CSE-506 (Fall 2016) Homework Assignment #2
(100 points, 17% of your overall grade)
Version 5b (11/5/2016)
Due Wednesday 11/09/2016 @ 11:59pm

* PURPOSE:

To become familiar with the VFS layer of Linux, and especially with extensible file systems APIs. To build a useful file system using stacking technologies.

You will use the "wrapfs" stackable file system as a starting point for this assignment. You will modify wrapfs to add OPERATION-TRACING support, save the trace file to disk, and be able to replay it in user-land.

* TEAMING:

For HW2, you may work alone or in a pair (1-2 people at most). If you choose to work in a pair, you will be graded the same as working alone, but at the end of the course, I take group sizes into account when assigning final course grades. See my grading policy online for details.

If you work in a pair, you MUST email me the full names of both team members, SBID, and especially your CS AD ID name (which should match your SBU NetID). I will create a shared GIT repo for each team, so you will NOT be using your individual GIT repos. You must send me this information no later than 10/22/2016: afterwards, I would not allow new teams to form or existing teams to split. You don't have to email me if you work alone.

Note that being the second homework assignment in the class, it is on purpose made a bit more open ended. Not every little detail would be given to you as in HW1. Your goal would be to follow the spec below, but to also think about how to address all sorts of corner cases that I do not specifically list below. You will have more freedom here to design a suitbale system and implement it. As time goes by this semester, I expect all of you to demonstrate an increased level of maturity and expertise in this class.

* RESOURCES

For this assignment, the following resources would be handy. Study them well.

(a) The Linux kernel source code, obviously. Pay special attention to the Documentation/ subdirectory, esp. files related to locking, filesystems/*.txt, vfs.txt, and others. There's a lot of good stuff there, which you'd have to find and read carefully.

(b) The Wrapfs kernel sources in the hw2-USER git repository (or the group repository) that each of you have, under fs/wrapfs. Note also the file Documentation/filesystems/wrapfs.txt. This Wrapfs file system is under 2,000 LoC, and hence is easier to study in its entirety.

(c) Assorted papers related to stackable file systems which were published here:

http://www.fsl.cs.sunysb.edu/project-fist.html

Especially useful would be the following:

"Tracefs: A File System to Trace Them All"
"A Stackable File System Interface for Linux"
"Extending File Systems Using Stackable Templates"
"FiST: A Language for Stackable File Systems"
"On Incremental File System Development"
"UnionFS: User- and Community-oriented Development of a Unification Filesystem"
"Versatility and Unix Semantics in Namespace Unification"
"I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System"

(d) Browsable GIT-Web sources here, especially wrapfs-4.6:

http://git.fsl.cs.sunysb.edu/

Look at source code for other file systems. The stackable file system eCryptfs may be helpful.

* INTRODUCTION:

In a stackable file system, each VFS-based object at the stackable file system (e.g., in Wrapfs) has a link to one other object on the lower file system (sometimes called the "hidden" object). We identify this symbolically as X->X' where "X" is an object at the upper layer, and X' is an object on the lower layer. This form of stacking is a single-layer linear stacking.

* DETAILS (KERNEL CODE):

Write a stackable, tracing file system called "trfs." Trfs should capture records of file system activity for a number of file system operations. For example, for the ->open method, you should record the file name, open mode, permissions, and the return result (or errno). Each record you produce should be appended to a file on disk (e.g., a file directly on the lower file system). Records written should use in binary format so they are compact. Records should include a single byte encoding the record type (e.g., open, close, read, write, etc.). For example, for ->open, the record written can be something like this (2B means "two bytes long"):

<4B: record ID>
<2B: record size that follows in bytes>
<1B: record type>
<4B: open flags>
<4B: permission mode>
<2B: length of pathname>
<variable length: null-terminated pathname>
<4B: return value or errno>

Each record should have a "record ID," a monotonically increasing sequence number, followed by the total number of bytes in the rest of the record. Each record should include the return status (or errno) of the f/s method.

