ECE566 Enterprise Storage Architecture Program: Intro to BUSE and RAID

In the last Program, you learned FUSE, which allowed you to create a filesystem written in userspace code. In this program, you'll be using an analogous library, BUSE ("Block device in USErspace"), developed by Adam Cozzette.

Directions:

- This assignment will be completed individually.
- Computing environments:
    - During development (Parts 1-5), you can use a VCM virtual machine for all of the BUSE driver work.
    - During performance analysis (Part 6), the assignment asks you to use your **group server**, even though this assignment is individual. Group members should coordinate and share this resource and avoid inspecting each other's code as they do so.
- Format:
    - **Software deliverables:** You'll be asked to write some programs. Submit a tarball (`<netid>-buse.tgz`) to the Canvas locker for this assignment that includes all the requested programs along with the full BUSE build/run environment.
    - **Written deliverables:** Part of the assignment will ask for performance measurements. Compile your data and analysis into a report to be submitted as `<netid>-buse-report.pdf` to Canvas.

1 Introducing BUSE

BUSE is very similar to FUSE, except it allows you to write block devices rather than file systems. You should familiarize yourself with BUSE by reviewing the README and example source code here:

https://github.com/acozzette/BUSE

When it comes to actually downloading the code, instead of that github, download this package, which includes the content of the BUSE repo plus additional instructor-provided materials:

https://people.duke.edu/~tkb13/courses/ece566-2025sp/homework/BUSE_with_RAID1.tgz

2 A basic RAID1 driver

This assignment will have you writing drivers for RAID 0 and RAID 4. To prepare you for this, a RAID1 driver has been prepared for you to use as a base.

Download and extract the provided package, `BUSE_with_RAID1.tgz`, linked above.

The provided RAID1 driver is a minimal implementation. It supports the core RAID1 algorithm (mirroring), plus interleaved reads using both drives, a basic degraded mode (in which a drive is missing), and the ability to rebuild the RAID offline at startup (basically copy the good drive over to the replacement drive).

Before using this code, you'll need to ensure you have C build tools like make and gcc (`sudo apt install build-essential`). To use it, you'll need to ensure the `nbd` kernel module[1] is loaded (`modprobe nbd`). You can check if this module loaded with `lsmod`.

You can build the library and the example programs, including `raid1`, by just typing `make`.

Running `raid1` with the `--help` argument will show its usage:

```
$ ./raid1 --help
Usage: raid1 [OPTION...] BLOCKSIZE RAIDDEVICE DEVICE1 DEVICE2
BUSE implementation of RAID1 for two devices.
`BLOCKSIZE` is an integer number of bytes.

`RAIDDEVICE` is a path to an NBD block device, for example "/dev/nbd0".

`DEVICE*` is a path to underlying block devices. Normal files can be used too.
A `DEVICE` may be specified as "MISSING" to run in degraded mode.

If you prepend '+' to a DEVICE, you are re-adding it as a replacement to the
RAID, and we will rebuild the array. This is synchronous; the rebuild will have
to finish before the RAID is started.

  -v, --verbose              Produce verbose output
  -?, --help                 Give this help list
      --usage                Give a short usage message
```

As you can see, it accepts a block size (in bytes), the resulting RAID device, and two underlying block devices. Because of how BUSE is implemented, the RAID device will always be `/dev/nbd#`, where `#` is a small integer of your choosing (such as 0).

Below is an annotated series of exercises designed to test `raid1` and show how it works. Note that the `raid1` program is always run synchronously; it does not background itself the way FUSE programs do by default. Therefore, running this test will involve multiple terminal sessions (one for BUSE, one for using the resulting block device).

[1] This network block device (nbd) module was originally intended for SAN-like protocols in Linux; it's used by the author of BUSE to provide easy access to user-programmable block devices without kernel code.

2.1 Creating block device files

As with the FUSE `twofs` program, the underlying device can either be an actual block device (`/dev/sdb`) or plain files of your own creation. For testing, we'll be making our own underlying devices as plain files. A bit of shell scripting and `dd` makes quick work of it:

```
$ for D in img0 img1 ; do dd if=/dev/zero of=$D bs=3M count=1 ; done
```

2.2 Running in normal mode

Let's assume a block size of 1024 bytes[2] and the block device images created above. We can instantiate a RAID1 as `/dev/nbd0` (verbose mode enabled) with the command:

```
$ sudo ./raid1 -v 1024 /dev/nbd0 img0 img1
Got device 'img0', size 3145728 bytes.
Got device 'img1', size 3145728 bytes.
RAID device resulting size: 3145728.
R - 0, 1024
R - 1024, 1024
R - 2048, 1024
R - 3072, 1024
 …
```

The reads ("R" lines) you see immediately are probes by the OS looking for the partition table of the block device (in this case, there is none). We'll continue this program running in one terminal and switch to another.

