Advanced OS - Lab 0

Dinesh Reddy Chinta; dc47444

August 3, 2024

1 Abstract

We explore the building and booting of a Virtual Machine running a Linux Kernel on the QEMU-KVM hypervisor. To demonstrate the advantages of maintaining a custom kernel, we compile the kernel with debugging capabilities and show some debugging techniques about how to investigate Linux data structures through examples. We investigate some key aspects like Timer, Device management and Page Table management of Virtual Machines.

2 Setup and Specifications

2.1 Host Hardware Specifications

I used an HP Pavilion Notebook (M2W75PA#ACJ) desktop class computer with Intel i5-5200U x86_64 CPU running at 2.2 GHz, 32KiB L1, 256KiB L2, and 3MiB L3 caches, respectively. It has 8 GB Samsung DDR3 RAM, M471B1G73QH0-YK0, running at 1600MHz clock frequency.

2.2 Host Software Specifications

The Host is running a Ubuntu 20.04.5 LTS Operating system with Linux 5.4.0-137-generic kernel version. The Hardware has KVM support.

Listing 1: Host Config

```
# hostnamectl
    Static hostname: UB20
          Icon name: computer-laptop
            Chassis: laptop
         Machine ID: 53e8104175234847a27d451b6e75a712
            Boot ID: 1024d4614f6e4ea2b825d0b7e8b1f8bf
  Operating System: Ubuntu 20.04.5 LTS
            Kernel: Linux 5.4.0-137-generic
      Architecture: x86-64

# uname -a
Linux UB20 5.4.0-137-generic #154-Ubuntu SMP Thu Jan 5 17:03:22
UTC 2023 x86_64 x86_64 x86_64 GNU/Linux

# grep -o vmx /proc/cpuinfo
vmx
```

2.3 Guest Hardware/Software Specifications

It should be noted that I used two different Virtual Machines for the lab.

The first Guest Operating System is a custom Embedded Operating System, "Buildroot," with Linux Kernel 6.1.7 as our custom-compiled kernel for experiments. This has been mainly used wherever debugging the kernel becomes important.

The second Guest Operating System is "Ubuntu 22.04.1 LTS." I used this to inspect the connected devices and to measure the difference between wall-clock time and kernel time while booting.

Listing 2: Guest Config

```
# cat /etc/os-release
NAME=Buildroot
VERSION=2022.11-1026-g5ec326fb97
ID=buildroot
VERSION_ID=2023.02-git
```

```
PRETTY_NAME="Buildroot 2023.02-git"

# uname -a
Linux buildroot 6.1.7 #3 SMP PREEMPT_DYNAMIC Thu Jan 26 21:20:51
CST 2023 x86_64 GNU/Linux
```

2.4 QEMU

We are using QEMU-KVM version, Debian 1:4.2-3ubuntu6.24. Here, the important options to note are the -s and -S, which enable us to stop QEMU instance until we connect a GDB.

Listing 3: QEMU command to start VM in debugging mode

```
# sudo qemu-system-x86_64 --version
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.24)

#qemu-system-x86_64 -s -S \
                -kernel arch/x86/boot/bzImage \
                -boot c \
                -m 2049M \
                -hda ../../code/buildroot/output/images/rootfs.ext4 \
                -append "root=/dev/sda rw console=ttyS0,115200 acpi=off nokaslr" \
                -serial stdio \
                -display none
```

3 Compiling & Booting the Kernel

3.1 Getting a new VM instance

I used VirtManager to install the new Ubuntu22 virtual machine instance. We can download Ubuntu Desktop from here. As described in this tutorial you can install the VM into a virtual disk with required number of supported CPUs and RAM. To start the VM, you can either use "virsh" or the QEMU command provided above by removing kernel related options – -kernel and -append.

3.2 Compiling the Kernel

As described in the lab document, I first created a .config file by using the first command in Listing 4. And then, to enable debugging, I modified the .config file using menuconfig method by executing the command2 in Listing4. The .config options described in the next section are enabled to facilitate kernel debugging.

Listing 4: Compiling the Kernel

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

3.3 Config options

All the debugging options increase the build time and kernel footprint and slow down the kernel. Also, in my version of the kernel, not all of the options are still supported. For example, CONFIG_FRAME_POINTER is not available in Linux 5.4.230 when compiling with gcc 9.4.0 version.

