You will be responsible for (1) writing test scripts to test your code, (2) measuring code coverage when you run your tests. You should automate your tests as shell scripts so that you are able to re-run them as necessary during development. These scripts will also be run when grading your submission.

Testing / shell scripting basics

[repeating some of your instructions for homework 1] Each test script should contain a series of tests, and after each test it should check to see whether the test passed; if not, it should exit with an error.

A shell script is just a file full of shell commands, like the ones you type in by hand, which can be run repeatedly. However you will need to use a number of shell features which you are unlikely to use by hand:

- line continuations – shell syntax is line-oriented; to continue a line use backslash:
```
echo this is a long line \
   that is continued on the next
```
- shell functions – see the 'fail' function described later
- shell variables:
```
variable=1      # no space on either side of '='
echo $variable
```
- command substitution - $(command) is replaced by the output generated by running 'command'. If the output has spaces in it you may need to use double-quotes - "$(command)".
- Conditional execution – you'll need a series of tests, and fail if any of them returns false:
```
check || fail
```
where '||' works the same as the logical OR operator in many languages. (if you want to check that a command fails, use '&&')
- String comparison – you'll use this to e.g. verify the output from 'ls -l':
```
test "$(ls -l dir/file.A)" = \
   '-rwxrwxrwx 1 student student 1000 Jul 13  2012 dir/file.A' || \
   fail Test 1 failed
```
- Compare two files with 'cmp':
```
cp /tmp/file.X dir/file.X
cmp /tmp/file.X dir/file.X || fail Test 1 failed
```
- Compare a file and standard input with 'cmp' (see later discussion of 'dd'):
```
cp /tmp/file.X dir/file.X
dd bs=100 if=dir/file.X | cmp - /tmp/file.X || fail Test 1 failed
```
- Verify file contents with 'cksum':
```
test "$(cat dir/file.A | cksum)" = '3509208153 1000' || \
      fail Test 1 failed
```
- Verify type:
```
test -d dir/dir1        # is a directory?
test -f dir/file.A      # is a file?
test ! -e dir/abcd      # 'abcd' does not exist
```

The following shell function can be used at the end of each test - '$*' refers to all the arguments passed, and '1' is a failing value.

```
fail(){
  echo FAILED: $*
  exit 1
}
```

Testing for questions 1 and 2

For these tests you will be running your executable with the '-cmdline' flag and providing a series of input lines. There's a shell scripting feature that lets you provide those lines from within the script:

```
./homework -cmdline -image disk.img <<EOF > /tmp/test1-output
ls
ls-l
get file.A /tmp/test1-file.A
quit
EOF
```

The lines up to (but not including) 'EOF' will be sent as input to the program. You can then use the same trick to compare with the expected results:

```
cmp /tmp/test1-output <<EOF || fail Test 1 failed
read/write block size: 1000
cmd> ls-l
dir1 drwxr-xr-x 0 0
file.A -rwxrwxrwx 1000 1
cmd> quit
EOF
```

The cmdline 'get' and 'put' commands copy a file out of the image ('get') and into the image ('put') from the external file system, and are used to test read and write. By default they use an operation size of 1000 bytes; you can change this with the 'blksiz' command:

```
$ ./homework -cmdline -image foo.img
read/write block size: 1000
cmd> blksiz 17
read/write block size: 17
```

After copying a file out of the disk image into the regular file system you'll need to verify its contents, which you can do by comparing it with the original file with 'cmp'. (unless you're still working on question 1, in which case you can use 'cksum' - at the end of the document there's a table of file checksums in the 'mktest' image.

You'll need to create a number of temporary files – a good place to put them is in the /tmp directory, and if you name them "test1-..." etc. you can clear out any old ones at the beginning of the test script with the command:

```
rm -f /tmp/test1-*
```

You'll need to create files of various sizes that you can write to your disk image and read back. There are a bunch of ways to do this, but a simple one to create a file e.g. 5100 bytes long is:

```
yes '0 1 2 3 4 5 6 7' | fmt | head –bytes 5100 > /tmp/test1-file.1
```

This creates a readable file with a distinctive pattern, which may help in identifying e.g. missing blocks.

Testing for question 3

Testing in FUSE mode is harder, since you have to convince the kernel to perform the operations rather than directly specifying them.

Although you'll want to run the homework program with the '-d' flag when you're debugging, it will be easiest to run it in the background when testing. The following lines will start your program running (in the background) mounted on the directory ./dir1, and ensure (see 'help trap') that the program is stopped and unmounted when the test script exits:

```
./homework -image disk.img ./dir1
test 'fusermount -u ./dir1' 0
```

Basic testing can use 'cat' or the 'cp' copy command. However you'll want to test with different read lengths (just like setting blksiz in command-line mode), which you can do with the 'dd' command:

```
dd if=dir/file.A bs=100 > /tmp/copy-of-file.A
```

where 'if' is the input file and 'bs' is the block size in bytes, causing it to read 'dir/file.A' with 100-byte read operations. As with 'fs_readdir', you won't see many path translation errors, as FUSE will usually call getattr first.

test with 'ls' or 'ls -l'. Note that you can't really test path translation errors here (or a lot of other cases) because FUSE will call fs_getattr first, and when it fails, FUSE won't call fs_readdir.

