Description

In this homework we will be implementing mirroring, striping, and RAID in a block device layer. The basic abstraction you will use is the following structure:

```
struct blkdev {
    struct blkdev_ops *ops;
    void *private;
};

#define BLOCK_SIZE 512   /* 512-byte unit for all blkdev addresses */

struct blkdev_ops {
    int (*size)(struct blkdev *dev);
    int (*read)(struct blkdev * dev, int LBA, int len, void *buf);
    int (*write(struct blkdev * dev, int LBA, int len, void *buf);
    void (*close)(struct blkdev *dev);
};
```

This is a common style of operating system structure, which provides the equivalent of a C++ abstract class by using a structure of function pointers for the virtual method table and a void* pointer for any subclass-specific data. Interfaces like this are used so that independently compiled drivers (e.g. network and graphics drivers) to be loaded into the kernel in an OS such as Windows or Linux and then invoked by direct function calls from within the OS.

The methods provided in the blkdev_ops structure are:

- `size` - the total size of this block device, in 512-byte sectors
- `read` - read '`len`' sectors into a buffer. The caller guarantees that '`buf`' points to a buffer large enough to hold the amount of data being requested, and that `len>0`. Legal return values are `SUCCESS`, `E_BADADDR`, and `E_UNAVAIL`. (defined in `blkdev.h`)
- `write` - write '`len`' sectors. The caller guarantees that '`buf`' points to a buffer holding the amount of data being written, and that `len>0`. Legal return values are `SUCCESS`, `E_BADADDR`, and `E_UNAVAIL`.
- `close` - the destructor method, this closes all blkdevs "underneath" this one and frees any memory allocated. Note that once this method is called, the blkdev object must not be accessed (not even to call `size`)

The `E_BADADDR` error is returned if any address in the requested range is illegal - i.e. less than zero or greater than `blkdev->ops->size(blkdev)-1.`

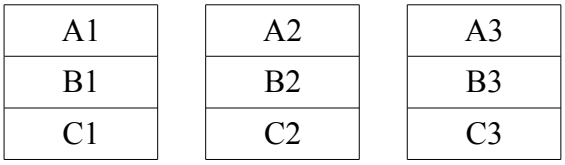
The `E_UNAVAIL` error is returned if a device fails. If your code receives this error, it should call `close()` on the corresponding blkdev; after this it cannot access the device anymore. Your code should only return this error if it is unable to read (or write) the requested data due to a single disk failure (for striping) or multiple disk failures (for mirroring and RAID 4)

We will be working with disk image files, rather than actual devices, for ease of running and debugging

your code. You may be familiar with image files in the form of .ISO files, which are byte-for-byte copies of a CD-ROM or DVD, and can be read by the same file system code which interacts with a physical disk; in our case we will be writing to the files as well as reading them.

You will be writing what are termed "filter drivers", which sit between the actual disk driver (or in our case, the image file access routines) and the file system above, so your code will read and write via a `struct blkdev`, and will export a `struct blkdev` to higher layers.

First, some terminology:



If we have 3 disks - *1 (A1, B1, C1,...), *2, and *3 in a striped or RAID configuration, then we call each of the single units (e.g. A1 or B3) a ***stripe***, and an entire row across (e.g. A1,A2,A3) a ***stripe set.***

You will need to write the following functions:

```
struct blkdev *mirror_create(struct blkdev *disks[2]);
```

This creates a mirrored volume across two devices. If the devices are not the same size, print an error message and return NULL.

```
int mirror_replace(int i, struct blkdev *newdisk);
```

This replaces disk 'i' ('i' = 0 or 1), which may or may not have failed, and copies the existing data onto it before returning `SUCCESS`. If the new disk is not the same size as the old one, return `E_SIZE`.

```
struct blkdev *striped_create(int N, struct blkdev *disks, int unit);
```

This creates a striped volume across N disks, with stripes of '`unit`' sectors. If the disks are not the same size, print a message and return NULL. If the disks are not a multiple of '`unit`' blocks, the last few sectors on each disk will not be used. (e.g. given two disks of 5 sectors each and a unit size of 2, you will create a striped volume holding 8 sectors, and the last sector of each disk will not be touched.)