- CONFIG_DEBUG_INFO: Global debug option to enable several debugging symbols for a debugger. When this is disabled other dependent debug options are ignored.
- CONFIG_DEBUG_INFO_DWARF4: Specifies the format DWARF4 (Debugging With Arbitrary Record Format) of the debugging symbols to be built with.
- CONFIG_GDB_SCRIPTS: Enables the Kernel to be debugged with scripts in gdb. This will help if you want to write gdb command scripts while debugging.
- CONFIG_GDB_INFO_REDUCED: This would optimize the memory footprint of the kernel while providing the debugging information for debugging with GDB. This option is helpful for all embedded systems that have memory resource constraints. We disabled it to ensure we have access to a complete set of debugging information.

Wall Clock Time	Kernel Reported Time
2.91	1.92
2.91	1.94
3.01	1.92
2.81	1.81
2.84	1.93
Average = 2.89	1.90

Table 1: Experiment of the kernel boot time measured by Wall Clock and the time reported by kernel in dmesg log.

- CONFIG_KGDB: This is a specific option to enable Kernel debugging with GDB (GNU Debugger). While the other options set general debugging support, this is to specify that the debugging tool is GDB.
- CONFIG_FRAME_POINTER: This enables the assembly portions of the Kernel to use an explicit Frame Pointer — a register relative addressing — to deal with stack memory. The compiler can sometimes get rid of stack pointers for performance. However, disabling this would help us analyze the stack trace in the debugger.
- CONFIG_SATA_AHCI: This would enable the kernel to be built with modules for QEMU booting using a virtual CDROM or disk file and interact with the devices after booting.
- CONFIG_KVM_GUEST: This would build our kernel to include drivers for interacting with hardware when it is run as Guest Operating System. Without this option, the compiled kernel can't be used to run as a virtual machine.
- CONFIG_RANDOMIZE_BASE: This is a security feature that enables address space layout randomization (ASLR) which makes it harder for an attacker to exploit memory-based vulnerabilities by randomizing the memory locations of key data structures, such as the kernel and library code. However, when debugging the Linux kernel, ASLR can make it more difficult to match up the memory addresses in a crash dump or other debugging output with the corresponding source code or data in the kernel.
- CONFIG_SMP: This compiles the kernel with support for Multiple CPU cores (Symmetric Multiprocessing) so that we can configure the QEMU command line to use this option to run a multicore kernel.

3.4 Guest OS Timer

I have used an external timer to measure the wall clock time. I also repeated the experiment 5 times to minimize the statistical aberration. Table 1. shows the 5 measured wall clock times and the time reported by the kernel.

The measured wall clock time and the time reported by the kernel in the "dmesg" log are different. The average kernel reported time, 1.90s, is less than the average wall clock time, 2.89, by a second. While there is still a response time delay (for me to have stopped the wall clock after seeing the login prompt) the 1-second delay is still significant to be attributed to the response time.

In a host, the kernel keeps track of the time based on the timer interrupts it receives from hardware timer, typically around 1000Hz. In a guest virtual machine, however, where there is a virtual clock instead of a hardware clock, a backlog gets generated due to the virtual machine not updating the clock, if the CPU is not available for the guest. The hypervisor corrects the guest operating system time whenever the backlog grows over a threshold.

In our case, the virtual clock maintained by the VM fell behind the host clock since it did not get the opportunity to update the clock when the hardware clock interrupts occurred. Hence, we observe that there is an approximately 1-second difference between the reported kernel time and the actual wall clock time.

Device ID:Vendor ID	Class
8086:1237	Host bridge
8086:7000	ISA bridge
8086:7010	IDE interface
8086:7113	Bridge
8086:100e	Ethernet controller
1b36:0100	VGA compatible controller
8086:2934	USB controller
8086:2935	USB controller
8086:2936	USB controller
1af4:1002	Unclassified device

Table 2: Device and it's corresponding Class.

3.5 Connected Devices

Classifying devices is important because it allows the operating system to understand and properly interact with the connected devices. By identifying the class of a device, the OS can use the appropriate driver or software to communicate with the device and make it available for use by the system and user.

The "lspci" command displays information about all PCI buses and connected devices in the system. To get the Class, Vendor and device ID of a specific device, I used the lspci -nn command. Table.2 shows the mapping between the connected devices and the corresponding classes. To extract the device getting discovered and connected information by the kernel, I used "grep" command to search in the "dmesg" log as shown in the Listing 1 (DMESG log for each device).

