Operating Systems

Term Paper

- Final Report–

December 10, 2019

Group 7

Park Minju

Hatice İrem Gökçek

Introduction

As a group we decided to analyze and understand basic file system in Linux kernel. More specifically, we chose the caching system in linux-5.3.1/fs/fscache/cache.c.

We started our project by examining the source code, which is around 350-400 lines excluding the comments. We examined the included files as well but we decided to not work on them. While analyzing the module, we referred to our course book Operating Systems Concepts by Abraham Silberschatz, Chapters 11: File System Interface and Chapter 12: File System Implementation.

The purpose of this module is to improve the performance of the network and the media. According to Howells (who implemented the entire module), FS-Cache is for any system that can cache data locally. Caching is an important topic because it improves the server and the network because it reduces the need to go to the network. On Figure 1, we observe that the FS-Cache is a layer between client file systems and caching services.

As the role of each team member, we mostly worked as a group by meeting regularly and working efficiently. For the code examination part, there were 8 functions, we split them into 2 parts so each member could have approximately same amount of work. For the improvement part, we decided together which functions to be improved and how. The implementation was mostly done by Minju. The diagram of the entire module and the final documentation was mostly done by İrem. Overall, we worked well as a team.

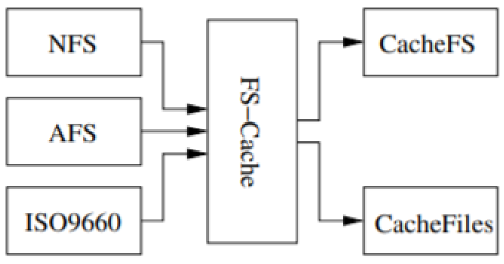


Figure 1: Cache architecture

Diagram of Cache.c

In the Diagram 1 of page 3, we can see how the caching system works in file system of linux-5.3.1. A cache can be initialized and then added into to the cache list. When a cache is added as "declared open for business", first it checks if there is an error, then if the cache tag is used. If it's either case, the cache tag is released. Otherwise, it looks for the cache tag and assigns it to the new cache. When a cache is withdrawn, all the objects within that cache must be withdrawn as well.

