CS380L: Advanced Operating Systems Lab #0

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Introduction

In this writeup, we demonstrate the steps to compile and boot the Linux kernel on the KVM-qemu virtual machine. We time the OS bootup time using both timer and RTC, real-time clock in Linux and explain the value difference using the difference between system time and RTC. We also trace the kernel during the execution of a test program `testprog` and we explain the difference between `/dev/random` and `/dev/urandom`.

2 Environment

We use a machine that has 2 Intel(R) Core(TM)2 Duo CPU @ 3.00GHz processors and 8GB of memory. The machine runs Ubuntu 18.04.1 LTS (kernel version 4.15.0-43-generic).

KVM is enabled on the machine, and we use QEMU (version 2.11.1) [2] to create and run a VM for the lab. The VM runs a Ubuntu 18.04 LTS with the kernel we built (version 4.20.4).

3 Getting a VM running in KVM

We use the Ubuntu cloud image to setup the VM. The image for QEMU can be downloaded from `https://cloud-images.ubuntu.com/releases/18.04/release/ubuntu-18.04-server-cloudimg-amd64.img`. We use the cloud image instead of the regular desktop image to save space (e.g., We do not need to have GUI installed).

Ubuntu cloud image needs additional metadata to boot (mainly containing the login password). The metadata can be provided via a seed image [1]. To create a seed image, we first create a file `my-user-data` with contents:

```
#cloud-config
```

[1] 20 hours spent on this lab.

[2] `qemu-system-x86_64 --version`

```
password: passw0rd
chpasswd: { expire: False }
ssh_pwauth: True
```

Then we create the seed image by running:

```
sudo apt-get install cloud-utils
cloud-localds my-seed.img my-user-data
```

We then use the downloaded Ubuntu cloud image to create root disk image for the VM [3]:

```
qemu-img create -f qcow2 \
-b ubuntu-16.04-server-cloudimg-amd64-disk1.img \
my-disk.img 15G
```

Now, we are ready to boot up our VM:

```
qemu-system-x86_64 \
-enable-kvm -curses \
-m 512 -smp 4 -redir tcp:4444::22 \
-hda my-disk.img -hdb my-seed.img \
-cpu host
```

This will start a VM with 4 CPU cores and 512MB of memory. We redirect port 4444 of local machine to port 22 of the VM in order to login the VM via SSH. The VM will run in the terminal, and login with user name `ubuntu` and `passw0rd` set in `my-user-data`. The login screen of VM is shown in Figure 1. Once we have our VM boot up, we can remote access it via SSH from host `ssh -p 4444 ubuntu@localhost`.

4 Obtaining and building the kernel

We first obtain the Linux Kernel source via `wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.20.4.tar.xz`. Then, we extract the files using `tar -xJf linux-4.20.4.tar.xz`. We make a new directory `kbuild` as the build directory for kernel and

[3]The default virtual size is 2G, we can resize the image via `qemu-img resize my-disk.img +10G`. We add additional 10G in this case.

