CSE-506 (Spring 2018) Homework Assignment #1

(100 points, 16% of your overall grade)
Version 2 (2/7/2018)
Due Wednesday 2/28/2018 @ 11:59pm

* PURPOSE:

To get your Linux kernel development environment working; to make small changes to the kernel and test them out; to learn about system calls.

* BACKGROUND:

Deduplication is very useful and important nowadays because it can save storage space, but many OSs do not support this feature natively (yet). Your task is to create a new system call that can take two input files, then deduplicate them if they have identical content.

Note that while we give you more details below, it is up to you to inspect the kernel sources to find out similar helpful examples of code; those will provide you with even greater details than what we provide here.

The expected amount of written code for this assignment would be 1000-2000 lines of kernel code, and another 200-300 lines of user-level code. Note, however, that a lot of time may be spent reading existing sources and debugging the code you write.

* TASK:

Create a Linux kernel module (in vanilla 4.6.y Linux that's in your HW1 GIT repository) that, when loaded into Linux, will support a new system call called

```
        sys_dedup(const char *infile1, const char *infile2, char *outfile,
                  u_int flags)
```

where "infile1" and "infile2" are the input files to deduplicate.
If the two files have identical content, and they're owned by the same user, then you should delete file infile2 and hard-link it to infile1: this'll deduplicate the two files by merging them into one. This would be similar to these command line steps:

```
$ if cmp file1 file2; then rm -f file2 && ln file1 file2; fi
```

You should be efficient: if the file's sizes or owners differ, the files cannot be identical, so there's no point in deduplicating them. Otherwise, open both files for reading, compare them byte-by-byte, and if they're identical, dedup them.

Your syscall should return -errno on any kind of error you found along the way. Upon success, the syscall should return the number of bytes successfully deduped (by default, if successful, it'd be the size of file1).

Flags (showing the cmd line option and the hex value your program should take):

```
-n 0x01: don't actually dedup.  Just return -errno or #identical bytes if
         the two files are identical.
-p 0x02: partial dedup output, using output file outfile
-d 0x04: enable debugging
```

If the -p option is used, and outfile is provided, and it is not null, and the two files have a common partial sequence starting at byte 0, even if the files have different sizes, then create a new file "outfile" whose contents is the partial bytes that matched. Only do so starting at byte 0 of infile1/infile2, and only do so for the first bytes starting at 0 that match. Don't worry if there are additional partial sequences of common bytes between the two files. For example, assume the two files have this content:

```
file1: abc123xyz
file2: abc145xyzw
```

then the -p option should write "abc1" to outfile and return the number 4 (number of duplicate bytes starting at offset 0). Note that -p means you do NOT deduplicate infile1 and infile2 (don't hardlink them).

Debugging: if the -d option is passed, enable debugging in the kernel. Basically printf some useful messages at various stages of the syscall, to show progress as well as errors. I expect at least 10 different debug messages supported for this option (the more, the better -- but don't go wild and add hundreds of printf messages).

** Error Checking:

The most important thing system calls do first is ensure the validity of the input they are given. You must check for ALL possible bad conditions that could occur as the result of bad inputs to the system call. In that case, the system call should return the proper errno value (EINVAL, EPERM, EACCES, etc.) Consult the system errno table and pick the right error numbers for different conditions.

The kinds of errors that could occur early during the system call's execution are as follows (this is a non-exhaustive list):

- missing arguments passed
- null arguments where non-null is expected; or vise versa
- pointers to bad addresses
- invalid flags or combination of flags and arguments
- input files cannot be opened or read
- output file cannot be opened or written
- input or output files are not regular, or they point to the same file (no point in deduplicating the same file)
- ANYTHING else you can think of (the more error checking you do, the better)

After checking for these errors, you should open the input and output files, compare inputs, and write out the output file if needed. Be sure to handle various flags and error conditions that may arise.

Your code must be efficient. Therefore, do not waste extra kernel memory (dynamic or stack) for the system call. Make sure you're not leaking any memory. On the other hand, for efficiency, you should copy the data in chunks that are native to the system this code is compiled on, the system page size (PAGE_CACHE_SIZE or PAGE_SIZE). Hint: allocate one page as temporary buffer.

