1. Documentation of the Linux Kernel Programming Project

Authors:
1. Krzysztof Kostrzewa, 380029
2. Rico Stanosek, 433500

In this document we first describe how we solved the project requirements and later (Section 4) explain other changes we made to the project (compared (`diff`ed) to the starting point: `f141d52`).

1.1. Helper functions for eviction policies

In `eviction_policy/eviction_policy.(h|c)` you will find helper functions that are used in the implementations of the eviction policies. These functions are:

1. A default eviction policy that serves as a fallback, if no other eviction policy is specified. It does nothing.
2. `register_eviction_policy` and `unregister_eviction_policy` to register and unregister eviction policies.
   - it uses a linked list (`list_head`) to keep track of the registered eviction policies. That's also partially the reason, why we need a default policy, to make the list handling easier (less null checks).
3. `set_eviction_policy` to set which eviction policy is to be used.
4. `traverse_dir` is used to recursively traverse directory. It is very flexible, which allows us to use it in different eviction policies by adjusting the functionality that is passed to it.
   - because the filesystem is a tree, we have used standard recursive approach to check all files. We know recursion isn't great in the kernel and this could probably be done without it, but ease of reasoning and readability were more important in those circumstances (Uni project).
5. `ouichefs_remove_file` is our implementation of deleting a file based on whether we have the `dentry` available or not.
   - we are traversing the "disk" so some files might not have been loaded into dentry cache, because the user hasn't accessed them yet, in that case they have only `inode` and `dentry` doesn't exist.
6. `ouichefs_file_in_use` is used to check whether a file is used.
   - Originally, we iterated over every process, grabbed their open files and checked their inode against our inode. This is highly inefficient, but we didn't find a better way to solve this issue until we joined the last Q&A session and learned about the `i_readcount` and `i_writecount` fields of `struct inode`. We now check, whether one of these fields are non-zero, which yields the result we want. You can still check out our old "implementation" under `ouichefs_file_in_use_legacy`.

1.2. Eviction policies

In both eviction policies you will find the following functions:

1. `clean_dir` gets called when a directory is full, and we need to free up space as another file is to be created.
2. `clean_partition` gets called when the partition exceeds a given threshold.
3. `leaf_action` performs some action on a file (leaf). Depending on the eviction policy it compares whether the file is to be deleted or not.
4. A data structure to track which file is to be deleted

1.2.1. Least Recently Used (LRU)

In `wich_lru.c` you will find the implementation of the LRU eviction policy. The ease of just modifying the `is_older` function to depend on either `i_atime`, `i_mtime` or `i_ctime` makes it very flexible and enabled us to implement three eviction strategies with minimal effort.

The default for LRU eviction policy is to use `i_ctime`. You can dynamically change the policy when inserting the module.

```
insmod wich_lru.ko mode=1 # i_atime
insmod wich_lru.ko mode=2 # i_mtime
insmod wich_lru.ko mode=3 # i_ctime, default
```

`clean_partition` collects basic information about the files in the partition and starts the traversal of the partition to find the file to be deleted. It uses `traverse_dir` to traverse the directories and to pass the information to `leaf_action` of the current eviction policy to determine whether the current node is e.g. the least-recently used.

Based on the inode that gets returned, we lastly perform `null` checks and prevent the deletion of the root directory. If these checks pass, we call `ouichefs_remove_file` to delete the file.

In `clean_dir` we can iterate over every file in the given directory and compare them using `is_older` to find the file to delete. We run into an error if the current folder only contains directories.

If we find a file to delete, we call `ouichefs_remove_file` to delete the file.

1.2.2. Size-based eviction

Most of the parts of this eviction policy are similar to the LRU eviction policy. As we do not want to repeat ourselves, we want to highlight the differences.

In `wich_size.c` you will find the implementation of the size-based eviction policy.

Instead of comparing dates in `leaf_action`, we compare the sizes of files to find the largest one.

`clean_partition` works exactly the same as in the LRU eviction policy, except we pass a different function to `traverse_dir` to compare the sizes of the files and another data structure to keep track of the largest file. Afterwards the same checks are performed to prevent the deletion of the root directory and to delete the file.

Again, `clean_dir` works exactly the same as in the LRU eviction policy, except we search for the biggest file instead of the oldest before deleting.

1.3. Printing policy

You will also find an eviction policy `wich_print.c` that will not evict anything. This eviction policy is used to print file and inode information that were very handy during the development of the eviction policies in a tree like fashion.

It gets triggered like every other policy.