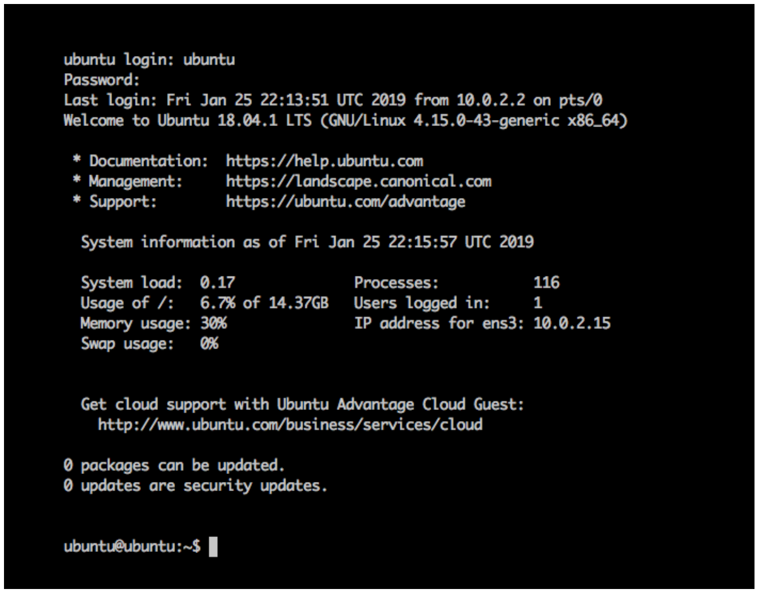

Figure 1: Login screen of our VM

`cd kbuild`, we generate `.config` file using `yes "" | make -C ../linux-4.20.4/ O=$( pwd)x86_64_defconfig`. Note that generating the `.config` file like this automatically set `CONFIG_SATA_AHCI=y`. We run `make -j4` [4]to build the kernel.

5 Installing and Copying Kernel Modules

We install the newly-built kernel by first making a new directory called `kinstall` as a sibling of `kbuild`. `kinstall` will contain the built kernel modules. Inside `kbuild`, we run `make INSTALL_MOD_PATH=../kinstall modules_install`.

We can see `lib` directory inside `kinstall`, which has to be copied to the root file system of the VM. We notice there are two symbolic links `build` and `source` inside `kinstall/lib/modules/4.20.4`, which links to the built kernel image and the source of the kernel. They are useless and may cause problems when copying files to the VM. Thus we just delete them. Next, we copy the entire 4.20.4 directory to `/lib/modules` in the guest system by doing `scp -P 4444 -r 4.20.4/ ubuntu@localhost:/home/ubuntu` and inside the guest sytem, do `sudo mv 4.20.4/ /lib/modules/`.

[4] `-j4` means 4 threads are used, which can speed up the build process

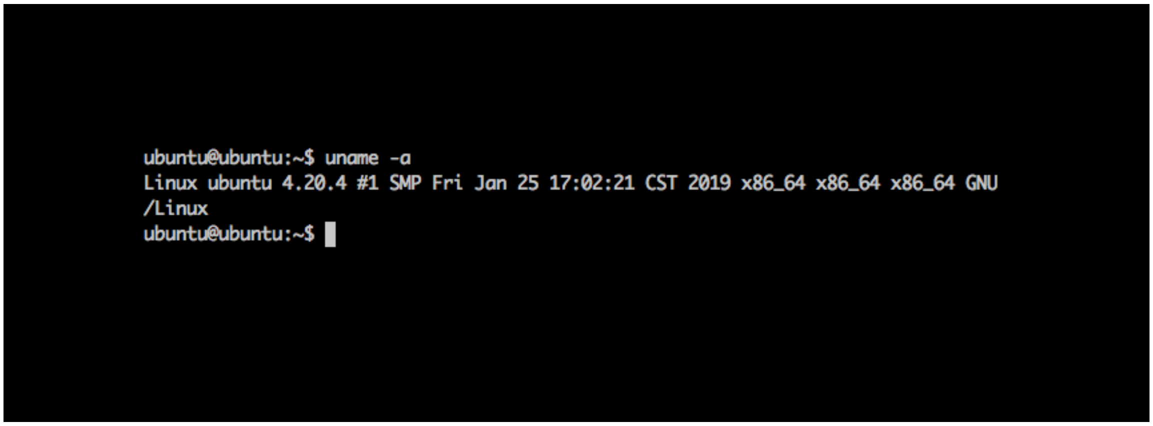

Figure 2: VM with our newly-built kernel

6 Booting KVM with your new Kernel

We can now start VM with our own Linux kernel. The shell command we run now:

```
qemu-system-x86_64 \
-enable-kvm -curses \
-m 512 -smp 4 -redir tcp:4444::22 \
-hda my-disk.img -hdb my-seed.img \
-kernel ~/380l-lab0/kbuild/arch/x86_64/boot/bzImage \
-append "root=/dev/sda1" \
-cpu host
```

Note that we append two new options `-kernel` and `-append` to `QEMU`. `-kernel` option tells the location of the kernel to use, and `-append` option suggests the parameters to start the kernel. The `root` parameter suggests the disk partition used as root file system. After login, use `uname -a` to check the kernel version string, which is shown in Figure 2.

