LAB #0 Booting KVM with new Kernel

CS308L: Advanced Operating Systems

Hochan Lee
UT EID: hl26847

1 Objective

The main goal of the lab #0 is to understand how to compile and boot the Linux virtual machine. To accomplish successful virtual machine booting, I exploit KVM and QEMU to generate and actually run the virtual machine (VM). The specification suggests students to use an old software that is verified and generally used. However, this report treats results that are collected through the recent Linux version, Linux 4.20. Therefore, I tried to experiment and solve not only popular issues, but also not well-known issues generated by using the recent kernel during the VM construction.

2 Hardware/Software Specification

The host machine consists of a single socket system with Intel i5-2467M CPUs with 2 cores and 4 hyper-threads. Also, it has 8GB of DRAM and 3072K last level cache and Transparent Huge Pages (THP) is disabled. The VM is set as a single socket 1 core, 2 hyper threads and 4GB DRAM whose THP is disabled. Those settings are half of the host machine's resources since the host machine also required HW resources for real-time VM analysis such as GDB. Last Level Cache (LLC) size is set by default as 16MB but I suspect that it would affect a performance of the VM. Therefore, I will perform experiments and adopt a proper cache size for performance by changing it after this lab.

To construct the VM, I used Virtual Machine Builder 1.5.1 which is graphic based virtual machine manager application. Users can generate a virtual machine just by passing ISO file path. The generated VM runs based on KVM and QEMU. KVM supports a full-virtualization system for Linux. QEMU supports both machine emulation and virtualization, but in this context, it virtualizes x86-64 architecture. I installed related packages through Ubuntu Advanced Packaging Tool (APT) except QEMU: qemu-kvm, libvirt-daemon-system, libvirt-clients, bridge-utils and libvirt-bin. I had to clone QEMU source code from the git repository, modified a source code and compiled it directly because of the compatibility issue between QEMU and GDB of the recent Linux kernel. This issue is treated in the later section: "Running a VM in KVM". I installed Ubuntu 18.04.1 LTS that adopts Linux kernel 4.15 for the initial VM construction. Then, I replaced the kernel with Linux kernel 4.20, the more recent version. Inside the VM, vim, gcc, g++ and openssh-client are used for the analysis. Finally, guestmount, guestfish,. and fusemount are used to mount and un-mount the VM disk.

3 Running a VM in KVM

The Virtual Machine Builder automatically and simply creates a VM from specified ISO image on a new virtual disk. Therefore, instead of introducing basic VM construction, this section focuses on how to replace default Linux kernel with the recent version and which additional changes are required for QEMU and kernel compilation process. Based on a purpose, two types of kernel are built: normal (non-debug) and debug (gdb-friendly).

3.1 Building and applying new Kernel

I downloaded Linux 4.20 kernel .tar file from kernel.org. After extracting source codes, I made an additional directory named 'kbuild' to manage object files separately from source codes. This method is a safer way than integrating and managing both files. Before building the kernel, I generated configuration file that specifies kernel build options by using below command.

```
# yes " " | make -C <kernel_dir> O=$(pwd) config
```

The generated configuration file is set by the default build options. I modified CONFIG_SATA_AHCI option from 'n' to 'y' since it allows me to boot VM through a SATA drive without loading any module. VM is operated on the virtual SATA drive generated by QEMU. After setting the configuration, I started to build Linux kernel by using below command.

```
# make -j3
```

For the kernel build, I used -j option. 'J' option is to specify the number of jobs to run simultaneously. There are a lot of independent source codes on Linux kernel and so, this option is effective to speed up the compilation. In general, people empirically know using (core+1) for the j option is enough. I also followed this assumption, -j3 option (=the number of cores, 2+1).