Note that there is no `striped_replace()` function, as striped volumes cannot recover from errors.

```
struct blkdev *raid4_create(int N, struct blkdev *disks, int unit);
```

This creates a RAID4 volume across N disks, striped in chunks of '`unit`' sectors, where `disks[N-1]` is the parity drive. Again, return NULL for a size mis-match, and do not use any sectors beyond the last multiple of the stripe size.

```
int raid4_replace(int i, struct blkdev *newdisk);
```

Again, replace disk 'i' (0<=i<N) with '`newdisk`', returning `SUCCESS`, or return `E_SIZE`. As in the mirrored case you will need to reconstruct and write data to the new drive before returning.

Deliverables

You will be responsible for the following files in your repository:

```
homework.c
mirror-test.c – this file creates and tests mirrored volumes
mirror-test.sh – run mirror tests
stripe-test.c – creates and tests a striped volumes
stripe-test.sh – run stripe tests
raid4-test.c – create and test raid4 volumes
raid4-test.sh – run RAID 4 tests
```

The `test-mirror.sh`, `test-stripe.sh`, and `test-raid4.sh` files will need to create any needed image files and pass them as arguments to `mirror-test`, `stripe-test`, and `raid4-test`.

C programs should handle test failures by using `assert()` statements or printing "ERROR" and calling `exit(1)` if a test failure is detected. Shell scripts should use the `echo` command to print "ERROR" and then call `exit 1`

You will need image files for your tests – you can create a zero-filled file with the dd command like this:

```
dd if=/dev/zero of=test.img bs=512 count=200
```

(the abbreviations stand for input file, output file, and block size) This creates a file named '`test.img`' containing 200 512-byte blocks (i.e. 102400 bytes) of zeroes. (or you can just create the file in a C program, write the correct number of zeroes to it, and close it)

You may find it useful to create files containing easy-to-distinguish values. To create a 1024-byte file containing only the character 'C', you can use the command:

```
dd if=/dev/zero bs=512 count=2 | tr '\0' 'C' > test.img
```

For stripe and RAID testing you may want to create multiple small (e.g. stripe-sized) files with different contents and concatenate them into one.

If you want to compare two binary files in a shell script you can use the cmp command, which returns true if the files are identical:

```
if ! Cmp file1 file2 ; then
      echo ERROR: files do not match
      exit 1
fi
```

If you are trying to figure out what exactly is in your disk image, you may find the 'od' ("octal dump") program useful; e.g. here is a dump of a file with 3 512-byte blocks initialized to 'C', 'D', and 'E' respectively, with addresses (on left) in decimal (-A d) and values printed as characters (-c):

```
Peters-MacBook-Air:hw3 pjd$ od -c -A d x
0000000    C   C   C   C   C   C   C   C   C   C   C   C   C   C   C   C
*
0000512    D   D   D   D   D   D   D   D   D   D   D   D   D   D   D   D
*
0001024    E   E   E   E   E   E   E   E   E   E   E   E   E   E   E   E
*
0001536
```

In a test script the following snippet of code may be handy:

```
Peters-MacBook-Air:hw3 pjd$ perl -e 'while (!eof(STDIN)) {$a = getc(STDIN);\
                                     printf “%s\n”, $a;}' < x  | uniq -c
 512 C
 512 D
 512 E
```

You may need to replace '%s' with '%x' or '%d' if you are working with RAID 4.

Mirror tests

You should test that your code:

- creates a volume properly
- returns the correct length
- can handle reads and writes of different sizes, and returns the same data as was written
- reads data from the proper location in the images, and doesn't overwrite incorrect locations on write.
- continues to read and write correctly after one of the disks fails. (call `image_fail()` on the image blkdev to force it to fail)
- continues to read and write (correctly returning data written before the failure) after the disk is replaced.
- reads and writes (returning data written before the first failure) after the other disk fails.

Stripe tests

You should test that your code:

- Passes all other tests with different stripe sizes (e.g. 2, 4, 7, and 32 sectors) and different numbers of disks.
- reports the correct size
- reads data from the right disks and locations. (prepare disks with known data at various locations and make sure you can read it back)
- overwrites the correct locations. (write to your prepared disks and check the results – using something other than your stripe code – to check that the write sections got modified.)
- fail a disk and verify that the volume fails.
- large (> 1 stripe set), small, unaligned reads and writes (i.e. starting, ending in the middle of a stripe), as well as small writes wrapping around the end of a stripe.

RAID 4 tests

RAID 4 tests are basically the same as the stripe tests, combined with the failure and recovery tests from mirroring.

Note that sharing **test** code on Piazza is **explicitly allowed** for this assignment. In particular, you may share code for `mirror-test.c`, `stripe-test.c`, and `raid4-test.c` and shell scripts.

Once you finish homework.c and start writing tests, you are probably less than half finished with the assignment..