7 Booting, kernel modules, and discovering devices

The wall clock time (tracked using a stopwatch) for our boot takes 34.08 seconds while the time reported by the Kernel takes 28.81 seconds. This difference may be due to the human delay on stopping the stopwatch and also due to a disagreement between human and OS on how to define boot finish status. Here, we stop our stopwatch when we see the login prompt but the last line of `dmesg` [5] shows:

[5] `dmesg` is used to inspect the kernel ring buffer, which contains the system log during kernel boot.

```
[ 28.811823] new mount options do not match the existing superblock, will
    be ignored
```

To eliminate the potential human error, we use real-time clock in Linux system to time the difference between the wall clock time and the time reported by Kernel.

```
$ dmesg -T | grep "RTC time"
[Fri Jan 25 23:54:33 2019] RTC time: 23:54:32, date: 01/25/19
```

`RTC` stands for "real-time clocks" [6]. We find that the time reported by Kernel is 1 second slower than the real-time clock at that moment. "RTC vs system clock" section in `man rtc` explains possible root cause for this 1 second difference: when the system is in a low power state, only RTC work not the system clock. The system clock is mantained by kernel implemented as counting of timer interrupts and the system clock will set to the wall clock time once the system boots and out of low power state. Thus, one possible explanation of the 1 second difference is due to the slower frequency of timer interrupts and another possible explanation is because the system clock has not aligned well with the wall clock time yet.

We also inspect the discovery of PCI devices at boot time from the boot log. We use the command `lspci` and there are 6 PCI devices in the VM:

```
$ 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)
```

We can search the boot log with the pattern of `0000:ID` (e.g., `0000:00:00.0`)from `lspci` to learn how the kernel discovers and identifies these devices during the boot process and the log message helps us to decide what kind of the device is.

[6] definition of RTC can be found via `man rtc`

```
$ dmesg | grep "0000:00:00.0"
[ 0.244811] pci 0000:00:00.0: [8086:1237] type 00 class 0x060000
[ 0.579080] pci 0000:00:00.0: Limiting direct PCI/PCI transfers

$ dmesg | grep "0000:00:01.0"
[ 0.245549] pci 0000:00:01.0: [8086:7000] type 00 class 0x060100
[ 0.578484] pci 0000:00:01.0: PIIX3: Enabling Passive Release
[ 0.586375] pci 0000:00:01.0: Activating ISA DMA hang workarounds

$ dmesg | grep "0000:00:01.1"
[ 0.246566] pci 0000:00:01.1: [8086:7010] type 00 class 0x010180
[ 0.250524] pci 0000:00:01.1: reg 0x20: [io 0xc040-0xc04f]
[ 0.252018] pci 0000:00:01.1: legacy IDE quirk: reg 0x10: [io 0x01f0-0x01f7
   ]
<-- snip -->

$ dmesg | grep "0000:00:01.3"
[ 0.256256] pci 0000:00:01.3: [8086:7113] type 00 class 0x068000
[ 0.257044] pci 0000:00:01.3: quirk: [io 0x0600-0x063f] claimed by PIIX4
   ACPI
[ 0.257208] pci 0000:00:01.3: quirk: [io 0x0700-0x070f] claimed by PIIX4
   SMB

$ dmesg | grep "0000:00:02.0"
[ 0.258317] pci 0000:00:02.0: [1234:1111] type 00 class 0x030000
[ 0.259810] pci 0000:00:02.0: reg 0x10: [mem 0xfd000000-0xfdffffff pref]
[ 0.262214] pci 0000:00:02.0: reg 0x18: [mem 0xfebb0000-0xfebb0fff]
<-- snip -->

$ dmesg | grep "0000:00:03.0"
[ 0.267327] pci 0000:00:03.0: [8086:100e] type 00 class 0x020000
[ 0.268194] pci 0000:00:03.0: reg 0x10: [mem 0xfeb80000-0xfeb9ffff]
```

```
[ 0.268973] pci 0000:00:03.0: reg 0x14: [io 0xc000-0xc03f]
<-- snip -->
```

