CS380L: Advanced Operating Systems

Lab #0

The goal of this assignment is to be able to compile and boot the Linux kernel on the KVM-qemu virtual machine. If you have never built or installed a kernel before then please start this assignment early. We will use the Ubuntu distribution of Linux.

There are three parts to this system: the KVM-qemu virtual machine, the virtual machine's disk image, and your own build of the Linux kernel. Modern versions of Linux allow you to boot another version of Linux inside a virtual machine. In order to boot Linux you need something that looks like a disk that has all of the programs and the file system that Linux needs in order to run. Finally, we want to play with the kernel, so we want to compile it ourselves.

Unfortunately, the CS machines do not support KVM, so you will need to find a machine that does. If you can't find one, you can use CloudLab.

If you can't get set up with either a personal machine or cloud VM, let us know.

The instructions for this lab are deliberately non-explicit. One of the main goals is giving you the opportunity to read some documentation and figure things out for yourself, and almost all of the steps here can be accomplished many different ways. That said, if you're stuck on something for a long time, do ask questions. The failure modes of many of these tools can be cryptic.

Getting a VM running in KVM

- Get a VM with a copy of the latest LTS release of Ubuntu or Debian running in KVM on your machine
- An easy to get started is to use a Debian "no cloud" qcow2 image with QEMU (see here).
- You can also use an Ubuntu cloud image from here. If you use an installer, be careful of Ubuntu installer defaults, and don't let it partition your virtual disk in ways that don't make sense for a virtual machine! See this gist for help logging into an Ubuntu cloud image.
- Regardless of your VM construction method, make sure you include the openssh-client package if you build a VM yourself.

You should be able to get your VM running inside KVM by specifying the location of the VM with -drive or -hda. You may want to append "--snapshot" after the file name of your VM. This ensures that no changes are made to your image during an execution so you can do something dangerous and have the original image file preserved.

Once you've been able to get your VM running inside KVM, try installing a package using aptitude to verify your VM has network access.

Obtaining and building the kernel

Make sure you have the necessary dependencies to build the kernel. You'll need at least the following (these are the `apt` package names):

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

Download the latest stable Linux kernel release (v5.8.2 when we built it last) from kernel.org.

- Extract the source into <kernel dir>. (<kernel dir> is an absolute path)
- Create a separate build directory <kbuild>. (You can build the kernel in its source directory, but it is a bit more wieldy to use a separate build directory.)
- In <kbuild>, run

  ```
  make -C <kernel dir> O=$(pwd) defconfig
  ```

- This makes a configuration file (.config) with the default options selected.

- Make sure in your config file that `CONFIG_SATA_AHCI=y` (the above should set it automatically). This builds the SATA disk driver into the kernel, it is not built as a module. That will allow your kernel to boot off a (virtual) SATA drive without having to load a module to do it.
- Next build the kernel by running make in <kbuild>. (Consider researching and using the -j option to speed this up if you have more than 1 core on your machine. Similarly, consider researching 'make menuconfig' if you want to pare down features and speed up the build.)

Installing and Copying Kernel Modules

In this step you will install the kernel modules locally and then copy them to your image. Depending on how you copy your modules, it can take a while. Feel free to build a kernel that does not need modules. Just document what options you changed from default in your write up. **NOTE.** For this step of the process you will need to have the guestmount package installed.

- Create a separate <install_mod_dir> directory (perhaps a sibling of <kbuild>) that will contain the built kernel modules
- In <kbuild>, execute

  ```
  make INSTALL_MOD_PATH=<install_mod_dir> modules_install
  ```

- In <install_mod_dir> you should see one directory-- lib
- Next create a directory on your local file system (<mount_point>) onto which you will mount your VM
- Mount your VM to <mount_point> using guestmount
- cd to <mount_point> and view the contents on your VM's disk to verify that you have mounted it correctly
- Copy all the kernel modules from <install_mod_dir>/lib/modules to <mount_point>/lib/modules
- Now unmount your VM from <mount_point> using fusermount

After your copy your kernel modules, depending on how you boot, you might need to run `grub`-related utility (the bootloader) to notify it that you are booting a new kernel from this disk. Hint: if you are using the "-kernel" option below, you shouldn't need to.

For the curious, other methods for manipulating your virtual disk and file system from the host include:

- You can also use nbd to mount a qcow2 image, though I have found it terribly slow.
- You can convert the qcow2 to a raw disk image `qemu-img convert -O raw test.qcow2 test.raw`, and then do a loopback mount `mount -o loop,offset=32256 /path/to/image.img /mnt/mountpoint`. Note, your offset might be different from mine (maybe fdisk can help you), but loopback mounts are fast.

Here is a good reference on qemu images.

Booting KVM with your new Kernel

Now that you have compiled the kernel and copied the new kernel modules to your VM, you should be able to boot your VM with the new kernel.