You should intercept as many f/s events as possible, mainly from inode,

dentry, file, and superblock operations, and possibly others.

You will need to designate a filename where the trace records are written. This should be passed as a mandatory mount option "-o tfile=/path/name". You will have to update the wrapfs code to allow a pathname to be passed as a mount option (see how, for example, eCryptfs and other f/s pass mount-time options). The tfile in question should be opened for appending and new records added to it. The tfile should NOT be exposed through the stackable file system itself, so it's best to place it outside the lower mounted-on directory.

You may find that some form of locking is required to ensure that records are written sequentially. A file system can be invoked in parallel with several syscalls operating inside the f/s code concurrently. So you will have to figure out a way to get the records appended sequentially to the tfile, without losing, duplicating, or reordering records (that's why a consistent, monotonically increasing record ID would be useful). You may have to use an auxiliary data structure to append records to, assisted by a kernel thread, rather than writing directly to the tfile from inside the f/s.

Often, it is useful to trace only a subset of system calls. Implement a bitmap in trfs, where each traced f/s method is enabled/disabled by a single bit. Your bitmap should be at least 32 bits long, and possibly 64-bits long. You should pass the bitmap to trfs via an ioctl(2), and store the bitmap in the kernel (think where's the best place to keep it in memory). Then, each time you're about to produce a record for an f/s method, only do so if the corresponding bit is 1 (0 means skip).
By default all tracing (and bits) should be enabled.

To implement trfs, you can just edit the fs/wrapfs/* code. A better way would be to make a copy of all wrapfs sources to fs/trfs, and edit them there. That way you can always refer to the unmodified wrapfs code. Remember to git add/commit/push any new source files.

Mounting trfs can be done as follows:

```
# mount -t trfs -o tfile=/tmp/tfile1.txt /some/lower/path /mnt/trfs
```

(Note that you may need to first run "insmod ./fs/trfs/trfs.ko" to insert the new trfs module; of course, you'll need to remove an older one that's already loaded, if any.)

After mounting, you should be able to "cd" to /mnt/trfs and issue normal file system commands. Those commands will cause the VFS to call methods in trfs. You can stick printk lines in trfs to see what gets called and when. Your actions in /mnt/trfs will invoke trfs methods, which you will intercept, and write to the tfile.

* DETAILS (USER CODE):

Write a user-level C program called "treplay" that will read in a tfile, and replay its operations one by one. Your program would have to parse the records, map them to the right system calls, and then execute the system calls.

You will find that VFS operations map closely but not perfectly to system

calls. Figure out what's the best way to map each VFS op to a system call (or perhaps more than one syscall).

Usage of the program is:

```
$ ./treplay [-ns] TFILE
```

TFILE: the trace file to replay
-n: show what records will be replayed but don't replay them
-s: strict mode -- abort replaying as soon as a deviation occurs

By default, treplay should display the details of records to be replayed, the system call to be replayed (if different than what the record states), then replay the syscall. Finally, display the result of the syscall executing now compared to what the return value was originally (e.g., a record of open(2) may say that the traced op succeeded, but when you replay it, it may fail, so show those differences). If the -n option is given, you just show the info without replaying it.

By default, if replaying a syscall produces a different result than before, then just keep going and replay the next record. But if the -s flag is given, then you abort replaying as soon as a failure occurred, where failure is defined as a deviation between the result of the traced record and its replaying right now. Note that if your traced record said that, say, mkdir(2) originally failed with -ENOENT, then a "success" here means that you replay that mkdir call and ALSO get an ENOENT! A "failure" here would be a deviation -- meaning that you replayed mkdir and got anything other than ENOENT (including when mkdir returned 0, i.e., succeeded).