8 Tracing the kernel

8.1 Make a debug build

To trace the kernel, we need to make a debug build of the kernel by modifying several debug options. Make a new directory `debug_bld2` for holding the debug build. In the created directory, run

```
make -C ../linux-4.15.9 O=$(pwd) x86_64_defconfig
make -C ../linux-4.15.9 O=$(pwd) kvmconfig
make -C ../linux-4.15.9 O=$(pwd) menuconfig
```

The last command will bring up a configuration menu and we change the options as follow [2]:

- Kernel hacking
  - Compile-time checks and compiler options
    - Compile the kernel with debug info (check this)
      - Generate dwarf4 debuginfo (check this)
      - Provide GDB scripts for kernel debugging (check this)
  - KGDB: kernel debugger (check this)
- General setup
  - Configure standard kernel features (expert users) (check this)
    - Configure standard kernel features (expert users) (check this)
- Processor type and features
  - Build a relocatable kernel (uncheck this)

We also want to explict set `CONFIG_DEBUG_INFO_REDUCED=n` explicitly in `.config` of `debug_bld2`. Then we compile the kernel `make -j16` and start the VM as

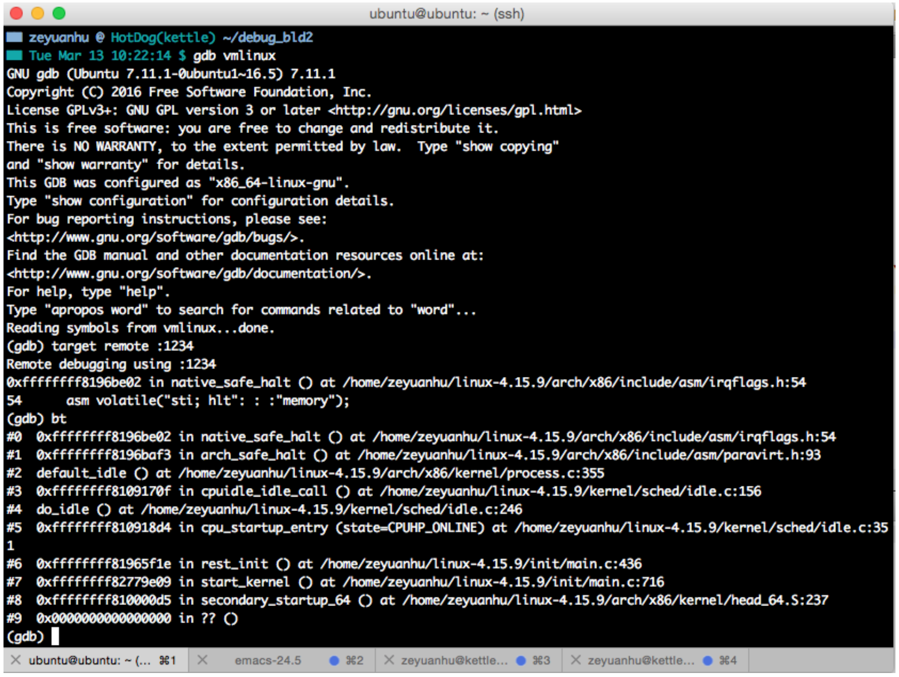

Figure 3: Fire up GDB and be ready to debug kernel

```
sudo qemu-system-x86_64 \
> -enable-kvm -nographic \
> -m 512 -smp 4 -redir tcp:4444::22 -s \
> -hda my-disk.img -hdb my-seed.img \
> -kernel ~/debug_bld2/arch/x86_64/boot/bzImage \
> -append "root=/dev/sda1"
```

Note that we add an option `-s`, which tells QEMU to start a GDB server on port 1234 for debugging [3] [7]. we can start GDB in `debug_bld2` directory via `gdb vmlinux`, and type `target remote :1234` to connect gdb to the kgdb server in the guest system. Figure 3 shows a screenshot of the GDB that is ready to debug the kernel.