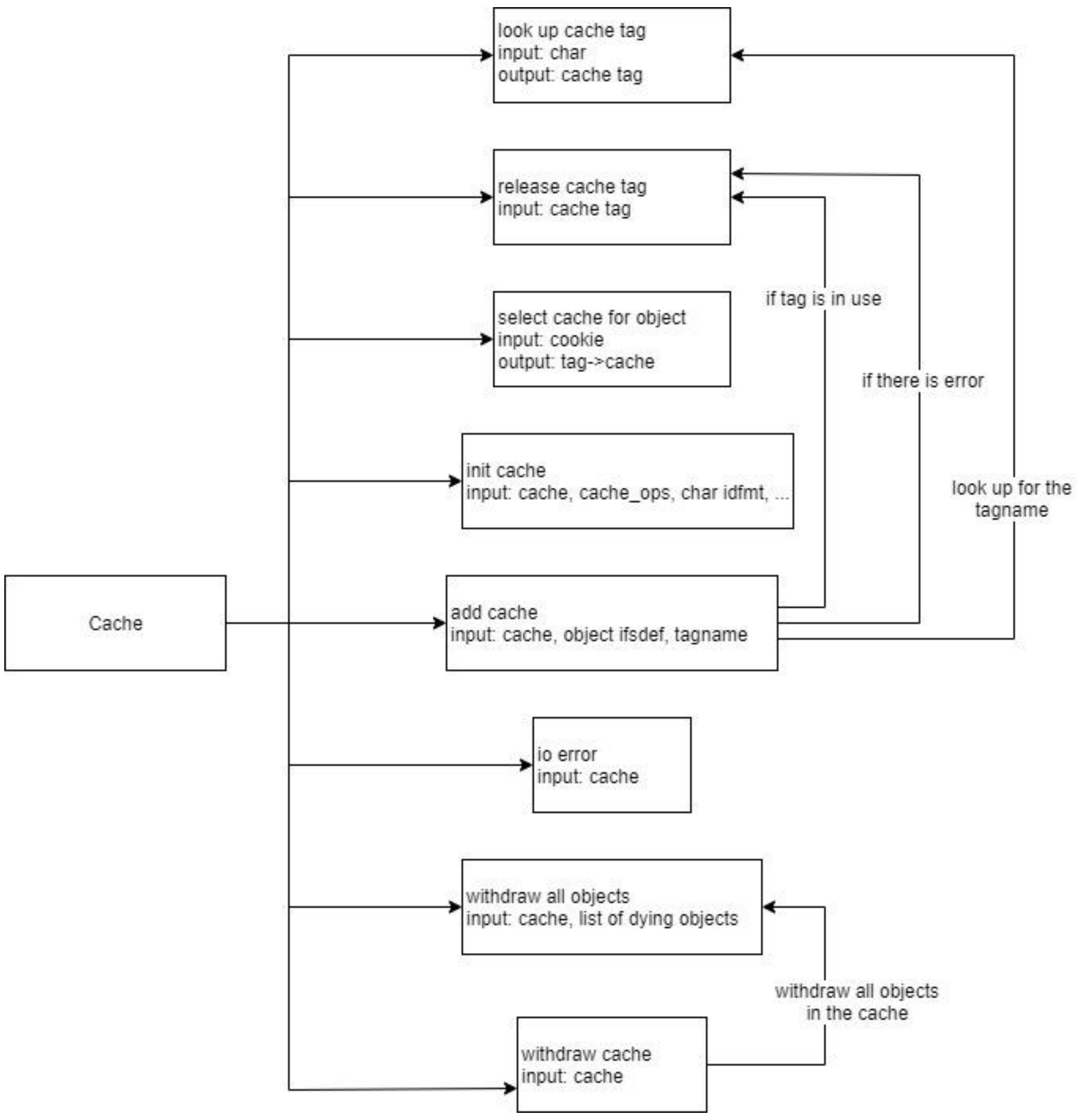


Diagram 1: The Entire Module

File System

In this report, we have analyzed the file system in Linux kernel, specifically caching. The module that we have previously chosen was linux-5.3.1/fs/fscache/cache.c. In this file, there are 3 included files <linux/module.h>, <linux/slab.h> and “internal.h” which contains debug tracing, assertions and cookie.c. And then a list of cache, a list of cache tag, an fscache_add_remove semaphore and 8 functions are declared.

Here is the detailed explanation of these functions that are in this file:

Function 1:

parameter: string (tag name)

structfscache_cache_tag *__fscache_lookup_cache_tag(cont char *name)

This structure is to look up a cache tag

* If there is a cache tag which name is same with the input parameter, it increases tag->usage by 1 and return the tag list. However, if there isn’t, it makes a new tag(xtag) with its usage=1 and links the tag to the tag list. It performs by using the semaphore (fscache_addremove_sem) which locks the data used by the function process.

Function 2:

parameter: tag (cache tag pointer)

void __fscache_release_cache_tag(structfscache_cache_tag *tag)

* This function is to release a reference to a cache tag i.e. to remove the tag. During the process use the data, it is locked by using the semaphore(fscache_addremove_sem) and then the tag is de-allocated.

Function 3:

parameter: cookie

structfscache_cache *fscache_select_cache_for_object(structfscache_cookie *cookie)

* The function is to select a cache store an object. Selecting a cache is done by the function select_cache which is in fscache_cookie(structure)->def. If any cache cannot be selected, the function returns the first node of the cache list. The data used are locked by using spin_lock.

Function 4:

parameter: pointers - cache, ops / string – idfmt (id format)

voidfscache_init_cache(structfscache_cache *cache, conststructfscache_cache_ops * ops, constchar *idfmt, …)

* This function is to initialize a cache record. The parameter ops becomes the cache operation(cache->ops) and idfmt is to be the format string becomes the cache identifier. The list structures of the cache and spinlocks are initialized by INIT_LIST_HEAD or spin_lock_init functions.