1.4. Automatic eviction

The two scenarios in which the eviction process is triggered are:

1. When a directory is full, and we need to free up space as another file is to be created. You can see our implementation in `ouichefs_create` in `inode.c`.
2. When the partition exceeds a given threshold, which can be configured when inserting the `ouichefs` module (see Section 1.5). The triggering of the eviction process is implemented in `ouichefs_write_end` in `file.c`.

1.5. Base setup

To compile our project you should be able to use the `Makefile` in the root directory. `make install` will additionally move the `ko` files and our `scripts` directory to the specified path in the virtual machine.

```
make
make install
```

To use the eviction policies mechanism, you need to insert the `ouichefs` module and your preferred policy module(s) (You can configure which inserted module is active, see Section 1.6). Of course, you also have to have a partition using ouichefs mounted.

```
insmod ouichefs.ko trigger_threshold=50
./scripts/mount.sh test.img
```

You can set the trigger threshold to a value of your choice. The default is 20 %, meaning that if 80 % of the partitions blocks are used, the eviction process is triggered.

Afterwards you can insert the eviction policies of your choice.

```
insmod wich_size.ko
insmod wich_lru.ko
```

1.6. Changing eviction policies

You can retrieve the available eviction policies by using the following command:

```
cat /proc/ouiche/eviction
```

```
Following eviction policies are available:
default (does nothing)
wich_size
wich_lru        [ACTIVE]
```

By default, the last inserted policy is active. You can change the active policy with:

```
echo -n "wich_size" > /proc/ouiche/eviction
```

```
ouichefs:evictions_proc_write: Received policy name: wich_size
set eviction policy to 'wich_size'
```

1.7. Triggering manual eviction

To manually trigger the eviction process, you need the target partition. You can retrieve the available partitions of ouichefs with:

```
cat /proc/ouiche/partitions
```

The will output the mountpoints and their indices (in previous version we wanted to use the name not the index but x86_64 Linux didn't want to cooperate)

```
Following partitions use ouiche_fs:
INDEX   NAME
0       0:/dev/loop1
```

You can then manually trigger the eviction process with:

```
echo -n 0 > /proc/ouiche/clean
```

where `0` is the index of the partition you want to clean.

1.8. Removing the modules

You can remove the module with:

```
./scripts/umount.sh test.img
rmmod wich_size.ko
rmmod wich_lru.ko
rmmod ouichefs.ko
```

2. List of features implemented but not fully functional/not implemented

According to the requirements, we have implemented every feature that was required. We performed extensive testing and are confident that our implementation is functional.

3. List of bugs

We have found two possible bugs and fixed one of them.

3.1. Bug 1: Minimal size of the image

In `mkfs/mkfs-ouichefs.c` the check for the minimal size used $\leq$ instead of $<$.

```
-if (stat_buf.st_size <= min_size) {
+if (stat_buf.st_size < min_size) {
```

3.2. Bug 2: `.` and `..` missing in `ls`

We have found that `.` and `..` are missing in the output of `ls`. But we have not found a solution for this issue.

---

4. Other changes to the project

You can see the whole `diff` here. (We've invited you to the private repo, but if you can't access it, drop us a line)

4.1.1. `.vscode`

The configuration needed for the IntelliSense and spellchecker in VS Code.

4.1.2. `Makefile`

We had to modify the `Makefile` to compile our `eviction_policy.(c|h)`, we have also added convenience option `install` to move the `ko` files to the vm.

But the biggest change was adding `LLVM` flag, which allowed building the project with `clang` under macOS. It acts the same as the option form Linux kernels Makefile.

4.1.3. `mkfs/Makefile`

Here we've two more sizes of the image to make testing easier (filling one MB is easier than 50). Here the `smol_img` is the smallest image possible with this mkfs as mentioned in Section 3.1.

4.1.4. `mac_kernel_patch_6-5-7.patch`

Patch of the kernel to make it buildable with `clang` on macOS.

Steps:

1. add llvm to path: `PATH="/opt/homebrew/Cellar/llvm/<current-version>/bin/:$PATH"`
2. make config: `make LLVM=1 ARCH=arm64 menuconfig`
3. build the kernel `make LLVM=1 ARCH=arm64 -j 8 HOSTCFLAGS="-I./"`
   - we need `HOSTCFLAGS="-I./"` to tell the compiler where to look for `*.h` files added by the patch

Discalimer!

We mention the patch here, because (1) it's interesting and (2) it's what Krzysztof used to develop. **BUT** we have tested the project on Arch Linux/arm64 and elementaryOS/x86_64 and both worked.