We can do a single small write directly and confirm it appears on the underlying image files. We'll become root and do the following:

```
$ sudo -i
[sudo] password:
root# echo hi > /dev/nbd0
root# sync
```

[2] For this RAID1 implementation, the block size hardly matters, as we just pass the IO through to the underlying devices. It's just used to round our RAID device's total size to the nearest lesser multiple of the block size. For your RAID implementations, block size will matter much more, which is why this example code accepts this parameter.

The sync command will ask the OS to flush any buffers it has. Upon the sync, we'll see that our raid1 program has reported a write at offset 0 ("`W - 0, 1024`"). We can confirm the change propagated to both image files by looking at hex dump of each:

```
$ hd img0
00000000  68 69 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |hi..............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00300000
$ hd img1
00000000  68 69 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |hi..............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00300000
```

We can go further and create a whole filesystem on the block device, mount it, write files, read files, and unmount the filesystem. Each step will create corresponding read/write requests on the `raid1` terminal.

```
root:~# mkfs.ext4 /dev/nbd0
mke2fs 1.42.13 (17-May-2015)
Discarding device blocks: done
Creating filesystem with 3072 1k blocks and 768 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

root:~# mkdir mountpoint
root:~# mount /dev/nbd0 mountpoint/
root:~# cd mountpoint/
root:~/mountpoint# echo wow > file1
root:~/mountpoint# echo hey > file2
root:~/mountpoint# echo secondline >>  file2
root:~/mountpoint# cat file1
wow
root:~/mountpoint# cat file2
hey
secondline
root:~/mountpoint# df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/nbd0       2.0M   27K  1.7M   2% /root/mountpoint
root:~/mountpoint# cd ..
root:~# umount mountpoint/
```

We can now end the RAID by pressing Ctrl+C on the `raid1` terminal.

2.3 Failing a drive

This RAID1 driver doesn't detect drive failures at runtime, but we can specify that a drive is failed ("MISSING") when launching `raid1`. We do so by specifying the word "`MISSING`" instead of one of the block device paths. This will cause the driver to run in degraded mode, which, for a RAID1, just means that all IOs map to the one remaining drive.

On the terminal we'll use for raid1, let's remove one of the images and run in degraded mode:

```
$ rm img0
$ sudo ./raid1 -v 1024 /dev/nbd0 MISSING img1
DEGRADED: Device number 0 is missing!
Got device 'img1', size 3145728 bytes.
RAID device resulting size: 3145728.
…
```

At this time, you can mount the block device and confirm the files are still present:

```
root:~# mount /dev/nbd0 mountpoint/
root:~# cd mountpoint/
root:~/mountpoint# ls
file1  file2  lost+found
root:~/mountpoint# echo a third file > file3
root:~/mountpoint# ls
file1  file2  file3  lost+found
root:~/mountpoint# cd ..
root:~# umount mountpoint/
```

2.4 Rebuilding the RAID

The provided RAID driver supports rebuilding the RAID, but in a simple way – it does so "offline" at start-up before making the block device available. For RAID1, this is as simple as copying all blocks from the surviving disk to a newly provided replacement disk. The replacement disk is specified on the command line by prefixing it with a '+'.

Below, we create a new replacement disk image, then launch `raid1` in rebuild mode:

```
$ dd if=/dev/zero of=img0 bs=3M count=1
1+0 records in
1+0 records out
3145728 bytes (3.1 MB, 3.0 MiB) copied, 0.0860238 s, 36.6 MB/s
$ sudo ./raid1 -v 1024 /dev/nbd0 +img0 img1
Got device 'img0', size 3145728 bytes.
Got device 'img1', size 3145728 bytes.
Doing RAID rebuild...
RAID device resulting size: 3145728.
…
```

We can again confirm that our filesystem is intact:

```
root:~# mount /dev/nbd0 mountpoint/
root:~# cd mountpoint/
root:~/mountpoint# ls
file1  file2  file3  lost+found
root:~/mountpoint# cat file1
wow
root:~/mountpoint# cat file3
a third file
root:~/mountpoint# echo even a fourth file > file4
root:~/mountpoint# rm file2
root:~/mountpoint# ls
file1  file3  file4  lost+found
root:~/mountpoint# cd ..
root:~# umount mountpoint/
```

