# Project 2

Adding a System Call to the Kernel,
Working with git, and Creating Kernel Patch Files

In this assignment you will use the virtual machine (VM) and kernel tree from Project 1 to develop a patch that adds a custom system call to the kernel.

## 1. Configure git so that it knows who you are

Whenever you make a commit to a git repository, you are actually adding a patch to the repository. This patch contains information like the author of the patch (i.e. you), the date and time, the files that were changed, and which lines in the files were changed.

So, before you can start making commits to your local copy of Linus's repository, you need to tell git who you are. That way git can put your name on all the patches you write. Don't worry. Linus won't see anything you commit to your local copy of his repository. You would have to push the changes to him for that to happen – and Linus's repository is not setup to accept pushes anyway.

Configuring git is easy. For example, if your name was **Jane Smith** and your e-mail address was **jsmith@drexel.edu**, you would type the following into your shell:

```
$ git config --global user.name "Jane Smith"
$ git config --global user.email jsmith@drexel.edu
```

That's it. Now git knows who you are! This information is saved in git's configuration file, which lives in your home directory at **`~/.gitconfig`** You can view this file if you are curious. It's just a text file and is easy to read and modify. Since your name and e-mail address are stored now in **`~/.gitconfig`**, git will remember this information. So, you don't need to do this again. You only ever have to do this once.

**2. Make sure your 'base' branch is ready**

In Project 1, you setup your **base** branch. If you recall, we based this branch on tag **v5.11**. You are going to produce your patch for version 5.11 of the kernel. So, make sure you are working in the **base** branch before starting.

```
~/src/linux$ git checkout base
```

Also, make sure your **base** branch is clean. In other words, let's check to make sure you haven't changed any files that are being tracked by the git repository. You can do this with git's **status** command:

```
~/src/linux$ git status
On branch base
nothing to commit, working tree clean
```

Great! If your tree wasn't clean, you would instead see a list of files that have been modified since the last commit. In a bit we will start modifying code. Once you make some changes to some files, you can use **git status** to keep track of which files you have messed with. This is incredibly useful.

While we are at it, let's make sure your **base** branch is still on the same commit as the **v5.11** tag. So, let's tell git to list all the commits that have occurred between the tag **v5.11** and your **base** branch:

```
~/src/linux$ git log v5.11..base
```

This *should* do nothing at all (unless you have been busy adding code to the kernel in your spare time). Excellent! You're all setup and ready to go!

### 3. How to view history

Since we're here, this is a good opportunity to show how you can use **log** to better understand the state of the repository. For example, you can see all of the commits that occurred between version v5.10 and v5.11 of the kernel with the following command:

```
~/src/linux$ git log v5.10..v5.11
```

Wow! That's a lot of commits! You can scroll up/down with **j**/**k** and quit with **q**. As you can see, the **log** command shows entire commit messages (but doesn't show the code changes). If you want a more concise list of commits, use **`log --oneline`** instead:

```
~/src/linux$ git log --oneline v5.10..v5.11
```

This will only show each commit's **unique SHA-1 hash** (this is how we refer to a specific commit) and **brief commit message**. This is incredibly useful. I use this constantly to see what code has changed between two points in the code history.

### 4. How to view an individual commit

To view the code changed by a specific commit, use git's **show** command. For example, when I ran the previous command, these are a few of the commits that were displayed:

```
cf050f96e097 Revert "drm/amd/display: Update NV1x SR latency values"
b8776f14a470 Merge git://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf
a0f85e38a3c2 cifs: do not disable noperm if multiuser mount option is not provided
e82553c10b08 Revert "mm: memcontrol: avoid workload stalls when lowering memory.high"
```