Next, write a program to control which f/s methods are intercepted and traced.

```
$ ./trctl CMD /mounted/path
```

This program should issue an ioctl(2) to trfs to set the bitmap of which f/s methods to trace. CMD can be:

"all": to enable tracing for all f/s methods
"none": to disable tracing for all f/s methods
0xNN: hex number to pass bitmap of methods to trace or not

/mounted/path is the path of your mounted f/s (e.g., /mnt/wrapfs). Also,

```
$ ./trctl /mounted/path
```

should retrieve the current value of the bitmap and display it in hex.

* GIT REPOSITORY

For this assignment, we've created clean GIT kernel repositories for each of you. Do not use the one from HW1 for this second assignment; but you can copy a good .config you've had in HW1 and use that for your HW2 kernel. You can clone your new GIT repo as follows, using similar instructions as in HW1:

```
# git clone ssh://USER@scm.cs.stonybrook.edu:130/scm/cse506git-f16/hw2-USER
```

Note that if you don't have enough disk space on your VM, it'll be a good idea to remove the older hw1-USER repo to make space.

If you want, you can use my kernel config as a starting point, and adapt it as needed:

http://www3.cs.stonybrook.edu/~ezk/cse506-f16/cse506-f16-kernel.config

* SUBMISSION

Simply git-commit and git-push your changes to your cloned git repository; a successful git-push will result in an email going to you, confirming the commit and push. Don't forget to include the README.HW2 file. If for some reason you decided to add other file(s) to your GIT repository, please mention this in README.HW2 so we don't miss it during grading (and justify why a new file was needed).

All new files should be added to the hw2-USER/hw2/ subdir in your repo (e.g., user-land code, additional Makefile, etc.).

Your README.HW2 should detail your design for trfs, the ioctl, mounting code, user-level code, anything special/different you did, extra credit design, etc.

Also note that we will just do a git clone of your final repository and run make, make modules_install, and make install as usual. You must not assume that we will do ANY modification in your code. Your code MUST compile and run as it is. You will lose all points in submission section IF your code doesn't compile/run as checked out.

If you attempt any EXTRA CREDIT functionality, your README.HW2 MUST specify exactly how to compile your code with extra credit enabled. By default, your code MUST NOT compile extra credit code.

* EXTRA CREDIT (OPTIONAL, MAX 20 pts, plus bug fixes)

If you do any of the extra credit work, then your EC code must be wrapped in

```
#ifdef EXTRA_CREDIT
        // EC code here
#else
        // base assignment code here
#endif
```

A. [10 pts] validate records' integrity

Prefix each record you write with a 32-bit CRC (checksum) value. You can calculate the CRC value of a record after you're constructed it in memory, then store and write the CRC to the record itself on disk. Use CryptoAPI to calculate CRCs. In user-level, the treplay program should recalculate the CRC of each record and compare it. If the record matches, continue; if it doesn't match, it means the record got corrupted (you should test for this condition). In that case, print an error message and abort or continue (based on the value of the -s flag).

Note: CRC algorithms aren't always standard. If not, then use a different, standard hashing algorithm, such as MD5 or SHA1.

B. [10 pts] incremental ioctl changes

Implement kernel support in trfs's ioctl, so that you can individually turn on/off any single f/s method's tracing. The user-land code should support additional meanings for CMD, for example

```
$ ./trctl +open -read +write /mounted/path
```

would issue a single ioctl to enable tracing open and write, and disable tracing read. You'll have to have a different ioctl type from the one in the main assignment, which re/sets the entire bitmap. Here, you'll need to manipulate individual bits in the bitmap, using logical OR and AND values.

C. [up to 10 pts] wrapfs bug fixes (optional)

If you find and fix true bugs in wrapfs in 4.6, you may receive special extra credit, depending on the severity of the bug and the quality of the fix.

* ChangeLog: a list of changes that this description had

v1: original version
v2: rename "tracefs" to "trfs" to avoid conflict w/ Linux kernel
v3: incorrect usage fixed for user-level program
v4: fix sizes of open() params
v5: clarify default tracing bits status; use MD5 if CRC not standard.