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

As a next step, I installed the kernel modules by using upper command. To copy the installed modules, I should mount VM disk to a directory of the host machine by using guestfish and guestmount. Guestfish is a shell managing a file system of VM. With the below commands, I figured out that /dev/sda1 is a device where the file system is located.

```
# sudo guestfish -a <vm_image>.qcow2 # enter the shell

# <fs> list-filesystems
```

Next, VM disk was mounted on the host machine directory and copied the installed modules to the mounted VM disk.

```
# sudo guestmount -a <vm_image>.qcow2 -m /dev/sda1 <mount_dir>

# sudo cp<install_mod_dir>/lib/modules/* <mount_dir>
```

After updating VM's kernel, I detached the mounted directory by using fusemount.

```
# sudo fusermount -u <mount_dir>
```

Linux kernel that is built through those progresses does not support kernel debugging function. To enable the kernel debugging, I modified kernel configuration options and QEMU codes. They will be explained in the later section.

3.2 Booting KVM with new Kernel

I used qemu-system-x86_64 in order to boot KVM with new Kernel.

```
# sudo qemu-system-x86_64 -enable-kvm -kernel kbuild/arch/x86_64/boot/bzImage
-append "root=/dev/sda1 console=ttyS0, 115200n8 -serial mon:stdio
-hda aos-vm-ubuntu18.04.qcow2 -cpu host
```

"-enable-kvm" option means I want to run VM with full virtualization support from KVM. "-kernel" option specifies kernel compressed image. "-append" option uses a kernel command line. In this case, I specified /dev/sda1 as a root directory and ttyS0, 115200n8 as a console in order to check boot messages. "-serial" option specifies host character device. Through this, I could redirect serial port to devices what I specified. "mon" option allows the monitor to be multiplexed onto the another serial port. It was useful for me since I could use interrupt signal for the VM on the console. "-hda" option specifies the disk image. Lastly, "-cpu" option specifies what CPU model will be used. I used "-cpu host" option in order to exploit most of the CPU functions. After booting the VM with the new kernel, I snapshotted the intial VM.

3.3 Additional Changes for GDB-friendly Kernel

Another goal of the lab is to debug and analyze Linux kernel by using GDB. To allow GDB to debug the kernel, several kernel option should be modified as mentioned on the description: compiling kernel with debug information, generating dwarf4 debug information, providing GDB scripts for kernel debugging, enabling KGDB option and including all symbols in kallsysm. However, since I am using the most recent version of Linux kernel, I had to do one more modification on the configuration: Kernel Address Space Layout Randomization (KASLR). This is the recently adopted configuration option that randomizes both virtual and physical addresses in order to prevent from kernel hacking. I figured out that if I turn on the option, GDB cannot find a breakpoint address and just skip it. Otherwise, GDB successfully stops at the breakpoint.

I also had to modify QEMU slightly. This is because GDB cannot approach to the VM's register information. I had to return the size of register on the gdb_read_register function on the QEMU's gdbstub.c. Below is the modification.

```
qemu/getstub.c

static int gdb_read_read_register(CPUState *, uint8_t *, int )
..
- return 0;
+ #ifdef TARGET_X86_64
+     return 8;
+ #endif
+     return 0;
}
```

3.4 Booting KVM with GDB friendly Kernel

Booting the kernel that allows GDB debugging requires more options than before.

```
# sudo qemu-system-x86_64 -enable-kvm -kernel kbuild/arch/x86_64/boot/bzImage
-append "root=/dev/sda1 console=ttyS0, 115200n8 nokaslr" -serial mon:stdio
```

```
    -hda aos-vm-ubuntu18.04.qcow2 -cpu host -s -S
```

"nokaslr" option explicitly specifies that disables the kernel's KASLR option. However, I already ghave done it with configuration phase before building it. "-s" option opens a gdbserver on TCP port 1234. Through the port number, gdb connects to the VM and can debug it. "-S" option blocks CPU at startup until GDB transfers 'continue' signal.

4 Booting Analysis

This section introduces observations about what happened during the booting and how long it takes to boot. In addition, it analyzes why time differences occurred between the elapsed wall clock time and the time reported by the kernel. Lastly, it reports what kind of device is discovered by the machine during the boot. To analyze those information, I exploited "dmesg" generated by the kernel during the boot.