Let's look at the commit that did something to the CIFS code (CIFS is the protocol used by Window's file sharing). To do this, supply the commit's SHA-1 hash to the **show** command:

```
~/src/linux$ git show a0f85e38a3c2
commit a0f85e38a3c23a334d9ea111cfa4fcfa4b3f9b64
Author: Ronnie Sahlberg <lsahlber@redhat.com>
Date:   Wed Feb 10 11:55:47 2021 +1000

    cifs: do not disable noperm if multiuser mount option is not provided

    Fixes small regression in implementation of new mount API.

    Signed-off-by: Ronnie Sahlberg <lsahlber@redhat.com>
    Reported-by: Hyunchul Lee <hyc.lee@gmail.com>
    Tested-by: Hyunchul Lee <hyc.lee@gmail.com>
    Signed-off-by: Steve French <stfrench@microsoft.com>

diff --git a/fs/cifs/fs_context.c b/fs/cifs/fs_context.c
index 5111aadfdb6b..1b1c56e52395 100644
--- a/fs/cifs/fs_context.c
+++ b/fs/cifs/fs_context.c
@@ -1533,8 +1533,8 @@ void smb3_update_mnt_flags(struct cifs_sb_info *cifs_sb)
                cifs_sb->mnt_cifs_flags |= (CIFS_MOUNT_MULTIUSER |
                                            CIFS_MOUNT_NO_PERM);
        else
-               cifs_sb->mnt_cifs_flags &= ~(CIFS_MOUNT_MULTIUSER |
-                                            CIFS_MOUNT_NO_PERM);
+               cifs_sb->mnt_cifs_flags &= ~CIFS_MOUNT_MULTIUSER;
+
        if (ctx->strict_io)
                cifs_sb->mnt_cifs_flags |= CIFS_MOUNT_STRICT_IO;
```

So, we are shown the author, the date the change was made, the short message, the long commit message, and a **diff** that shows what was removed by this commit as well as what was added. Well, that's pretty awesome. As you can see **git** makes managing changes to a large project very easy.

Maintainers don't actually write much code. They do this sort of stuff with git – all the time. Everyday. That's their job. They add changes (like this one) submitted by other people, sometimes they remove changes (or revert them) – git makes managing all these changes much easier.

## 5. Making your own commit

Alright, now it's time to make your own commit in your **base** branch of the kernel code.

You are in the **base** branch, the **base** branch is clean, and we know that your **base** branch is currently on the same commit as tag v5.11. Awesome. We're ready to start!

Before you can make your own commit, you will need to modify some code! So let's get started on making some changes!

## 6. Adding a system call to the kernel

In Lecture 9 (Week 5), we dived pretty deep into how system calls are implemented by Linux for the 64-bit Intel x86 architecture. So, watch that lecture video again if you need a refresher before jumping into making code changes.

Ready? Alright, your goal is to add a 64-bit Intel x86 system call to the kernel.

Your system call should:

- Be named **`bitmerge`**
- Accept two 32-bit parameters called **`high_bits`** and **`low_bits`**.
- Return a 64-bit value whose upper 32 bits are equal to **`high_bits`** and whose lower 32 bits are equal to **`low_bits`**.
- If the most significant bit of the resulting merged parameters is set, the system call should instead return an error code signifying that an invalid value was passed to it.

You will also need to write some non-kernel code (i.e. normal userspace code) that tests that your system call works correctly. In lecture, I did this in assembly so that you could see how the architecture works. Fortunately, there is a C function for calling arbitrary system calls name **`syscall()`**, so you can write your test program in C. (see **`man 2 syscall`**)

Name your userspace test program **`bitmergetest.c`** – this file will not live in the kernel tree. This is just a program that will call your system call to make sure it is working correctly. It should test passing both valid and invalid values to the **bitmerge** system call to make sure everything is working properly. Don't forget, in Homework 5 we learned how error codes are passed to userspace by system calls. (*Hint: You'll need to check* **`errno`***).*

Focus on this for a bit until you get everything in working order. When you're finished, move to the next section where we'll generate a kernel patch that adds the system call you just wrote. For this assignment, you'll be submitting this patch file and your **`bitmergetest.c`** file.

### 7. Review your code changes

Got it working? Excellent! Hopefully you didn't need to rebuild the kernel *too many* times. Waiting for code to compile can be tiresome. So, let's look at something a bit more exciting – generating a kernel patch!