Test cases

fs_getattr: Test the following cases, which should cover all your path translation cases:

- /a/b/c, /b/c where 'b' doesn't exist
- /a/b/c, /b/c where 'b' is a file
- /a/b/c, /a/c where 'c' doesn't exist
- /a/b/c, '/c where 'c' is a file
- /a/b/c, '/c' where 'c' is a directory

fs_readdir: You should test the following cases:

- / i.e. the root directory
- /a, /b/a where 'a' is a file
- /a, /b/a where 'a' is a directory

fs_read: You should test with block sizes:

- less than 1024 (e.g. 111 bytes)
- 1024 bytes
- 1024 < size < 2048 (e.g. 1517 bytes)
- > 2048 bytes (e.g. 2701 bytes)

Test reading the following file sizes:

- zero-length (e.g. dir1/file.0 on the mktest image)
- small (< 6KB, i.e. using only direct blocks)
- medium (< 262KB)
- large (>262KB)

In each case verify that you got the right number of bytes and that they're the right ones.

fs_mknod, fs_mkdir: the cases are the same for both:

- bad path /a/b/c - b doesn't exist
- bad path /a/b/c - b isn't directory
- bad path /a/b/c - c exists, is file
- bad path /a/b/c - c exists, is directory
- good path – verify successful completion
- good path, but fail due to full directory

Note that you can create a zero-length file with the 'touch' command.

fs_unlink:

- bad path /a/b/c - b doesn't exist
- bad path /a/b/c - b isn't directory
- bad path /a/b/c - c doesn't exist
- bad path /a/b/c - c is directory
- /a/b/c - actually deletes 'c'

Remove zero-length, small (<6K), medium (<262K), large (>262K) files. Use read-img afterwards to verify that all blocks were freed. In particular, the last 2 lines of output will only contain 4 words ("unreadable inodes:" and "unreachable blocks:") if there are no lost blocks:

```
test "$(./read-img foo.img | tail -2 | wc --words)" = 4 || fail Test 1 failed
```

fs_truncate: you can test this with the 'truncate' shell command ('man truncate' for details). Check that it works for offset=0, and that it leaves the file alone for offset >0. Like fs_unlink, check that it works for zero, small, medium, and large files and that there are no lost blocks. (you should have factored out the common code between fs_truncate and fs_unlink, anyway)

fs_rmdir: very similar to fs_unlink.

- bad path /a/b/c - b doesn't exist
- bad path /a/b/c - b isn't directory
- bad path /a/b/c - c doesn't exist
- bad path /a/b/c - c is file
- directory not empty
- /a/b/c - actually removes 'c'

fs_write:

- ignore path validation (can't test with FUSE, should be the same factored code anyway)
- write with the same set of 4 different block sizes. (<1024, 1024, >1024, >2048)
- overwrite existing files with same set of block sizes, using the 'conv=notrunc' option to 'dd'
- create small, medium, and large files
- run out of space on the disk. verify that the resulting file isn't "broken" after you do that – i.e. you can read it, and it is as long as 'ls -l' says it is. Note that with a repeatable test script the file should always be the same size, so you can do something like this (change '2048' as necessary):

```
test $(wc --bytes dir/file) = 2048 || fail Test 1
```

fs_chmod, fs_utime: test that they actually change the values:

```
chmod 754 dir/file.A
test "$(ls -l dir/file.A)" = \
     '-rwxr-xr-- 1 student student 1000 Jul 13  2012 dir/file.A'
touch -d 'Jan 01  2000' dir/file.A
test "$(ls -l dir/file.A)" = \
     '-rwxr-xr-- 1 student student 1000 Jan  1  2000 dir/file.A'
```

fs_rename: This has a bunch of cases. Remember that we cheat in two cases – (a) no moving files across directories, and no replacing the destination file.

- mv /a /b, mv /d/a /d/b - 'a' doesn't exist
- mv /d/a /d/b - 'd' is a file
- mv /a /b, mv /d/a /d/b - 'b' exists and is a file
- mv /a /b, mv /d/a /d/b - 'b' exists and is a directory
- mv /d1/a /d2/b – no moving between directories
- mv /a /b, mv /d/a /d/b - success

Code coverage

You should verify how thoroughly your code has been tested by using the gcov tool. To do this:

1. erase your old binaries and compile new ones with code coverage support compiled in:
   ```
   make clean
   make COVERAGE=1
   ```
2. If you've been changing your code you'll probably want to remove the old test output files:
   ```
   rm *.gcno *.gcda
   ```
3. Run your tests, which will create various files named *.gcno and *.gcda.
4. Now run the 'gcov' tool, which takes a source file name as its argument:
   ```
   $ gcov homework
   File 'homework.c'
   Lines executed:58.60% of 558
   Creating 'homework.c.gcov'
   ```

Now you have a file named 'homework.c.gcov' which is annotated to indicate how many times each line was executed, with '#####' flagging lines which have not been executed. Oh, and 58.6% is pretty lousy test coverage – I only ran a few tests before generating this example.

Note that you aren't going to get 100% test coverage, as a few failure cases are going to be difficult to trigger. However you should look through the .gcov file so that you understand all of the cases where code wasn't executed and are fairly certain that the missed code is correct. (in particular, think about whether my grading scripts might catch errors in some of those cases...)

For your final submission you should run all your test cases (i.e. test-1.sh, test-2.sh, test-3.sh) and then generate a copy of homework.c.gcov that you submit to your repository.

Contents of disk image generated by ‘mktest’

```
$ ls -l
total 4
drwxr-xr-x 1 student student    0 Jul 13  2012 dir1
-rwxrwxrwx 1 student student 6644 Jul 13  2012 file.7
-rwxrwxrwx 1 student student 1000 Jul 13  2012 file.A
$ cksum file.*
94780536 6644 file.7
3509208153 1000 file.A
$ cd dir1
$ ls -l
total 136
-rwxrwxrwx 1 student student      0 Jul 13  2012 file.0
-rwxrwxrwx 1 student student   2012 Jul 13  2012 file.2
-rwxrwxrwx 1 student student 276177 Jul 13  2012 file.270
$ cksum file.2*
3106598394 2012 file.2
1733278825 276177 file.270
```