Listing 5: DMESG log for each device

```
root@ub22:/home/os# dmesg | grep -i "8086:1237"
[    0.073842] pci 0000:00:00.0: [8086:1237] type 00 class 0x060000
root@ub22:/home/os# dmesg | grep -i "8086:7000"
[    0.074715] pci 0000:00:01.0: [8086:7000] type 00 class 0x060100
root@ub22:/home/os# dmesg | grep -i "8086:7010"
[    0.075442] pci 0000:00:01.1: [8086:7010] type 00 class 0x010180
root@ub22:/home/os# dmesg | grep -i "8086:7113"
[    0.078558] pci 0000:00:01.3: [8086:7113] type 00 class 0x068000
root@ub22:/home/os# dmesg | grep -i "1b36:0100"
[    0.079893] pci 0000:00:02.0: [1b36:0100] type 00 class 0x030000
root@ub22:/home/os# dmesg | grep -i "8086:100e"
[    0.094209] pci 0000:00:03.0: [8086:100e] type 00 class 0x020000
root@ub22:/home/os# dmesg | grep -i "8086:2934"
[    0.099236] pci 0000:00:04.0: [8086:2934] type 00 class 0x0c0300
root@ub22:/home/os# dmesg | grep -i "8086:2935"
[    0.102629] pci 0000:00:04.1: [8086:2935] type 00 class 0x0c0300
root@ub22:/home/os# dmesg | grep -i "8086:2936"
[    0.105050] pci 0000:00:04.2: [8086:2936] type 00 class 0x0c0300
root@ub22:/home/os# dmesg | grep -i "8086:293a"
[    0.107500] pci 0000:00:04.7: [8086:293a] type 00 class 0x0c0320
```

4 Debugging & Tracing the Kernel

4.1 Setup

I started QEMU in one terminal and in a separate terminal I started and connected GDB to the remote QEMU node. Some of the loaded symbols were helpful in the kernel investigation by allowing us to observe the kernel data structures and set conditional breakpoints. Traditionally, people use the "comm" attribute of the task_struct to set conditional breakpoints. However, my Kernel did not have "malloc" included. Hence, it disallowed me to use "comm" parameter for comparison, which would require a string comparison.

Listing 6: Error accessing comm parameter of task$_struct$

```
(gdb) b show_cpuinfo if $lx_current().comm == "random"
Note: breakpoint 1 also set at pc 0xffffffff8103b720.
Breakpoint 3 at 0xffffffff8103b720: file
.../linux-6.1.7/arch/x86/kernel/cpu/proc.c,
line 67.
```

To work around this, I Introduced a few lines in the application code related to reading CPU info to isolate the thread for debugging. In the application, I read the file "/proc/cpuinfo". Before running

the application, I set a breakpoint in the kernel at show_cpuinfo function. After the Kernel pauses in the function, we get the PID of the process. From here on, we use this PID to set conditional breakpoints.

Listing 7: Modified application code

```
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>

int main()
{
        char data[4096];

        int fd2 = open("/proc/cpuinfo",
            O_RDONLY);
        read(fd2, &data, 4096);
        close(fd2);

        ...
}
```

Listing 8: GDB functioning with modified code

```
(gdb) p $lx_current().pid
$9 = 144
```

5 Spin locks

In my kernel, spin_lock macro uses __raw_spin_lock() function in "kernel/locking/qspinlock.c" file. The three substantially different subsystems where Linux Kernel uses spin-locks that I worked on are:

1. Timers: spin_lock is used to synchronize access to the timer data structures, such as the timer wheel and the per-CPU timer queue. When the APIC timer raises an interrupt, any CPU can update the jiffies. To prevent race condition, the kernel acquires a spin_lock before updating. Setting a breakpoint for this case wasn't so difficult. Timer interrupts occur often that even setting an unconditional breakpoint inside __raw_spin_lock would let us isolate this scenario.