Before you move forward, take a minute to review the changes you have made to the kernel source code. You can see the files you have changed by using **git status**, like we mentioned previously:

```
~/src/linux$ git status
```

You're patches should be surgical. Precise. Like a ninja surgeon. Your patch shouldn't make changes to files that don't need to be modified in order to implement the system call. If you did this right, **git status** should only show that 3 files have been modified.

You can see the changes that you made to a specific file by using **git diff**. So, take a moment to look at the files that were shown as modified when you ran **git status** using **git diff**. Just supply the file name to the command:

```
~/src/linux$ git diff arch/x86/somefile.c
```

This is incredibly important. Review what **git diff** displays for each file you modified. Did you add any unnecessary lines? Did you add some temporary comments that are still there? Do you have blocks of code that are commented out? If you do, open the file up in your text editor and clean up all that stuff. That doesn't need to be included in your patch. Only essential changes.

Did you modify a file that you shouldn't have modified at all? You can completely undo all changes you have made to a file by simply "checking out" the version of that file from the most recent commit. Be careful doing this since it will undo any changes you have made to the file:

```
~/src/linux$ git checkout some_accidentally_edited_file.c
```

Take a few minutes now to clean up your code. Once you are finished, you'll be ready to move on to the next step!

**8. Committing your changes**

Alright! All cleaned up and ready to go? Excellent. Let's commit your changes to the repository!

There are two phases to this: (1) selecting which modified files you want to commit and (2) actually committing them.

This might seem unnecessary at first, but this is actually quite useful if you don't want to commit all of the files that you have modified all at once as a single giant commit.

The **staging process** is where you select which which files git should include in the commit you are about to make. Once you have selected all the files to include in the commit by staging them, you then perform the actual commit.

We are going to keep it simple today. We are going to put all of our changes in one single commit. The general rule is "one logical change" = "one commit". Luckily for us, all we did was one logical change. We added a system call. So, that's one commit.

Let's stage (i.e. select) all our files:

| | |
|---|---|
| `~/src/linux$ git add file1.c` | (replace `file1.c` with actual file name) |
| `~/src/linux$ git status` | (notice how the file is now staged) |
| `~/src/linux$ git add file2.c` | (replace `file2.c` with actual file name) |
| `~/src/linux$ git add file3.c` | (replace `file3.c` with actual file name) |
| `~/src/linux$ git status` | (notice how all your modified files are now staged) |

You are now ready to make your first commit! Let's do it!

```
~/src/linux$ git commit --signoff
```

A text editor (probably vim) will open up automatically. This is where you type your commit message. Most people do this wrong. Very very wrong. Let's see how to do it correctly.

**9. Writing your commit message**

Again, most people do this very very wrong. Let's learn how to do this correctly!

Line 1 of the commit message is very important. Remember when we ran:

```
~/src/linux$ git log --oneline
```

earlier in Section 3? Remember how it only shows the SHA-1 hash of the commit and a short description of what the commit does?

Line 1 of your commit message defines the short description. Anything not on Line 1 is not part of the short description. **The short description should not be longer than 50 characters.**

If you look through the short descriptions used in the kernel's git repository, you'll notice that they all have the same format (ignore the Merge commits, they are special). In fact, it looks like all of these short descriptions could have all been written by the same person. Let's keep it that way. Organized and uniform.

A properly formatted short message looks like this:
**`subsystem: imperative description of what was done`**

Here are a few examples pulled from the actual log:
```
net: dsa: felix: implement port flushing on .phylink_mac_link_down
drm/i915/tgl+: Make sure TypeC FIA is powered up when initializing it
cifs: fix dfs-links
x86/build: Disable CET instrumentation in the kernel for 32-bit too
scsi: scsi_debug: Fix a memory leak
```

You get the idea. This makes it very easy to find commits that touch a specific part of the kernel. A good short description for your commit would be something like this:
**`x86/entry/64: add bitmerge system call`**

Line 2 should be empty. Your long description starts on Line 3.