Next, we create a program `testprog.c` on the guest system like the following [8]:

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

[7]We also use `-nographic` instead of `-curses` because we find out that typing `./testprog` can be quite sluggish on the guest system (due to the constant checking of the breakpoint) and using `-nographic` instead of `-curses` to boot up the VM and login the VM via SSH helps to alleviate this effect.

[8]We modify the program by appending extra line `while (1){}`. Doing so make sure that the breakpoint will be hit evetually when the program is being executed (since the program is non-terminal). Since the program is fairly short and the execution is very quick. If we do not add this line, sometimes the program will finish execution without the breakpoint getting hit and that hurts reproducibility

```c
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);
    while (1) {}
}
```

Compile it with gcc: `gcc -o testprog -g testprog.c`. Now, we want to trace into the kernel when the process contains `testprog` is running [9]. To do so, we set a conditional breakpoint in `spin_lock` in kernel code that will only stop execution if the above process is running. `spin_lock` is an inline Macro and the actual symbol name is `__raw_spin_lock`, which is defined in `include/linux/spinlock_api_smp.h`. To ensure the breakpoint only be triggered during the execution of `testprog`, we have to add a condition to the breakpoint. We use the helper script provided by kernel to figure out the PID of `testprog`. We achieve so via `$lx_current()`, which reads `task_struct` of current task in GDB and `task_struct` contains all the information we need to identify the current proces. Specifically, `$lx_current().pid` gives the PID of the current running process and `$lx_current().comm` gives the command line content, which we will use it to identify the process.

The command we run is following

```
b __raw_spin_lock if $_streq($lx_current().comm, "testprog")
```

Figure 4 shows the result of `testprog` hits the breakpoint for the first time. From the figure we can see that `$lx current().pid` gives 2442 and `$lx current().comm` gives `"testprog\000\000\000\000\000\000\000"`, which confirm that we are in `testprog` process when we hit `spin_lock` breakpoint. Then, we use `bt` to examine the call stack.

In the first time the breakpoint is triggered, the stack looks like:

```
#0 _raw_spin_lock (lock=0xffff88001fc1bbc0) at /home/zeyuanhu/linux-4.15.9/
    kernel/locking/spinlock.c:144
```

[9] We first run `target remote :1234` and then we setup the breakpoint. Afterward, we issue `continue` in the GDB so that we can run `testprog` on the guest system.

```
Type "apropos word" to search for commands related to "word"...
The target architecture is assumed to be i386:x86-64:intel
Reading symbols from vmlinux...done.
(gdb) target remote :1234
Remote debugging using :1234
0xffffffff8196be02 in native_safe_halt () at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/irqflags.h:54
54      asm volatile("sti; hlt": : :"memory");
(gdb) b __raw_spin_lock if $_streq($lx_current().comm, "./testprog\n")
Breakpoint 1 at 0xffffffff8196c210: file /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h, line 187.
(gdb) b __raw_spin_lock if $_streq($lx_current().comm, "./testprog")
Note: breakpoint 1 also set at pc 0xffffffff8196c210.
Breakpoint 2 at 0xffffffff8196c210: file /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h, line 187.
(gdb) b __raw_spin_lock if $_streq($lx_current().comm, "testprog")
Note: breakpoints 1 and 2 also set at pc 0xffffffff8196c210.
Breakpoint 3 at 0xffffffff8196c210: file /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h, line 187.
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "./testprog\n")
2       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "./testprog")
3       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "testprog")
(gdb) c
Continuing.

Thread 1 hit Breakpoint 3, _raw_spin_lock (lock=0xffff88001fc1bbc0) at /home/zeyuanhu/linux-4.15.9/kernel/locking/spinlock.c:144
144     __raw_spin_lock(lock);
(gdb) p $lx_current().comm
$1 = "testprog\000\000\000\000\000\000\000"
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "./testprog\n")
2       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "./testprog")
3       breakpoint     keep y   0xffffffff8196c210 in _raw_spin_lock at /home/zeyuanhu/linux-4.15.9/arch/x86/include/asm/atomic.h:187
  stop only if $_streq($lx_current().comm, "testprog")
  breakpoint already hit 1 time
(gdb) p $lx_current().pid
$2 = 2442
(gdb)
```