Listing 9: Backtrace for Timer Interrupt

```
(gdb) bt
#0  __raw_spin_lock (lock=0xffffffff82807a40 <jiffies_lock>)
    at ...linux-6.1.7/include/linux/spinlock_api_smp.h:132
...
#2  0xffffffff8110ccd0 in tick_do_update_jiffies64 (now=77709956346)
    at ...linux-6.1.7/kernel/time/tick-sched.c:91
...
#10 0xffffffff81ccac89 in sysvec_apic_timer_interrupt (regs=0xffffc9000021fd58)
    at ...linux-6.1.7/arch/x86/kernel/apic/apic.c:1107
```

2. File system: spin_locks are used to synchronize access to the file system data structures, such as inode and dentry caches. For example, the kernel locks file descriptor table while assigning a new file descriptor after trapping into open system call. In GDB, I set a breakpoint in fd_open kernel function to capture the spin_lock usage. The lock was necessary to prevent race condition between multiple threads trying to allocate file descriptors. I set a conditional breakpoint – checking with PID match – at alloc_fd function to capture the scenario.

Listing 10: Relevant Application Code

```
...
FILE* fp = fopen("my_file.txt", "w");
/* Check if the file was successfully opened
    . */
if (fp == NULL)
{
    printf("Failed_to_open_file\n");
    return 1;
}

fprintf(fp, "%s\n", data);

fclose(fp);

...
```

Listing 11: GDB backtrace

```
(gdb) bt
...
#1  0xffffffff81249c6b in spin_lock (lock
    =<...>)
    at ...linux-6.1.7/include/linux/spinlock
        .h:350
#2  alloc_fd (start=start@entry=0, end=1024,
     flags=32768) at .../linux-6.1.7/fs/
    file.c:506
...
#6  0xffffffff81224cef in do_sys_open (dfd=<
    optimized out>, ...
...
#9  0xffffffff81e0009b in entry_SYSCALL_64
    () at ...
```

3. Process Scheduling: spin_lock is used to synchronize access to the scheduler queues. In the application code, I triggered dynamic memory allocation using "malloc," system call that subsequently caused a Page Fault and a process context switch. To trap the scenario, I set a breakpoint in the "pte_alloc_one" function and executed the code step-by-step in GDB until the spin_lock was used. In the __schedule function, it locked the queue to schedule another process if ready.

Listing 12: Relevant Application Code

```
...
int n = 1024;
int *array = malloc(n * sizeof(int));
if (array == NULL) {
    printf("Error: memory allocation failed
        .\n");
    return 1;
}

for (int i = 0; i < n; i++) {
    array[i] = i;
}

free(array);
...
```

Listing 13: GDB backtrace

```
(gdb) bt
...
#2  0xffffffff81ccd880 in raw_spin_rq_lock (
    rq=<...>)
    at .../kernel/sched/sched.h:1644
...
#5  0xffffffff81cce042 in
    preempt_schedule_common () at .../linux
    -6.1.7/kernel/sched/core.c:6721
...
#12 pte_alloc_one (mm=mm@entry=0
    xffff888003dd0800) at .../linux-6.1.7/
    arch/x86/mm/pgtable.c:33
...
#20 exc_page_fault (regs=0xffffc900001d7f58,
    error_code=6) at .../linux-6.1.7/arch/
    x86/mm/fault.c:1575
```

6 System Calls

In my kernel, the system calls trap into the kernel at the entry_SYSCALL_64() function. I set a conditional breakpoint in the function only when the process is executed based on the PID of the process.

When a system call is executed, the operating system switches from user to kernel mode, if Page Table Isolation is enabled the kernel uses a separate set of pagetables. As a result, the CR3 register will also change to point to the kernel-space page tables.

After GDB hits breakpoint at entry_SYSCALL_64 in the kernel, we can investigate the internals of the system calls. Firstly, we can get CR3 register – 0x3dde000 – using "info registers" command in GDB, as shown in Listings 6,7. Secondly, we can observe that the macro, "SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp", changes the CR3 registers.

In my QEMU/VM system, the CR3 register did not change when the Page Table Isolation (PTI) was enabled or disabled. One possible explanation for this is that in the host operating system, which has direct control of the CPU registers, the changes in the CR3 registers can be noticed in the system call context using GDB. However, in a guest Virtual Machine, since GDB uses the CPU registers state that the Operating System provides, so if the state is not being updated in the OS itself then GDB will not show the changes even when PTI is disabled.

