backed by various files we create
    - The networking devices will forward host port 5555 (an arbitrary choice) to guest port 22 (SSH) so that we can SSH into the guest.

Steps

We are going to need a bunch of different files for this to work. We will build them one by one. We will collect all of our assets in a single directory for convenience:

```
mkdir linux-qemu
```

Creating the initramfs

First, we want to build the initramfs for our kernel. Recall the initramfs is the root filesystem we boot into initially, so it needs to have the minimum stuff need for an "init" process to run. To do this, we will use a popular project called busybox that provides a bunch of utilities for booting in minimal kernel environments (e.g. embedded systems or early boot). We will set up a minimal root filesystem, use busybox to provide the basic utilities, and then package it all up into an initramfs.

```
mkdir initramfs
cd initramfs/
mkdir -pv bin lib dev etc mnt/root proc root sbin sys
touch dev/{null,console,tty}

cd .. # linux-qemu/
wget https://busybox.net/downloads/busybox-1.33.0.tar.bz2 # or latest version
tar xvf busybox-1.33.0.tar.bz2
cd busybox-*
make defconfig
make # consider adding -j
make install
```

That last command generates the `_install` directory, which contains all of busybox's compiled utilities, which is what we want.

```
cp -avR _install/* ../initramfs/
```

One catch, though: the built binaries may use shared executables from your host system, which obviously won't be present in the VM. So we need to copy them over… this is a bit of a hack, but it works. We need to be careful to copy the binaries and not just symlinks, and we need to be careful to put them in the right places:

```
cd ../initramfs
ldd bin/busybox
```

On my system, this returns the following:

```
ldd bin/busybox
       linux-vdso.so.1 (0x00007ffff852a000)
       libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f22b266f000)
       libresolv.so.2 => /lib/x86_64-linux-gnu/libresolv.so.2 (0x00007f22b2653000)
       libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f22b2461000)
       /lib64/ld-linux-x86-64.so.2 (0x00007f22b28d5000)
```

So bin/busybox expect there to be a shared lib called `libm.so.6` at the path `/lib/x86_64-linux-gnu/libm.so.6` in the initramfs. Note that your paths and filenames may not match mine (e.g. they may point to /usr/lib or elsewhere, whereas mine are in /lib/x86_64...), but you need to make sure that your initramfs has files at the exact same paths as those output by ldd. Otherwise, your kernel will output cryptic error messages about not finding init or not being able to execute it. Also note that ldd will output a line about linux-vdso or linux-gate, and you can ignore that line -- VDSO is a special kernel mechanism for making some system calls fast and it has a weird special implementation.

```
mkdir -pv lib/x86_64-linux-gnu/ lib64/ # in initramfs/
cp -aLv /lib/x86_64-linux-gnu/lib[mc].so.6 lib/x86_64-linux-gnu/
cp -aLv /lib/x86_64-linux-gnu/libresolv* lib/x86_64-linux-gnu/
cp -aLv /lib64/ld-linux-x86-64.so.2 lib64/
```

Note that the -L flag tells cp that it should follow symlinks and copy the file itself, rather than making a copy of the symlink.

Now we need to create our init program. This will be process 0 on our new system and it is the first process our kernel will execute after booting. It is responsible for setting up the rest of the system. Create a file called `init` with the following contents in `linux-qemu/initramfs/`:

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mknod -m 666 /dev/ttyS0 c 4 64
mdev -s

echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"

echo "Switching root..."

mkdir /newroot
mount /dev/vda1 /newroot

mount --move /sys /newroot/sys
mount --move /proc /newroot/proc
mount --move /dev /newroot/dev

echo -e "\nWelcome to Ubuntu Cloud Image.\n$(uname -a)\n"

exec switch_root /newroot /sbin/init
```

Mark the file as executable:

```
# in initramfs/
chmod +x init
```

The init script is fairly simple: first, it mounts the special procfs, sysfs, and devfs filesystems. It then executes the `mdev` program from busybox. This basic program knows how to probe hardware and figure out what is attached to the computer. When `mdev` returns, the `/dev/` directory will be populated with a bunch more devices, including our storage and networking devices. Afterwards, we mount `/dev/vda1`, which is the main partition of the Ubuntu root file system we will create shortly. Finally, we `switch_root` to the new root file system and then execute Ubuntu's init program `/sbin/init`, which does the heavy lifting.

Ok, finally, we will package all of the above up into an initramfs archive:

```
# in initramfs/
find . -print0 | cpio --null -ov --format=newc > ../my-initramfs.cpio
```

Creating our disk images

Next, we want to build our storage devices that will be our eventual system. To do this, we will use Ubuntu cloud images. These are pre-built official Ubuntu images to be used with various cloud providers/tools (e.g. Azure, Vagrant, KVM, etc)...

```
cd .. # linux-qemu/
wget -O focal-server-cloudimg-amd64.qcow2 \
    https://cloud-images.ubuntu.com/focal/current/focal-server-cloudimg-amd64.img
qemu-img convert -O raw focal-server-cloudimg-amd64.qcow2 \
    focal-server-cloudimg-amd64.img
```

This will be our new root file system device. The init program in this root device will look for a second device that contains some user configuration information on it -- most notably a list of users and allowed SSH keys. To create this second device, we will use the `cloud-localds` utility:

```
sudo apt install cloud-utils # to install cloud-* utils
```

We create a small config called `cloud.yaml` in `linux-qemu/` with the following contents:

```
#cloud-config
users:
  - name: markm
        ssh-authorized-keys:
        - ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0ukRW3EmWNEDyLYX4/5tLEWjV/A9Xbfv6A/xgh7SU3rlR9PDhy6hjK/2S2ARGzoa8V7ibRwqqq2gCwwSGK17X4kBI1PQq+198Cf
GZ4lmqDSgTavqZs6gEGSZ7PEEk
        sudo: ['ALL=(ALL) NOPASSWD:ALL']
        groups: sudo
        shell: /bin/bash
```

Note that the first line is NOT A COMMENT; don't omit it. Also, make sure to update the username and SSH public key to your own. Then, create another config called `cloud-net.yaml` also in `linux-qemu/` with the following contents:

```
version: 2
ethernets:
  interface0:
```

```
match:
  name: enp*
dhcp4: true
```

The “name:” key is the name of the ethernet device which is our virtio-net-pci device. On older kernels, the name of the device may be something like “eth0”. You can look at the kernel’s boot log to find the correct name if it turns out to be incorrect.

Then, we can create our image like so:

```
cloud-localds -v --network-config=cloud-net.yaml cloud.img cloud.yaml
```

This will generate a file called cloud.img, which will be our second storage device.

Building the kernel

Finally, we will build our kernel. You can find lots of instructions about this elsewhere, so I won’t go in detail here. The main things to note are that there are some important kernel configurations to have on:

```
        CONFIG_BLK_DEV_INITRD=y
        CONFIG_PCI=y
        CONFIG_BINFMT_ELF=y
        CONFIG_SERIAL_8250=y
        CONFIG_EXT2_FS=y
        CONFIG_NET=y
        CONFIG_PACKET=y
        CONFIG_UNIX=y
        CONFIG_INET=y
        CONFIG_WIRELESS=n
        CONFIG_ATA=y
        CONFIG_NETDEVICES=y
        CONFIG_NET_VENDOR_REALTEK=y
        CONFIG_8139TOO=y
        CONFIG_WLAN=n
        CONFIG_DEVTMPFS=y
        CONFIG_VIRTIO=y
        CONFIG_VIRTIO_BLK=y
        CONFIG_VIRTIO_NET=y
        CONFIG_ISO9660_FS=m
        CONFIG_EXT4_FS=y

The following will make debugging on QEMU with GDB much easier:

        # Turn on debugging symbols
        CONFIG_DEBUG_INFO_DWARF5=y

        # Reduce timer interrupts during virtualization...
        CONFIG_X86_X2APIC=y
        CONFIG_PARAVIRT=y
        CONFIG_KVM_GUEST=y