- You will need to add -kernel "<kbuild>/arch/x86/boot/bzImage" and -append "root=/dev/sda1 console=ttyS0,115200n8" to the parameters you pass to KVM. (The root parameter specifies where the root partition of the hard disk is and the console parameter adds a serial console at boot so you can see boot messages.)
- You can also specify -nographic option, and use your current console to log in to the VM, but you need to create file /etc/init/ttyS0.conf in the VM image, which contains

  ```
  # ttyS0 - getty
  #
  # This service maintains a getty on ttyS0 from the point the system is
  # started until it is shut down again.
  start on stopped rc or RUNLEVEL=[2345]
  stop on runlevel [!2345]
  respawn
  exec /sbin/getty -L 115200 ttyS0 vt102
  ```

- If you have trouble booting your VM, you might want to explore some of the other command line options to KVM such as -serial or -gdb.
- Once the VM has booted completely, you should be able to login and try installing another package from aptitude.

Your writeup

Please complete a short writeup that you will print out and bring to class.

Identify completely your hardware and software.

Explain any changes you made to the kernel build process and why.

Report your qemu command line.

Booting, kernel modules, and discovering devices

Boot the Kernel, and run `dmesg` to capture the kernel output from the boot. In your writeup, report the elapsed wall clock time for your boot, and compare that with the time reported by the kernel. Are they the same? Why or why not?

For each device, quote the parts of the dmesg output that show the kernel discovering that device, and then report what kind of device it is, and how you made your determination. You may also use `lspci`.**Please only quote a maximum of 3 lines.**

Tracing the kernel

- Compile the following code, transfer the program to your simulated disk and run the program.

```
#include<unistd.h>
#include<fcntl.h>
int main()
{
    int fd = open("/dev/urandom", O_RDONLY);
    char data[4096];
    read(fd, &data, 4096);
    close(fd);
    fd = open("/dev/null", O_WRONLY);
    write(fd, &data, 4096);
    close(fd);
}
```

- Attach gdb to the kernel. You will find that you need to compile your kernel with debug symbols, disable KASLR, and depending on your kernel version, support for kvm, and proper CONFIG_DEBUG* options. See these instructions. For each change that you make to the default .config file (including those mentioned in this page), explain why is that change required in one sentence.
- Set a conditional breakpoint in spin_lock in kernel code that will only stop execution if the above process is running. (Note: spin_lock might be an inline function or macro; please check the kernel source of your version to find out the actual symbol name where you need to set the breakpoint.)
    - To find the PID of the currently executing process from GDB, you will need to inspect (condition on) the value stored in the task_struct of the current process.
    - To find out the address of the task_struct of the current process, you can use the `$lx_current()` helper function provided by GDB.
- Find three instances of spin_lock being called from three substantially different contexts in the kernel, where you define substantially different. Report exactly how you set the breakpoints, in what functions, how you made sure your process was running, etc. For each of the three calls to spin_lock that you choose, put the minimum number of lines of the kernel backtrace into your report and then summarize what is going on in the kernel in AT MOST two sentences.
- Find the entry point for system calls in your version of the kernel and set a breakpoint there. After the breakpoint is hit (this can be outside of the process context), step through the code in GDB and answer the following questions:

        - What is the value of CR3 register value when this breakpoint is hit? You can use `info registers` on QEMU console to get this value. If you have more than one CPU provisioned on your VM, make sure that you are looking at the correct register value.
        - Do you see code to update the CR3 value? If so, what is the CR3 value after this code is executed? Briefly explain why is this required? What happens when you repeat this after updating your kernel command line to pass `nopti`? You can use `–append` in your QEMU command line to update the kernel command line.
    - In your write up, briefly explain the difference between /dev/random and /dev/urandom.

Building a KVM+GDB-friendly kernel

- Backup your .config in kbuild dir.
- cd into kbuild
- Make sure that the following options in .config file are set as stated below.
    - CONFIG_DEBUG_INFO y
    - CONFIG_DEBUG_INFO_DWARF4 y
    - CONFIG_GDB_SCRIPTS y
    - CONFIG_GDB_INFO_REDUCED n
    - CONFIG_KGDB y
    - CONFIG_FRAME_POINTER y
    - CONFIG_SATA_AHCI y
    - CONFIG_KVM_GUEST y
    - CONFIG_RANDOMIZE_BASE n
    - CONFIG_SMP y

Comments

Please think of your report like the papers we are reading for the class. Organize the information that you gathered and present it logically. For example, early in your report specify your experimental platform, including the hardware, the OS, whether you are running qemu or a hypervisor, etc. Include anything relevent to understand the results, but keep your description concise. Of course balancing completeness and concision is difficult, but that balance is generally well accomplished by the (modern) papers we read.

Some people use `virsh` to manage their VM. This can be sort of useful, but it obscures exactly what is going on, for example with your exact command line to qemu. I would suggest not using it.

Please report how much time you spent on the lab.