However, I noticed that GDB's access to some symbols changes after CR3 is modified. As it can be observed from Listings 10, 11, before CR3 changes, accessing PID value through "$lx_curren().pid" throws a memory violation error. However, this vanishes after CR3 is modified. I speculate that the MMU updates the access permissions of the process when CR3 changes – which happens in hardware –, resultantly allowing the kernel to access the required data structures.

Listing 14: Before

```
(gdb) info registers
...
cr0            0x80050033          [ PG AM
    WP NE ET MP PE ]
cr2            0x556caa85f004
    93925205733380
cr3            0x3dde000           [ PDBR=1
    PCID=0 ]
...
```

Listing 15: After

```
(gdb) info registers
...
cr0            0x80050033          [ PG AM
    WP NE ET MP PE ]
cr2            0x556caa85f004
    93925205733380
cr3            0x3dde000           [ PDBR=1
    PCID=0 ]
...
```

With no Page Table Isolation:

Listing 16: PTI Before

```
(gdb) info registers
...
cr0            0x80050033          [ PG AM
    WP NE ET MP PE ]
cr2            0x55efad5b3004
    94487893979140
cr3            0x3dc8002           [ PDBR=1
    PCID=0 ]
...
```

Listing 17: PTI After

```
(gdb) info registers
...
cr0            0x80050033          [ PG AM
    WP NE ET MP PE ]
cr2            0x55efad5b3004
    94487893979140
cr3            0x3dc8002           [ PDBR=1
    PCID=0 ]
...
```

Listing 18: PID inaccessible with PTI

```
(gdb) info registers
...
Breakpoint 3, entry_SYSCALL_64 ()
    at ...linux-6.1.7/arch/x86/entry/
        entry_64.S:91
93              movq    %rsp, PER_CPU_VAR(
    cpu_tss_rw + TSS_sp2)
(gdb) p $lx_current().pid
Python Exception <class 'gdb.MemoryError'>
    Cannot access memory at address 0
    xffffffff82621860:
Error occurred in Python: Cannot access
    memory at address 0xffffffff82621860
(gdb) n
94              SWITCH_TO_KERNEL_CR3
    scratch_reg=%rsp
...
(gdb) n
entry_SYSCALL_64 () at ...linux-6.1.7/arch/
    x86/entry/entry_64.S:95
95              movq    PER_CPU_VAR(
    cpu_current_top_of_stack), %rsp
(gdb) p $lx_current().pid
$8 = 123
(gdb)
...
```

Listing 19: PID accessible with noPTI

```
(gdb) info registers
...
Breakpoint 2, entry_SYSCALL_64 ()
    at ...linux-6.1.7/arch/x86/entry/
        entry_64.S:91
91              swapgs
(gdb) p $lx_current().pid
$7 = 122
...
(gdb)
94              SWITCH_TO_KERNEL_CR3
    scratch_reg=%rsp
...
(gdb) p $lx_current().pid
$8 = 122
```

7 /dev/random vs /dev/urandom

In Linux, /dev/random and /dev/urandom are both pseudo-random number generators (PRNGs) that generate random numbers for cryptographic purposes. However, there are some key differences between the two. "/dev/random" is a PRNG that blocks (i.e. waits) when the kernel's pool of entropy (randomness) is exhausted. This means that the generator will not return any more random numbers until it has "gathered more entropy" from various sources such as keyboard timings, disk timings, etc. However, "/dev/urandom:" does not block when the kernel's pool of entropy is exhausted. Instead, it will continue to generate pseudo-random numbers using the existing entropy.

In our virtual machine, whether reading from /dev/random or /dev/urandom while debugging the kernel did not make a significant difference, as the amount of entropy available in the system was typically limited. In a virtual machine, the kernel may have less access to hardware events, such as keyboard and disk timings, that it uses to gather entropy. This means that the entropy pool may be smaller and the quality of the random numbers generated by /dev/urandom may be lower.

Interestingly, in contrary to what I expected – the application reading from /dev/random blocking because there is not enough entropy – both the applications did not block.

One possible explanation that I got convinced with is that in some cases the virtual machine may not have access to the physical hardware random number generator (HRNG) which is the source of true randomness, and as a result, the quality of the random numbers produced by /dev/random or /dev/urandom may be lower and they don't block

8 Time and Effort spent

I have spent more than 50 hours working on the lab. Not being able to boot the compiled kernel with an error saying missing "/sbin/init" process took a lot of time.