EECS 750 Mini Project #1

MemGuard on Raspberry Pi 3

In this mini-project, you will first learn how to build your own kernel on raspberry pi3. You then will learn to compile and use an out-of-source-tree kernel module (MemGuard). You should return **mini-proj1-1.txt, mini-proj1-2.txt, mini-proj1-3.txt, mini-proj1-4.txt.**

Table of Content **1**

Part 0. Install Raspbian

First thing first. Install the Raspbian OS on the Pi 3. A 32GB micro-SD card should be included in the DonkeyCar system box you received. Follow the detailed instruction in the following. Using NOOB may be easiest, but directly installing Raspbian is also not that hard.
https://www.raspberrypi.org/downloads/

Part 1. Build your own kernel

Once you boot to the Pi 3, it's time to install your own kernel. All the necessary information can be found in the following document.
https://www.raspberrypi.org/documentation/linux/kernel/building.md

Below is essentially a copy of the document above, with some additional commentary.

```
$ sudo apt-get install git bc
$ git clone --depth=1 https://github.com/raspberrypi/linux
$ cd linux
$ KERNEL=kernel7
$ make bcm2709_defconfig
```

Before you begin, let's backup the current kernel just in case.

```
$ sudo cp /boot/kernel7.img /boot/kernel7.img.backup
```

Then, do the following to build the kernel.

```
$ make -j4 zImage modules dtbs
```

Now, this will take about 1.5 hour. So, take a break, sleep, or watch a movie while the pi is busy compiling the kernel. After a long time, if the compilation was successful, proceed the following to install the compiled kernel (and other stuff).

```
$ sudo make modules_install
$ sudo cp arch/arm/boot/dts/*.dtb /boot/
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
$ sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
$ sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

If everything went smoothly, then reboot the system and check if the kernel is the one you just compiled.

```
$ uname -a
Linux raspberrypi 4.9.80-v7+ #1 SMP Mon Feb 12 23:07:22 UTC 2018 armv7l
GNU/Linux
```

Copy the output of ‘uname -a’ command on your Pi 3 and save it as ‘mini-proj1-1.txt’ file. You should return the file as a proof.

Part 2. Worst-case memory interference experiment

Install the IsolBench benchmark suite (as in HW#3) on the Pi 3.

```
$ git clone https://github.com/CSL-KU/IsolBench
$ cd IsolBench/bench
$ make
```

You will use the ‘latency’ and ‘bandwidth’ benchmarks in the rest of the project. So, let’s install them.

```
$ sudo cp latency bandwidth /usr/local/bin/
```

Now, we will conduct an experiment to see worst-case memory interference on the Pi 3. The basic setup is that you first measure the performance of the ‘latency’ benchmark first alone in

isolation (Solo experiment) and then with three instances of the ‘bandwidth’ benchmark (Corun experiment).

First, let’s measure the ‘Solo’ baseline as follows. Here, the ‘-c’ option specify the core.

```
$ latency -c 0
...
average 184.95 ns | bandwidth 346.03 MB (330.00 MiB)/s
```

Next, launch three instances of the ‘bandwidth’ benchmark on core 1, 2, and 3, followed by the ‘latency’ benchmark on core 0 as follows.

```
$ for c in 1 2 3; do bandwidth -c $c -t 1000 & done
$ latency -c 0
...
average 563.22 ns | bandwidth 113.63 MB (108.37 MiB)/s
readsum  214745088000
```

You can see the performance of the ‘latency’ benchmark is significantly worse than the solo performance.

Next, repeat the Corun experiment, but with using ‘-a write’ option when launching the bandwidth benchmark, which changes its memory access pattern from read to write.

```
$ killall -9 bandwidth
$ for c in 1 2 3; do bandwidth -a write -c $c -t 1000 & done
$ latency -c 0
...
average 1124.04 ns | bandwidth 56.94 MB (54.30 MiB)/s
readsum  214745088000
```

You can see the performance of the latency benchmark is dropped even further. Compared to its original Solo performance of 346 MB/s, it is now only able to achieve a pitiful 57 MB/s (just about 16% of the its original speed.) This is because the three co-scheduled bandwidth benchmark instances cause significant memory interference.

Save the outputs of the experiments above as ‘mini-proj1-2.txt’ file. You should return the file as a proof.

Part 3. Build MemGuard kernel module

Now, let’s use MemGuard to mitigate the situation we observed in Part 2. First, build the memguard kernel module as follows.

```
$ git clone https://github.com/heechul/memguard
$ cd memguard
$ make
```

If the build was successful, do the following to load the MemGuard kernel module. Here, “g_hw_type=armv7” notifies the module to use L2 data cache refill performance counter.

```
$ sudo bash
# insmod ./memguard.ko g_hw_type=armv7
```

If the load was successful (Note that the currently running kernel’s version must match with the version of the kernel source code you used to build the module. If you completed Part 1, they should match,) do the following.

```
# pushd /sys/kernel/debug/memguard/
# ls
control limit usage
# cat limit
cpu  |budget (MB/s,pct,weight)
------------------------------
CPU0: 16384 (1000MB/s)
CPU1: 16384 (1000MB/s)
CPU2: 16384 (1000MB/s)
CPU3: 16384 (1000MB/s)
```

The ‘limit’ file shows the current budget of each core. Change the budgets as follows.

```
# echo mb 1000 100 100 100 > limit
# cat limit
cpu  |budget (MB/s,pct,weight)
------------------------------
CPU0: 16384 (1000MB/s)
CPU1: 1638 (100MB/s)
CPU2: 1638 (100MB/s)
CPU3: 1638 (100MB/s)
```

As you can see, the budgets of core1,2 and 3 are now adjusted to 100MB/s (1638 LLC misses per 1ms), while the core0’s budget is 1GB/s (16384 LLC misses per 1ms).

Now, let’s get back to the user-mode as follows.

```
# suspend
```

Next, check if each core's budget is indeed enforced by MemGuard. Launch the bandwidth benchmark on Core 3 and 0, respectively, as follows.

```
$ bandwidth -c 3
...
CPU3: B/W = 99.99 MB/s | CPU3: average = 610.43 ns

$ bandwidth -c 0
...
CPU0: B/W = 999.16 MB/s | CPU0: average = 61.09 ns
```

Save the outputs of the experiments above as 'mini-proj1-3.txt' file. You should return the file as a proof.

Part 4. Worst-case memory interference experiment w/ MemGuard

You now repeat the Part 2. Because MemGuard is running, core 1, 2, and 3 are regulated at 100 MB/s at best.

Solo baseline (incl. Memguard overhead)\

```
$ latency -c 0
...
average 185.01 ns | bandwidth 345.92 MB (329.89 MiB)/s
```

Read interference

```
$ for c in 1 2 3; do bandwidth -c $c -t 1000 & done
$ latency -c 0
...
average 209.59 ns | bandwidth 305.36 MB (291.21 MiB)/s
```

Write interference

```
$ for c in 1 2 3; do bandwidth -a write -c $c -t 1000 & done
$ latency -c 0
...
average 233.36 ns | bandwidth 274.25 MB (261.54 MiB)/s
```

Save the outputs of the experiments above as 'mini-proj1-4.txt' file. You should return the file as a proof.