Figure 4: The first time that `testprog` hits breakpoint

```
#1 0xffffffff810bd8d0 in hrtimer_interrupt (dev=<optimized out>) at /home/
    zeyuanhu/linux-4.15.9/kernel/time/hrtimer.c:1303
...
```

From the tace, we see the kernel is running handler for the timer interrupt.If we take a look at function `hrtimer_interrupt` in `kernel/time/hrtimer.c`, we know the `hrtimer_bases`, a per-CPU variable [4], acquired a lock [10].

We continue the kernel tracing and the stack looks like below when we hit the breakpoint for the second time:

```
#0 _raw_spin_lock (lock=0xffffffff82206a04 <jiffies_lock+4>) at /home/
    zeyuanhu/linux-4.15.9/kernel/locking/spinlock.c:144
...
#4 0xffffffff810cb57f in tick_sched_timer (timer=0xffff88001fc1bfe0) at /
    home/zeyuanhu/linux-4.15.9/kernel/time/tick-sched.c:1187
...
#9 smp_apic_timer_interrupt (regs=<optimized out>) at /home/zeyuanhu/linux
    -4.15.9/arch/x86/kernel/apic/apic.c:1050
```

It is still inside the handler for timer interrupt, and `jiffies_lock` is acquired, which is a global variable. Function `tick_do_update_jiffies64` updates current jiffies.

The breakpoint is hit in a different context happens inside function `update_process_times`, still during handler for timer interrupt:

```
(gdb) bt
#0 _raw_spin_lock (lock=0xffff88001fc207c0) at /home/zeyuanhu/linux-4.15.9/
    kernel/locking/spinlock.c:144
...
#3 0xffffffff810bc9ab in update_process_times (user_tick=0) at /home/
    zeyuanhu/linux-4.15.9/kernel/time/timer.c:1633
...
#8 0xffffffff810bd90d in hrtimer_interrupt (dev=<optimized out>) at /home/
    zeyuanhu/linux-4.15.9/kernel/time/hrtimer.c:1316
```

[10] In GDB, the helper script also provides a function `$lx_per_cpu` to obtain per-CPU variables (actually `$lx_current()` is a shorthand to `$lx_per_cpu("current task")`)

Here the lock for per-CPU variable `runqueues` is acquired.

Continuing trace will let us see something out of timer interrupt. One example is breakpoint hit during the page fault:

```
0 _raw_spin_lock (lock=0xffff88001cc1ec6c) at /home/zeyuanhu/linux-4.15.9/
    kernel/locking/spinlock.c:144
...
#3 0xffffffff81167316 in pud_alloc (address=<optimized out>, p4d=<optimized
     out>, mm=<optimized out>) at /home/zeyuanhu/linux-4.15.9/include/linux
    /mm.h:1733
#4 __handle_mm_fault (vma=<optimized out>, address=6295640, flags=<
    optimized out>) at /home/zeyuanhu/linux-4.15.9/mm/memory.c:4008
#5 0xffffffff811678ad in handle_mm_fault (vma=<optimized out>, address=<
    optimized out>, flags=<optimized out>) at /home/zeyuanhu/linux-4.15.9/
    mm/memory.c:4104
#6 0xffffffff8104bede in __do_page_fault (regs=0xffffc90000317ce8,
    error_code=2, address=6295640) at /home/zeyuanhu/linux-4.15.9/arch/x86/
    mm/fault.c:1426
#7 0xffffffff81a0168b in async_page_fault () at /home/zeyuanhu/linux
    -4.15.9/arch/x86/entry/entry_64.S:1118
```