You can write anything you want in the long description that helps to clarify the work that was actually done. State what your changes do, maybe give an example of how the system call should be called, describe how the system call behaves. It would be helpful to mention what system call number was chosen. You get the idea. Write a good description here. Don't put more than 72 characters on a line. Be sure to leave the **Signed-off-by** message at the bottom. When you are done, save and exit (**`:wq`** in vim).

### 10. Look at your fancy new commit

Now you should be back at the terminal. Let's see the changes between version tag **v5.11** and our **base** branch.

```
~/src/linux$ git log --oneline v5.11..base
~/src/linux$ git log v5.11..base
```

That's your commit! Congratulations!

If you look at **git status**, you'll see that you don't have any modified files or files staged for commit. Your changes are now in the repository!

### 11. Create a patch file from your commit

Linux development is done using a mailing list. Patch files and series of patch files get sent to the mailing list when somebody wants them to be considered to be included in the kernel. The patches get reviewed and then they are accepted, rejected, or discussed/revised. That is what you do. You send a patches by e-mail. You don't send Linus a pull request. There are no pull requests. There are patch files and there is e-mail.

So, let's create a patch file from our commit! This is very easy to do:

```
~/src/linux$ git format-patch v5.11..base
```

That's it. You did it!

If you look in your current directory, you'll find a file name something like:

```
0001-x86-entry-64-add-bitmerge-system-call.patch
```

This is your patch file. You *could* send this to the kernel mailing list, but instead you're going to send it to me!

But before you do, let's make sure is nothing wrong with your coding style.

### 12. Run checkpatch.pl on your patch

So far you have only been running **checkpatch.pl** on individual source files by passing the **--file** argument and the name of an individual C source file. Now you are going to run it on your entire patch file, which describes changes to multiple C files. Don't worry, this is easy!

```
~/src/linux$ ./scripts/checkpatch.pl 0001-name-of-your-file.patch
```

If you didn't make any code style mistakes, you should get some output that looks like this:

```
total: 0 errors, 0 warnings, 31 lines checked

0001-x86-entry-64-add-bitmerge-system-call.patch has no obvious style problems and
is ready for submission.
```

That's it!

If you have 0 errors and 0 warnings, then there is nothing wrong with your patch!

You are done! Again, Congratulations!

All that's left to do is upload your patch file and **bitmergetest.c** file to BBLearn!

Good Job! Excellent work.

### 13. You had checkpatch.pl errors or warnings

Wait, you aren't done? **checkpatch.pl** gave you some errors and warnings? Not a big deal. This is pretty easy to fix.

First, edit each of the files that you received an error or warning for and fix the problems. Once you are finished, simply **amend** your existing commit to include these fixes:

```
~/src/linux$ git commit --amend -a
```

(the **-a** flag automatically stages all modified files, isn't that convenient!)

Your old commit message will pop up. You can choose to modify your commit message, or if you are still happy with it, you can just save and exit.

Congratulations! You fixed the issues with your commit!

Now you can regenerate your patch file:

```
~/src/linux$ git format-patch v5.11..base
```

Your patch file now includes all the fixes you just made! If you examine the patch file in your text editor, you'll see that there is no record of your previous mistakes! Hopefully, it passes the **checkpatch.pl** test now. You'll probably want to check again, just to make sure. If the patch still has issues, just repeat this process until you fix them all. You can make amendments to a commit as many times as you would like.

### 14. Your git state is a disaster?

Your git history got out of control? You have multiple commits when you should only have one? You want to undo all your git mistakes but you don't want to lose your code?

This is easy to fix. Don't panic.

If you made a few git mistakes but don't want to lose all your hard work, you can simply **reset** your **base** branch to the **v5.11** tag:

```
~/src/linux$ git reset v5.11
```

This is called a **soft reset**. All the files you modified still contain your changes. You can see this if you now run **git status**. However, now your **base** branch will not contain any of the commits you created.

So, basically, this rewinds time back to Step 8 in this document.

If you find yourself in trouble with git and can't figure out what went wrong, just reset your branch to tag **v5.11** and go back to Step 8 of this document and give it another try. I'm sure you'll get it right this time now that you have some experience under your belt. We learn from our mistakes, and everybody who is a git master has made their share to get to where they are today.