        # Turn off KASLR so that symbols don’t move around...
        CONFIG_RANDOMIZE_BASE=n

        # Generate GDB debugging scripts…
        CONFIG_GDB_SCRIPTS=y
```

You can turn on whatever other configuration options you want, but these are a minimum. They are necessary drivers that allow you to use the QEMU devices we are using. Make sure you configure the drivers as built-in not modules. Also, you may need to select some dependencies of these config options too…

Once you have configured your kernel you can use “make” to build it. Note that we don’t need a full distribution here (e.g. binrpm-pkg or the like). Just “make” will be sufficient. This will build a vmlinux and the corresponding bzImage file, which is what we want.

Running with QEMU

We now have all the components we need. We just pass them all to QEMU. I like to use the following script. Feel free to customize it:

```
#!/bin/bash

MEM=$1

if [ -z "$MEM" ] ; then
        echo "Usage: ./run <memory_size>"
        exit -1 ;
fi

/usr/bin/qemu-system-x86_64 \
        -m "$MEM" \
        -smp 4,sockets=1,cores=4,threads=1 \
        -kernel /home/mark/linux/kbuild/arch/x86/boot/bzImage \
        -initrd $LINUX_QEMU/my-initramfs.cpio \
        -append "console=ttyS0,115200n8" \
        -nographic \
        -device virtio-blk-pci,id=vd0,drive=drive0,num-queues=4 \
                -drive file=$LINUX_QEMU/focal-server-cloudimg-amd64.img,format=raw,if=none,id=drive0 \
        -device virtio-blk-pci,id=vd1,drive=drive1,num-queues=4 \
                -drive file=$LINUX_QEMU/cloud.img,format=raw,if=none,id=drive1 \
        -device virtio-net-pci,netdev=net0 \
                -netdev user,id=net0,hostfwd=tcp::5555-:22 \
        -enable-kvm \
        -s
```

You'll need to adjust file paths appropriately.

When you run this command, your kernel will boot up in QEMU using the initramfs, which will then start the cloud image init. The first time you boot up, the cloud-init system (the part of cloud-utils that runs on the guest) will set up the machine using the config options placed on the secondary drive. If you accidentally mess up your config, you can just delete and regenerate the two storage device files using the same methodology above.

Here is what all of the flags do:

-m	The amount of virtualized memory the VM has (in MB). I use 1GB, but you can use more or less (I think 512MB is the minimum).
-smp	Configures the vCPUs the VM has. In this case, I give it 4 vCPUs. Feel free to adjust.
-kernel	This is the bzImage for our custom kernel (a compressed form of our kernel that is unpacked and run at boot time). You can find at the arch/x86/boot/bzImage in whatever directory you built your kernel in.
-initrd	This is the initramfs we built at the beginning of this document.
-append	This adds parameters to the kernel boot command line. In this case, the parameters simply tell the kernel to output a serial console to ttyS0, which allows us to see the console log as QEMU boots the kernel
-nographic	Tells QEMU not to open a window representing a physical screen. This is useful when you are working on a remote machine. Instead, we will interact with the VM via SSH or the console.
-device, -drive, -netdev	These flags set up our virtual devices. The -device flag creates a "frontend". This is the virtual device itself. The -drive/-netdev flags create a "backend". This is the way the device appears to the VM. Thus, the flags go in pairs. In the above example, we have 3 pairs (2 storage devs and 1 NIC). For all of them, the first argument in the -device flag is the type of emulated device. QEMU is able to emulate some types of real devices. In this case, though, we just use virtio devices for everything. Each device has an ID and is associated with some backend (i.e. "drive=..." and "netdev="). They also may have some device-specific arguments, such as the size of the IO queue. The -drive flags allow you to specify the file, format, and frontend for your storage devices. The -netdev flag allows you to specify arguments for your virtual NIC, along with the frontend of your NIC. The "user" netdev type specifies "user-mode networking," which is one of the networking implementations that QEMU supports. The "hostfwd=..." argument forwards TCP traffic on port 5555 on the host to guest port 22 (SSH). This allows you to SSH into the machine by passing SSH -p 5555 on the host.
-enable-kvm	Enable hardware acceleration via KVM. Much faster.
-s	Start the gdb stub server. This allows you to connect to the VM with gdb.

SSH into the machine and Debugging with GDB

Ok, now we have built and booted our custom kernel, and we want to run stuff in the VM and observe the kernel using GDB.

We can SSH into the machine by running this from the host on which we are running the VM:

```
ssh -p 5555 markm@localhost
```

We can attach gdb to the kernel as follows:

```
gdb /path/to/linux/kbuild/vmlinux
(gdb) target remote :1234
```

And that's it!

Troubleshooting

Here are some of the issues I ran across while trying to get this to work…

initramfs /init won't run :(

I got stuck here a lot with various errors, including:

- Kernel panic saying something about VFS and unknown device or root device
- Kernel cannot find /init
- Kernel cannot execute /init
- Kernel attempts to execute /init and fails

Unfortunately, all of these give pretty opaque error messages that all look pretty similar and give little information about anything.

For me, these errors turned out to be due to problems with how I created my initramfs. In particular, watch out for:

- Copying symlinks, rather than actual .so objects when looking at dependencies of bin/busybox with ldd
- Copying those files to the wrong paths
- Creating the init script in the wrong place (it should be directly inside of initramfs/)

Various help forums recommended passing "init=..." or "root=..." in the kernel boot params, and these didn't help for me.

initramfs cannot mount /dev/vd{a,b}

This happened when I forgot to convert an image from qcow2 to raw. I don't know why this is necessary, as QEMU does support qcow2 files, but for some reason it didn't work for me.

You can check what format your images are in with:

`qemu-img info foo.img`

where foo.img is your image file.

Guest devices (storage or NIC) not working

There are many possible causes for this.

Also recall that every time you change either cloud.yaml or cloud-net.yaml, you will need to regenerate both images. I recommend passing the `-snapshot` flag to QEMU, which tells it to write all disk changes to tmp files instead of changing your disk image.

Some sources of info:

- When the kernel boots, it's print messages have a ton of useful information about what devices it detects.
  - Check if it detects the devices you told QEMU to plug into your VM. For storage devices, you should see something about virtio or SCSI. For network devices the printing comes somewhat later (after initramfs) as far as I can tell, but you can at least tell if something related to networking happened.
  - If the kernel doesn't detect your devices, then it could mean you didn't compile the kernel to include the right drivers (make sure they are built-in, not modules).
  - Alternatively, it could mean you passed the wrong flags to QEMU somehow.
- After initramfs, the cloud-init system prints some useful information about what network interfaces are up. You should see an interface called enp0s5 or eth0 or something similar and it should be up and have an IP address associated with it. If not, then you may have misconfigured your cloud-net.yaml file. Make sure that the interface name in that file matches the name Linux assigned to the device (e.g. enp0s5).
- You can change the initramfs /init to "exec /bin/sh" instead of "exec switch_root …". This will give you a root shell just after the kernel boots and let's you dig around. The environment is very limited, but there are some useful things you can look at:
  - **`ls /dev/`**
    This will list all the devices detected so far. If /dev/vd{a,b} are not listed, then your kernel is not detecting the block devices.
    - Make sure you compiled virtio-blk support into the kernel.
    - Make sure you passed the right flags to QEMU.
    - Make sure your files are in raw, rather than qcow2 format.
  - **`ip addr`**
    This will list the network interfaces detected so far. If enp0s5/eth0/whatever is not listed, then your kernel is not detecting your vNIC. Note that you may see devices named "lo" or "sit0". These are not the devices you want. "lo" is the loopback device (i.e. localhost) and "sit" is a IP6-IP4 tunnel. You want something that looks like an actual NIC.
    - Make sure you compiled virtio-blk support into the kernel.
    - Make sure you passed the right flags to QEMU.

SSH/Login failures

If you are unable to log in to the account, even though the VM seems to boot properly, it may be that cloud-init didn't create your account properly. Some things to check:

- You are logging in with the account name used in cloud.yaml
- You are logging in with the key used in cloud.yaml
- The key in cloud.yaml is not interrupted with spaces, new lines, dashes, typos, or other copy-paste errors

Another way you can check for problems after you have tried booting the machine at least once is to mount the image files and look for problems. You can do this with a loopback mount, which allows you to mount an image as if it was a directory in your normal filesystem. NOTE: you probably need sudo permissions to run `mount`.

For the cloud.img:

```
mkdir mnt # doesn't matter where this is
mount cloud.img ./mnt
```

Now you can `cd mnt`, and you will be seeing the contents of the image. In `cloud.img`, you should basically just see 2-3 files, most notably, the cloud-init config files with your network settings, user, and SSH key. These are used by cloud-init the first time you boot up to set up your guest user account.

For the focal-server-cloudimg-amd64.img, things are a bit more complicated because this disk image has multiple partitions, and the one that will eventually be the root partition in the guest is only one of them. We need to find the offset into the image file of this partition before we can mount it. This can be done with `fdisk`:

```
fdisk -lu focal-server-cloudimg-amd64.img
```

Here is sample output:

```
Disk focal-server-cloudimg-amd64.img: 2.2 GiB, 2361393152 bytes, 4612096 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 1198075D-9ADB-45E8-ACB0-9D575124C46E

Device                              Start End Sectors  Size Type
focal-server-cloudimg-amd64.img1  227328 4612062 4384735  2.1G Linux filesystem
focal-server-cloudimg-amd64.img14   2048   102398192  4M BIOS boot
focal-server-cloudimg-amd64.img15  10240  227327  217088  106M EFI System

Partition table entries are not in disk order.
```

The first partition listed (the one with Type "Linux filesystem") is what we want. We see that it's start offset is 227328, and that the sector size is 512B (from the output at the top), so the start offset we want is 512 * 227328 = 116391936. Now we can mount (make sure to use your offset in the command):

```
mkdir mnt # if not already done
mount -o loop,offset=116391936 focal-server-cloudimg-amd64.img ./mnt
```

This time when you `cd mnt` you should see what looks like a standard root file system (like if you did `ls /` on any normal linux machine). You can `cd /home/$YOUR_USER_NAME` and look in `.ssh/authorized_keys`. If you have booted the machine before, but your home directory does not exist or your key is missing from `authorized_keys`, then you likely have an issue with your cloud-init configs somewhere.

NOTE: they will not exist before you boot the machine for the first time or if your machine doesn't boot far enough to run the cloud-init scripts. You should see lots of output about `cloud-init` stuff if the boot process gets far enough.

If you find something that needs to be fixed, regenerate BOTH image files (cloud.img and `focal-server-cloudimg-amd64.img`).

Consistent failures after fixing everything else…

The cloud-init init scripts do some special first-time setup the first time the VM boots. Most notably, it will create the user account and install your SSH keys. If something else is broken when this first-time setup first happens, you may end up with a VM you are unable to login with even after you fix the other problem.

To solve this, delete and regenerate both disk images (cloud.img and `focal-server-cloudimg-amd64.img`). Then, reboot the machine. You should see messages about generating users accounts and keys and stuff.

Bibliography

https://mgalgs.github.io/2012/03/23/how-to-build-a-custom-linux-kernel-for-qemu.html

https://stackoverflow.com/questions/36529881/qemu-bin-sh-cant-access-tty-job-control-turned-off

https://cloudinit.readthedocs.io/en/latest/topics/network-config-format-v2.html

https://www.linux-kvm.org/page/Virtio

https://wiki.qemu.org/Documentation/Networking#Network_Backends

https://gist.github.com/smoser/635897f845f7cb56c0a7ac3018a4f476

https://www.cs.utexas.edu/~rossbach/380L/lab/lab0.html

https://gist.github.com/george-hawkins/16ee37063213f348a17717a7007d2c79