Another one is the scheduler wakes up process and queues the process:

```
...
#2 ttwu_queue (wake_flags=<optimized out>, cpu=<optimized out>, p=<
    optimized out>) at /home/zeyuanhu/linux-4.15.9/kernel/sched/core.c:1863
#3 try_to_wake_up (p=0xffff88001cf9a4c0, state=<optimized out>, wake_flags
    =0) at /home/zeyuanhu/linux-4.15.9/kernel/sched/core.c:2078
#4 0xffffffff8107cebc in wake_up_process (p=<optimized out>) at /home/
    zeyuanhu/linux-4.15.9/kernel/sched/core.c:2151
#5 0xffffffff8106d0e3 in wake_up_worker (pool=<optimized out>) at /home/
    zeyuanhu/linux-4.15.9/kernel/workqueue.c:840
#6 insert_work (pwq=<optimized out>, work=<optimized out>, head=<optimized
    out>, extra_flags=<optimized out>) at /home/zeyuanhu/linux-4.15.9/
```

```
    kernel/workqueue.c:1313
#7 0xffffffff8106d212 in __queue_work (cpu=<optimized out>, wq=0x0 <
    irq_stack_union>, work=0xffff88001fc00000) at /home/zeyuanhu/linux
    -4.15.9/kernel/workqueue.c:1463
#8 0xffffffff810bad36 in call_timer_fn (timer=0xffff88001fc207c0, fn=0x0 <
    irq_stack_union>) at /home/zeyuanhu/linux-4.15.9/kernel/time/timer.c
    :1318
#9 0xffffffff810bb209 in expire_timers (head=<optimized out>, base=<
    optimized out>) at /home/zeyuanhu/linux-4.15.9/kernel/time/timer.c:1351
#10 __run_timers (base=<optimized out>) at /home/zeyuanhu/linux-4.15.9/
    kernel/time/timer.c:1658
...
```

9 Differences between /dev/random and /dev/urandom

Both `/dev/random` and `/dev/urandom` are interfaces to the kernel's random number generator [5] and both of them are fed by the same cryptographically secure pseudorandom number generator [6]. However, they are different on how they handle their repective entropy pool when the pool is empty. `/dev/random` will block the reads if its entropy pool is empty and the reads will be blocked until additional environmental noise is gathered. However, `/dev/urandom` will not block waiting for more entropy and as a result, the returned values may have theoretical vulunerability. There is an argument on when to use which and some suggests that use `/dev/urandom` is strictly better as the thoeretical vulunerability may not lead to computational vulunerability [6] and thus should be used all the time. But, `man` page seems to suggest that it is a case-by-case situation [5].

References

[1] S. Moser, "Using ubuntu cloud-images without a cloud." http://ubuntu-smoser.blogspot.com/2013/02/using-ubuntu-cloud-images-without-cloud.html, 2011.

[2] "Cs380l: Advanced operating systems lab 0." https://www.cs.utexas.edu/~rossbach/380L/lab/lab0.html#debug-config, 2018.

[3] "Debugging kernel and modules via gdb." https://01.org/linuxgraphics/gfx-docs/drm/dev-tools/gdb-kernel-debugging.html, 2018.

[4] "A brief introduction to per-cpu variables." http://thinkiii.blogspot.com/2014/05/a-brief-introduction-to-per-cpu.html, 2014.

[5] "urandom(4) - linux man page." https://linux.die.net/man/4/urandom.

[6] "Myths about /dev/urandom." https://www.2uo.de/myths-about-urandom/.