Time Comparison: Wall Clock vs Dmesg Time

First, the elapsed wall clock time was measured by iPhone timer. During the two measurements, the average time was 17.62 seconds. Second, I got the time duration reported by the kernel during the booting from dmesg log. As soon as booting was completed, I printed out the dmesg log and extracted booting time from it was about 11.97 seconds. Below is the last line of the dmesg.

Last line of the dmesg

```
...
[ 8.169474] IPv6: ADDRCONF(NETDEV_UP): enp0s3: link is not ready
[ 8.178019] e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 8.181938] IPv6: ADDRCONF(NETDEV_UP): enp0s3: link is not ready
[ 8.182987] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
[ 11.973757] 01-ifupdown (2050) used greatest stack depth: 11368 bytes left
```

Therefore, there is about 6 second differneces between the actual duration and the duration reported by the kernel. This is because "dmesg" calculates the timestamp by using the kernel ring buffer. Whenever new messages come into the kernel, the kernel ring buffer records the messages and calculates the timestamp. Therefore, "dmesg" calculates just uptimes which kernel is in not sleep or suspension, but active status for the timestamp. The time differences between them are came from the reason.

Discovered Device Analysis

To track discovered devices during the booting, I found devices that were attached to the Linux kernel after booting by using "lspci". The "lspci" command displays information about PCI buses in the system and devcies connected to them. Below is the printed device lists. Each line on the output below shows a PCI device. The first column number means a given bus number and the second column represents the device name. Based on the printed results from the lspci, I attempted to find the devices that were connected during the booting by searching "PCI" labels from the dmesg. If a log of the demsg contains any PCI bus number, then I decided it as the information related to the device detection.

lspci

```
lhclhc-vm: /Practice/lab0$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller
(rev 03)
```

Below is the demsg logs that are related to the device discoveries. In the above of the each line, the comments explain what device is related to that dmesg line. I made that determination by comparing PCI bus number respectively. In order to read comfortably, I reorderd the dmesg based on the PCI bus number.

lspci

```
# Below is the host bridge. This device's PCI bus number is 00:00.0.
[ 0.340883] pci 0000:00:00.0: [8086:1237] type 00 class 0x060000
# Below is the ISA bridge. This device's PCI bus number is 00:01.0.
[ 0.341805] pci 0000:00:01.0: [8086:7000] type 00 class 0x060100
# Below is the IDE interface. This device's PCI bus number is 00:01.1.
[ 0.343002] pci 0000:00:01.1: [8086:7010] type 00 class 0x010180
[ 0.352792] pci 0000:00:01.1: reg 0x20: [io 0xc040-0xc04f]
[ 0.356835] pci 0000:00:01.1: legacy IDE quirk: reg 0x10: [io 0x01f0-0x01f7]
[ 0.357780] pci 0000:00:01.1: legacy IDE quirk: reg 0x14: [io 0x03f6]
[ 0.358784] pci 0000:00:01.1: legacy IDE quirk: reg 0x18: [io 0x0170-0x0177]
[ 0.359782] pci 0000:00:01.1: legacy IDE quirk: reg 0x1c: [io 0x0376]
# Below is the southbridge for the PCI based devices.
# This device's PCI bus number is 00:01.3.
[ 0.361213] pci 0000:00:01.3: [8086:7113] type 00 class 0x068000
[ 0.362144] pci 0000:00:01.3: quirk: [io 0x0600-0x063f] claimed by PIIX4 ACPI
[ 0.362803] pci 0000:00:01.3: quirk: [io 0x0700-0x070f] claimed by PIIX4 SMB
# Below is the VGA compatible controller.
# This device's PCI bus number is 00:02.0.
[ 0.364559] pci 0000:00:02.0: [1234:1111] type 00 class 0x030000
[ 0.365806] pci 0000:00:02.0: reg 0x10: [mem 0xfd000000-0xfdffffff pref]
[ 0.369791] pci 0000:00:02.0: reg 0x18: [mem 0xfebf0000-0xfebf0fff]
[ 0.377790] pci 0000:00:02.0: reg 0x30: [mem 0xfebe0000-0xfebeffff pref]
[ 0.398093] pci 0000:00:02.0: vgaarb: setting as boot VGA device
[ 0.398771] pci 0000:00:02.0: vgaarb: VGA device added:
[ 0.398792] pci 0000:00:02.0: vgaarb: bridge control possible
# Below is the ethernet controller. This device's PCI bus number is 00:03.0.
[ 0.378295] pci 0000:00:03.0: [8086:100e] type 00 class 0x020000
[ 0.379791] pci 0000:00:03.0: reg 0x10: [mem 0xfebc0000-0xfebdffff]
[ 0.381781] pci 0000:00:03.0: reg 0x14: [io 0xc000-0xc03f]
[ 0.391784] pci 0000:00:03.0: reg 0x30: [mem 0xfeb80000-0xfebbffff pref]
```