3 Task: RAID0 driver (25 points)

Using `raid1` as a base, create a `raid0` driver. The driver need only support two-disk RAID0 setups, but should feature configurable block size (as `raid1` does). As a reminder, if you have 2 drives (numbered 0 and 1) and a block size of BS, then:

- An IO to offset L of size S will be from block number L/BS to block number (L+S)/BS.
- An IO of block number B should go to drive B%2, block B/2

Because RAID0 offers no redundancy, this driver should not have any concept of "`MISSING`" or added ('+') drives.

Conduct thorough testing on your driver for correctness (because we will do so in grading it).

4 Task: RAID4 driver (25 points)

Using `raid1` as a base, create a `raid4` driver. The driver should support an arbitrary number of drives between (at least 3 by definition with a maximum limit of 16) and should feature configurable block size (as `raid1` does).

Because RAID4 does provide redundancy, this driver should support the "`MISSING`" and added ('+') drive concepts like `raid1` does.

For writes, be sure to implement the fast small write algorithm we covered in class.

Conduct thorough testing on your driver for correctness (because we will do so in grading it).

5 Task: Your choice of driver (30 points)

You have your choice of which **one** of the following three options you do for the part of this assignment:

5.1 RAID 5

Implement a RAID-5 driver that functions in a manner similar to the RAID 4 driver.

5.2 RAID 10

Implement a single RAID-10 driver that functions in a manner similar to the other drivers (i.e., it does execute the existing RAID1 and RAID0 drivers). You can assume that the number of devices at the RAID-1 layer is always 2, and therefore an even number of devices greater than or equal to 4 is required.

5.3 Rudimentary SAN (extra credit)

Now that you have a way to write a block device driver in userspace, use it to put together a rudimentary SAN protocol. This is easier than it may sound: all you have to put over the network is "read block" and "write block" (and perhaps `flush`).

To do this, you'll write an initiator using BUSE and a target using plain C (or actually any language you wish), and have the two communicate via a socket. You will design the network protocol yourself. Note that no RAID logic is needed within this SAN driver.

Note: because this option involves two programs that are dissimilar to the RAID drivers, **you** must provide adequate documentation for us to compile and use your SAN initiator and target programs.

If you choose this option, demo it to the instructor for up to 10 points extra credit on top of the 20 points this task is worth. Reach out if you have questions. Have fun!

6 Task: Performance analysis (20 points)

Once you have your drivers working, use the `iops` benchmark tool from Lab 1 to assess both the **sequential** and **random** performance of your drivers. Specifically, you should test the following disk configurations:

- Your RAID0 driver run against your first two hard disk drives (likely `/dev/sdb` & `/dev/sdc`).
- Our RAID1 driver run against your first two hard disk drives (likely `/dev/sdb` & `/dev/sdc`).
- Your RAID4 driver run against your four hard disk drives (likely `/dev/sdb` … `/dev/sde`).
- Your RAID4 driver in degraded mode, with the *second* provided drive missing (likely `/dev/sdb`, `MISSING`, `/dev/sdd`, and `/dev/sde`).
- Your driver from Part 5 – one of:
    - RAID 10 run against all four hard disk drives
    - RAID 5 run against all four hard disk drives
    - Rudimentary SAN where the initiator is a VCM virtual machine and the target is your group server exporting a single drive directly (likely `/dev/sdb`)
- A single drive directly, by itself (likely `/dev/sdb`)

The setup for the benchmark should be 15 seconds runtime, and 4096 block size; 1 thread for sequential and 8 threads for random.

The RAID drivers should also use a block size of 4096. Be sure to disable verbose mode (`-v`) in `iops` when benchmarking. All those prints can have a big performance effect. Further, between each test, have the OS drop its block device cache to ensure a fair test. To do this, see directions here.

Produce a report showing your results as **bar graphs** for random IO (IOPS) and sequential IO (MB/s) that compare the above 5 configurations. Do multiple repetitions and include error bars to show standard deviation. Also, in an appendix, include the raw results in a **well-formatted table**.

Write a **brief analysis** of the above, describing how the trends observed match (or disagree with!) the theory discussed in class. Note that BUSE has appreciable overhead and your implementation likely lacks many of the possible performance optimizations, so you may not see the exact improvement you're expecting.