Function 5:

parameter: pointers - cache, ifsdef / string – tagname

intfscache_add_cache(structfscache_cache *cache, structfscache_object *ifsdef, const char *tagname)

This structure is to add a cache

* This function is to declare a cache as being open for business. The parameter cache is the record describing the cache, the parameter ifsdef is the record of the cache object describing the index, the parameter tagname is describing the cache. It adds a cache to the system. By this, the cache is available for netfs to use.

Function 6:

parameter: pointers - cache

<u>voidfscache_io_error(structfscache_cache *cache)</u>

* This function is for when an I/O error occurs, it makes sure that the cache will not be used for anything later. The error is reported into the kernel log. Parameter cache is the record describing the cache with the I/O error.

Function 7:

parameter: pointers - cache, dying_objects list

<u>static void fscache_withdraw_all_objects(structfscache_cache *cache, structlist_head *dying_objects)</u>

* This function withdraws all the objects in a cache. The objects that are being withdrawn are moved onto the supplied list. The list gets locked, then while loop is used until the whole list is emptied and then it raises an fscache raise event for the dead object. When the loop ends, the cache is unlocked.

Function 8:

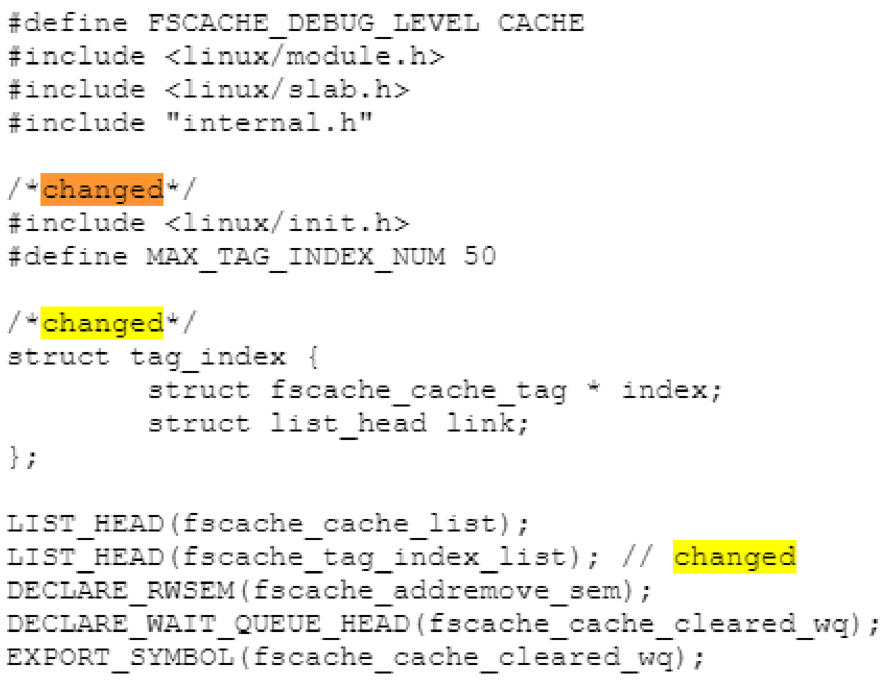
parameter: pointers - cache

<u>voidfscache_withdraw_cache(structfscache_cache *cache)</u>

* This function withdraw a cache from the active service. It unbinds all its cache objetcs from the netfs cookies. The parameter cache is the record describing the cache. First, it makes the cache unavailable for cookie acquisition and destroys all the active objects of this cache. Then, it uses the above function, fscache_withdraw_all_objects to remove all the objects inside. When the job is finished, it releases the cache tag of the withdrawn cache.

Improvement of cache.c

We started with changing the first parts of the file. We included the linux library linux/init.h and defined a maximum number for the tag index which is 50. Next change is, we declared our own tag_index structure. It has a pointer object of type fscache_cache_tag and a link of type list_head. Then, we added our own list of fscache tag indexes.