Note that the last page you read/write could be partially filled and that your code should handle zero length input files as well. (Two zero-length files are considered identical in content, even if there's no actual content. So you can "dedup" them and return 0 as the number of bytes deduplicated.)

The output file should be created with the user/group ownership of the running process, and its protection mode should NOT be less than the input file(s).

Both the input and output files may be specified using relative or absolute pathnames. Do not assume that the files are always in the current working directory.

If an error occurred in trying to write some of the output file, the system call should NOT produce a partial output file. Instead, remove any partially-written output file and return the appropriate error code.

Similarly, think about partial failures when you determine that infile1 and infile2 are indeed identical: you have to delete infile2 and then hardlink them. What if one of those operations succeeds and the second one fails? How would you recover.

Write a C program called "xdedup" that will call your syscall. The program should have no output upon success and use perror() to print out information about what errors occurred. The program should take any combination of flags as listed above, as well as three arguments:

- input1 file name
- input2 file name
- output file name (optional)

You can process options using getopt(3). You should be able to execute the following command:

```
        ./xdedup [-npd] outfile.txt file1.txt file2.txt
```

* SYSTEM CALLS IN the Linux Kernel:

As of kernel 2.6, a kernel module is not allowed to override system calls (long story, I'll tell you in class :-) So I am giving you a patch that can add a new syscall to Linux. Note that the patch was written for an older 64-bit kernel, and it was written to create a system call called "xcrypt". So you'll have to update the patch (after applying it) before it will work for your kernel (change all "xcrypt" mentions to "xdedup").

Note also that the patch not just updates kernel code, but also creates a sample hw1/ subfolder under your kernel tree. The files in the hw1/ folder provide a working example of a module that can hook into the kernel and set the new syscall dispatch routine, as well as sample user code. Study these files carefully to understand what the code does. And don't forget to git-add (and commit + push) those files.

The patch creates a single syscall that takes only one parameter: a "void*", into which you'd have to pack your args depending on the mode.

You can download the patch from:

```
        http://www.cs.stonybrook.edu/~ezk/cse506-s18/cse506-syscall.patch
```

It is recommended you first get a working kernel in your VM without the special syscall patch (this'll be a challenge on its own). Only then apply the patch and test it. Afterwards, you can get started with the heart of the assignment -- the dedup system call.

* A BASELINE KERNEL TEMPLATE

To make getting started easier, we've provided you with a baseline template and your own Virtual Machine (VM). The template VM includes a working Linux kernel you'd have to configure and build. See the online class instructions how to start your own VM (using VMware VSphere client) and login to it. In this assignment, you will do all programming in your own personal VM.

To get root privileges, use sudo, but find the proper instructions here:

1. login to the scm machine, then run
2. cat /scm/cse506git-s18/.p
3. once you login as "student" to your VM, change the root passwd asap with

```
   the command "passwd" and follow the prompts.
4. You can then use "sudo" to become root.

You will have to login as root to your own VM, then you'll need to compile
the kernel and the test software:

# cd /usr/src
# git clone ssh://USER@scm.cs.stonybrook.edu:130/scm/cse506git-s18/hw1-USER
        (where "USER" is your CS userid)
# cd hw1-USER
# make config
        NOTE: Check online instructions how to configure a minimal kernel.
              your hw1 will be graded on this minimal configured kernel.
              Refer to "SUBMISSION" section for details.
# make
# make modules
# make modules_install install
# reboot
        Ensure you've booted into YOUR 4.6 kernel...

If the above works, download the cse506-syscall.patch and apply it to your
hw1-USER tree.  See the patch(1) program for help how to apply patches.  You
may have to reconfigure your kernel to auto-generate the new system call
vector numbers.  Don't forget to "git add hw1", then "git commit -a" all new
files, and finally "git push" so these changes are pushed to your remote git
repository permanently.  Use the "git status" command frequently to find out
which files in your repository are new, modified, never added, etc.

After rebuilding your patched kernel, you'd have to reinstall the kernel as
per the instructions just above, and reboot again to run the patched kernel
that supports the new syscall.  Once it comes back up, if all works well,
then you can build the overriding syscall module and try the new system
call:

# cd hw1-USER/hw1
# make
        To build the HW1 sample files.

Check the source files in the hw1 subdir and study them.  The sys_xdedup.c
implements a dummy system call that simply returns 0 if you pass a non-null
argument to the system call, and returns EINVAL if you pass zero.  This is
your syscall template to implement.

The xhw1.c file is a sample user level program to pass a number to the
system call.  And the install_module.sh script is used to load up the new
kernel module (and unload an old one first, if any).  To test this system
call, try this:

# sh install_module.sh
# dmesg | tail
        (use this optional command to see the kernel modules loaded.
         You'll see some messages when a module is un/loaded.)
# ./xhw1 17
syscall returned 0
# ./xhw1
syscall returned -1 (errno=22)

Run the "dmesg" command to see the last printk messages from the kernel.

Note that the system call is designed to pass one "void*" arg from userland
to the kernel.  So, in order to pass multiple arguments, pack them into a
single void*, for example:

struct myargs {
        int arg1;
        char arg2[24];
};
struct myargs args;
args.arg1 = 100;
strcpy(args.arg2, "hello world");
rc = mysyscall((void *) &args);
```