To sum up, all the devices detected by the "lspci" were detected and attached to the Linux kernel during the booting time.

5 Tracing Kernel

The final objective of the lab is to trace and debug the kernel, while executing an example program. The example source code performs open/read/write/close operations by using "/dev/urandom" and "/dev/null" device descriptors. Specifically, we also need to track and understand when and where spin_locks are called, and how it is used during those operations. This tracing practice was executed on the kernel whose GDB option was enabled; the detail setting method is mentioned on the section named "Running a VM in KVM- Additional Changes for GDB-friendly Kernel."

5.0.1 Tracing a PID of the running process

To accomplish this objective, at first, I needed to get a PID of the running process. This PID information is used by GDB not only to detect function calls that were invoked by a specific process, but also to make a conditional breakpoint. Fortunately, GDB community provided a convenient package named "lx" for the kernel debugging. By exploiting "$lx_current().pid" and "$lx_current().comm" functions of the package, I could easily get the PID. The former function prints out PID and the latter function prints out process name of the currently highest priority process. I verified the PID by comparing the process name and the target process name. Below box lists the mentioned GDB commands.

gdb commands

```
# connect to the VM and start debugging
(gdb) target remote :1234
# get PID of the highest priority process
(gdb) $lx_current().pid
# get process name of the highest priority process
(gdb) $lx_current().comm
```

However, there is a constraint that the process must be running during debugging. Since the example process performs only simple and short tasks, the process was finished and released immediately, and it was hard to stop the VM before the process end. To solve this issue, I used a trick; just put an empty loop and delays the execution finish. By modifying the source code like below, I could get the process information and exploited it for debugging. The front loop took 2 to 3 seconds, so it was enough to stop the VM before the process finished.

```
#include<unistd.h>
#include<fcntl.h>
int main()
{
    \textcolor{red}{for (int i = 0; i < 50000000; i++) ;}
    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);
}
```

Lastly, I set a conditional breakpoint by using the below command in order to track spin_lock() call. Note that I made a breakpoint for __raw_spin_lock() because the spin_lock() could be either an inline function or a macro; it is determined depending on a target architecture. General way to exploit GDB is to make a breakpoint at a normal function that is neither inline nor macro. However, GCC higher than version 4 started to support inline function call breakpoint if the compilation is performed with gdwarf2 option or higher versions. Since I compiled the new kernel with gdwarf4 option, I could set the breakpoint at an inline function. But, still there is no way to set a breakpoint for a macro. So, I tracked function calls that are occurred in spin_lock() and finally, I decided to set the breakpoint at __raw_spin_lock(). The below command stops the VM whenever the specified process called __raw_spin_lock().

gdb commands

```
(gdb) b __raw_spin_lock if $lx_current().pid == [TARGET PROCESS PID]
```