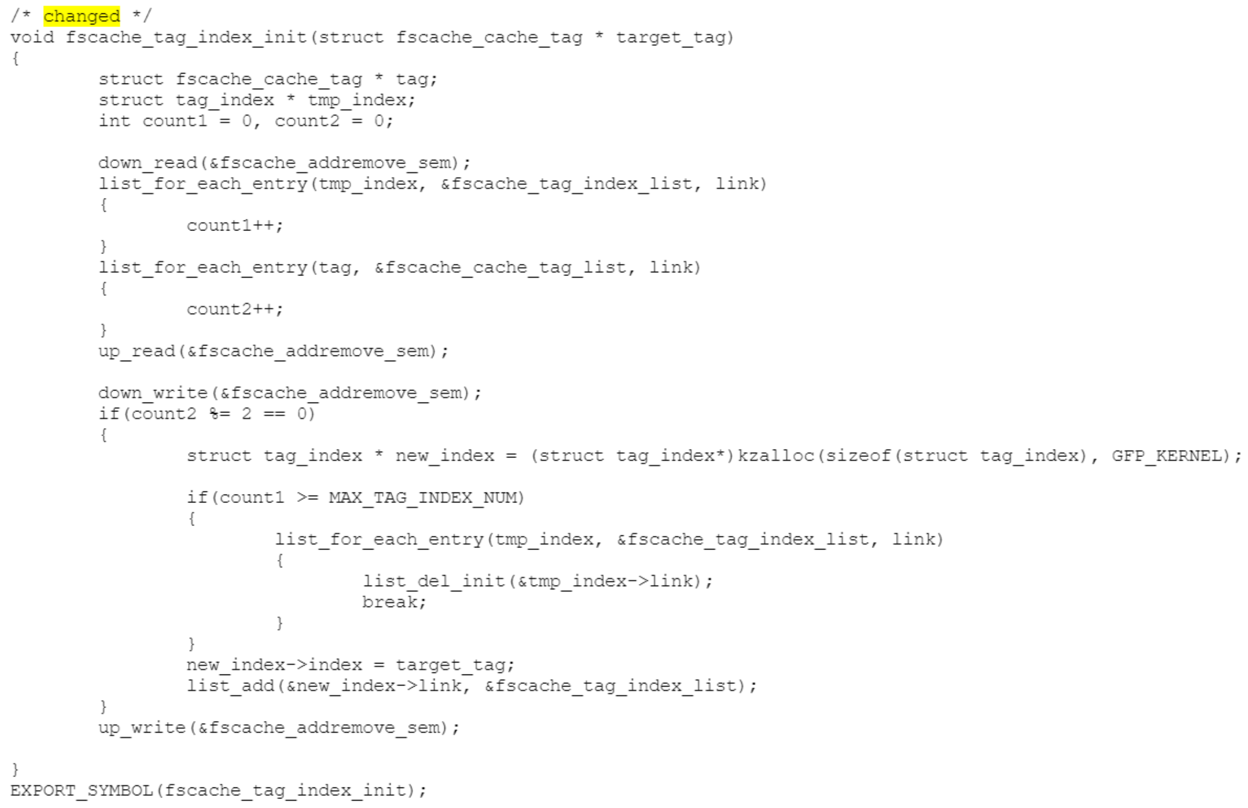
```
#define FSCACHE_DEBUG_LEVEL CACHE
#include <linux/module.h>
#include <linux/slab.h>
#include "internal.h"

/*changed*/
#include <linux/init.h>
#define MAX_TAG_INDEX_NUM 50

/*changed*/
struct tag_index {
        struct fscache_cache_tag * index;
        struct list_head link;
};

LIST_HEAD(fscache_cache_list);
LIST_HEAD(fscache_tag_index_list); // changed
DECLARE_RWSEM(fscache_addremove_sem);
DECLARE_WAIT_QUEUE_HEAD(fscache_cache_cleared_wq);
EXPORT_SYMBOL(fscache_cache_cleared_wq);
```

Figure 2: The code part 1

Next, we added 2 functions: fscache_tag_index_init(fscache_cache_tag* target_tag) and fscache_tag_index_release(fscache_cache_tag* target_tag).