* READING FILES FROM INSIDE THE KERNEL

Here's an (old) example function that can open a file from inside the
kernel, read some data off of it, then close it. This will help you in this
assignment. You can easily extrapolate from this how to write data to
another file. (Warning: the old code below is from 2.4. Adapt it as
needed.)

```
/*
 * Read "len" bytes from "filename" into "buf".
 * "buf" is in kernel space.
 */
int
wrapfs_read_file(const char *filename, void *buf, int len)
{
    struct file *filp;
    mm_segment_t oldfs;
    int bytes;

    filp = filp_open(filename, O_RDONLY, 0);
    if (!filp || IS_ERR(filp)) {
        printk("wrapfs_read_file err %d\n", (int) PTR_ERR(filp));
        return -1;  /* or do something else */
    }

    if (!filp->f_op->read)
        return -2;  /* file(system) doesn't allow reads */

    /* now read len bytes from offset 0 */
    filp->f_pos = 0;  /* start offset */
    oldfs = get_fs();
    set_fs(KERNEL_DS);
    bytes = filp->f_op->read(filp, buf, len, &filp->f_pos);
    set_fs(oldfs);

    /* close the file */
    filp_close(filp, NULL);

    return bytes;
}
```

* TESTING YOUR CODE:

You may choose to hard-code the syscall into your kernel, or do it as a
loadable kernel module (loadable kernel modules makes it easier to
unload/reload a new version of the code). Write user-level code to test
your program carefully.

If you choose a kernel module, then once your module is loaded, the new
system call behavior should exist, and you can run your program on various
input files. Check that each error condition you coded for works as it
should. Check that the modified file is changed correctly.

Finally, although you may develop your code on any Linux machine, we will
test your code using the same Virtual Machine distribution (with all
officially released patches applied as of the date this assignment is
released), and using the Linux 4.6.y kernel. It is YOUR responsibility to
ensure that your code runs well under these conditions. We will NOT test or

demo your code on your own machine or laptop! So please plan your work accordingly to allow yourself enough time to test your code on the machines for which we have given you a login account (these are the same exact machines we will test your code on when we grade it).

Additionally, we strongly suggest that you enable CONFIG_DEBUG_SLAB and other useful debugging features under the "Kernel hacking" configuration menu. When grading the homework, we will use a kernel tuned for debugging---which may expose bugs in your code that you can't easily catch without debugging support. So it's better for YOU to have caught and fixed those bugs before we do.

Lastly, note that even if your system call appears to work well, it's possible that you've corrupted some memory state in the kernel, and you may not notice the impact until much later. If your code begins behaving strangely after having worked better before, consider rebooting your VM.

* STYLE AND MORE:

Aside from testing the proper functionality of your code, we will also carefully evaluate the quality of your code. Be sure to use a consistent style, well documented, and break your code into separate functions and/or source files as it makes sense.

To be sure your code is very clean, it should compile with "gcc -Wall -Werror" without any errors or warnings! We'll deduct points for any warning that we feel should be easy to fix.

Read Documentation/CodingStyle to understand which coding style is preferred in the kernel and stick to it for this assignment. Run your kernel code through the syntax checker in scripts/checkpatch.pl (with the "strict" option turned on), and fix every warning that comes up. Cleaner code tends to be less buggy.