5.0.2 Understanding usage of spin_lock

During the debugging, there were mainly three instances of __raw_spin_lock call from the kernel. Below is the logs that are generated by GDB when the upper condition was satisfied. To summarize the log, I removed unimportant logs.

gdb log - first breakpoint

```
#0 _raw_spin_lock (lock=0xffff88800743c4a4) at kernel/locking/spinlock.c:144
#1 0xffffffff810bac16 in handle_irq_event (desc=0xffff88800743c400) at kernel/irq/handle.c:208
...
#4 handle_irq (desc=0xffff88800743c400) at arch/x86/kernel/irq_64.c:78
#5 0xffffffff81c016e1 in do_IRQ (regs=0xffffc90000bb7f58) at arch/x86/kernel/irq.c:246
#6 0xffffffff81c0094f in common_interrupt () at arch/x86/entry/entry_64.S:583
...
```

gdb log - second breakpoint

```
#0 _raw_spin_lock (lock=0xffffffff82406a04 <jiffies_lock+4>) at linux-
4.20.6/kernel/locking/spinlock.c:144
...
#3 tick_do_update_jiffies64 (now=1031321200702) at linux-4.20.6/kernel/time/tick-
sched.c:70
...
#6 0xffffffff810e80c7 in tick_sched_timer (timer=0xffff88800781bfc0) at linux-
4.20.6/kernel/time/tick-sched.c:1267
...
at linux-4.20.6/kernel/time/hrtimer.c:1460 #9 0xffffffff810d8ab2 in hrtimer_interrupt
(dev=<optimized out>) at linux-4.20.6/kernel/time/hrtimer.c:1518
...
#12 0xffffffff81c0154f in apic_timer_interrupt () at linux-
4.20.6/arch/x86/entry/entry_64.S:807
...
```

gdb log - third breakpoint

```
#0 _raw_spin_lock (lock=0xffff888007820a80) at kernel/locking/spinlock.c:144
...
#2 scheduler_tick () at linux-4.20.6/kernel/sched/core.c:3045
#3 0xffffffff810d76a2 in update_process_times (user_tick=0) at kernel/time/timer.c:1641
...
#5 0xffffffff810e80d7 in tick_sched_timer (timer=0xffff88800781bfc0) at linux-
4.20.6/kernel/time/tick-sched.c:1274
#6 0xffffffff810d832a in __run_hrtimer (flags=<optimized out>, now=<optimized out>,
timer=<optimized out>, base=<optimized out>, cpu_base=<optimized out>) at ker-
nel/time/hrtimer.c:1398
...
#8 0xffffffff810d8ab2 in hrtimer_interrupt at kernel/time/hrtimer.c:1518
...
#13 0xffffffff81c0154a in apic_timer_interrupt () at arch/x86/entry/entry_64.S:807
...
```

Through these information, I could get meaningful information only from the second backtracking information. **The second spin lock was called whenever the jiffies was increased.** To understand the first and the third spin lock context, I retried tests with two breakpoints that were for spin_lock() and __raw_spin_lock(); I figured out that spin_lock() is an inline function in the x86_64 architecture. Then, I tracked a spin_lock() call that had the same address with the spin_lock() that was printed on the backtracking of the __raw_spin_lock(). Through this matching process, I figured out that **the first lock and the third lock were called when the file was opened or closed, and when the character device was opened and closed.**

6 Differences between /dev/random and /dev/urandom

Both are used to generate random number. However, in the Linux kernel envrionment, they show different actions when there is no entropy in the system for random number generation. First, /dev/random device only returns random bytes within entropy pool. Therefore, if the entropy pool becomes empty, the device would be blocked until additional entropy (noise) is gathered. Second, /dev/urandom also returns random bytes as requested. However, if the entropy pool becomes empty, it would not block an operation and generate new random data using algorithms such as MD5 or SHA.

7 Time Consuming

I took about 24 hours for the lab. Since I tried to accomplish the lab with the recent version of software, I needed to consider and treat several configuration issues. In addition, since my English skill is not good enough, I spent more than 5 hours to write the document.