If the various sources you use require common definitions, then do not duplicate the definitions. Make use of C's code-sharing facilities such as common header files.

You must include a README file with this and any assignment. The README file should briefly describe what you did, what approach you took, results of any measurements you might have made, which files are included in your submission and what they are for, etc. Feel free to include any other information you think is helpful to us in this README; it can only help your grade (esp. for Extra Credit).

* SUBMISSION

You will need to submit all of your sources, headers, scripts, Makefiles, and README. Submit all of your files using GIT. See general GIT submission guidelines on the class Web site. You must remember to "git add", "git commit" and "git push" the right files (don't add binaries and regenerable files). If you don't follow git instructions properly, your code will not arrive to the GIT server for grading!

As part of this assignment, you should also build a 4.6.y kernel that's as small as you can get (but without breaking the normal CentOS boot). For example, there are dozens of file systems available: you need at least an ext3 or ext4), but you don't need XFS or Reiserfs. Commit your .config kernel file into GIT, but rename it "kernel.config". We will grade you on how small your kernel configuration is with the following exceptions:

1. All start time servers that run by default in the VM provided, should start without failing.

2. We won't count "kernel hacking" options: so you may enable as many of

them as you'd like.

Ideally, your .config file should have fewer than 1000 CONFIG* lines (not counting the "kernel hacking" options). (My minimal kernel config files are around 600-700 options.)

To submit new files, create a directory named "hw1" inside hw1-<user> directory that you checked out. Remember to git add, commit, and push this new directory. Put ALL NEW FILES that you add in this directory, including the kernel.config, README, etc. This may include user space program (.c and .h files), README, kernel files (in case you are implementing system call as a loadable kernel module), Makefile, kernel.config, or anything else you deem appropriate.

For existing kernel source to which you make modification, use git add, commit, and push as mentioned on class web site.

There must be a Makefile in hw1/ directory. Doing a "make" in hw1/ should accomplish the following:

1. Compile user space program to produce an executable by the name "xdedup". This will be used to test your system call.

2. In case you are implementing system call as a loadable kernel module, the "make" command should also produce a sys_xdedup.ko file which can be insmod into the kernel.

(Use gcc -Wall -Werror in makefile commands. We will anyway add them if you don't :-)

The hw1/ directory should also contain a "kernel.config" file which will be used to bring up your kernel.

Note that in case you are implementing system call directly in the kernel code (and not as a loadable kernel module), then just compiling and installing your kernel should activate the system call.

Just to give you an idea how we will grade your submission: We will check out your git repository. We will use kernel.config file in hw1/ subdirectory to compile and then install your kernel. We will then do a make in hw1/ subdirectory. If your implementation is based on a loadable module, we will expect sys_xdedup.ko to be present in hw1/ after doing a make. We will then insmod it and use hw1/xdedup (also generated as part of make) to test your system call on various inputs. Note that insmod step will be skipped in case you implement system call directly into the kernel.

PLEASE test your code before submitting it, by doing a dry run of above steps. DO NOT make the common mistake of writing code until the very last minute, and then trying to figure out how to use GIT and skipping the testing of what you submitted. You will lose valuable points if you do not get to submit on time or if you submission is incomplete!!!

Make sure that you follow above submission guidelines strictly. We *will* deduct points for not following this instructions precisely.

* EXTRA CREDIT (OPTIONAL, total 15 points)

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 points.

Support a new option -s, to calculate infile1 and infile2 checksum using SHA1, using Linux's built-in CryptoAPI support. Write the two checksums to outfile (must be provided) in the same format that sha1sum does:

```
$ sha1sum file1 file2
9b320b386b6c295091cc3a828d729e7309ea00be  file1
cac2e3fac2ecb54051ab8d20358f10e1a9bed522  file2
```

You'd have to figure out how this -s option interacts with the remaining flags and what would make the most sense to support, in terms of combinations of options.

[D] 5 points

Extra credit at grader's discretion, for any particularly clever solutions/enhancements (up to 5 pts). Be sure to document anything extra you did in your README so the graders notice it.

Good luck.

* Copyright Statement

* Change History:

2/4/18: first draft
2/7/18: reviewed by TAs