Fscache_tag_index_init initializes a cache tag. Then, gets the count of the elements in fscache tag index list and fscache cache tag list and stores them as count1 and count2.

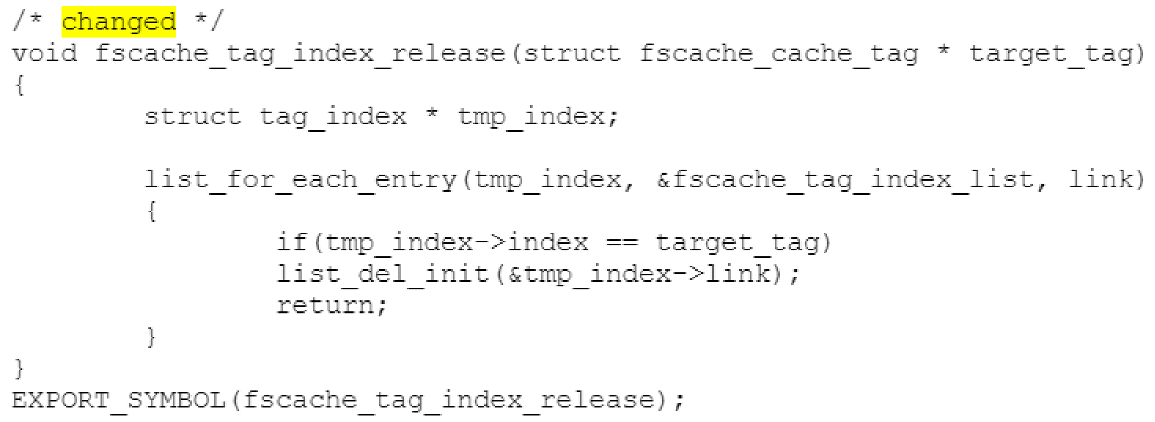
If count2 is even, a new tag index is allocated. If count of the fscache tag index list (which is count1) is equal or more than the max number of tag indexes (which is 50), the first tag index in the fscache tag index list is removed. This way we make sure that the tag index list doesn't get too large and it is maintained easily. We set the tag index's index as the tag and added the new index to the tag index list.



```
/* changed */
void fscache_tag_index_init(struct fscache_cache_tag * target_tag)
{
        struct fscache_cache_tag * tag;
        struct tag_index * tmp_index;
        int count1 = 0, count2 = 0;

        down_read(&fscache_addremove_sem);
        list_for_each_entry(tmp_index, &fscache_tag_index_list, link)
        {
                count1++;
        }
        list_for_each_entry(tag, &fscache_cache_tag_list, link)
        {
                count2++;
        }
        up_read(&fscache_addremove_sem);

        down_write(&fscache_addremove_sem);
        if(count2 %= 2 == 0)
        {
                struct tag_index * new_index = (struct tag_index*)kzalloc(sizeof(struct tag_index), GFP_KERNEL);

                if(count1 >= MAX_TAG_INDEX_NUM)
                {
                        list_for_each_entry(tmp_index, &fscache_tag_index_list, link)
                        {
                                list_del_init(&tmp_index->link);
                                break;
                        }
                }
                new_index->index = target_tag;
                list_add(&new_index->link, &fscache_tag_index_list);
        }
        up_write(&fscache_addremove_sem);

}
EXPORT_SYMBOL(fscache_tag_index_init);
```

Figure 3: The code part 2

The second added function is fscache tag index release. It creates a temporary tag index pointer. For each entry in the fscache tag index list, we check if the temporary tag index’s index is equal to the targeted tag’s index, then it is deleted from the list. This is how we decided to relase a fscache cache tag.



```
/* changed */
void fscache_tag_index_release(struct fscache_cache_tag * target_tag)
{
        struct tag_index * tmp_index;

        list_for_each_entry(tmp_index, &fscache_tag_index_list, link)
        {
                if(tmp_index->index == target_tag)
                list_del_init(&tmp_index->link);
                return;
        }
}
EXPORT_SYMBOL(fscache_tag_index_release);
```

Figure 4: The code part 3

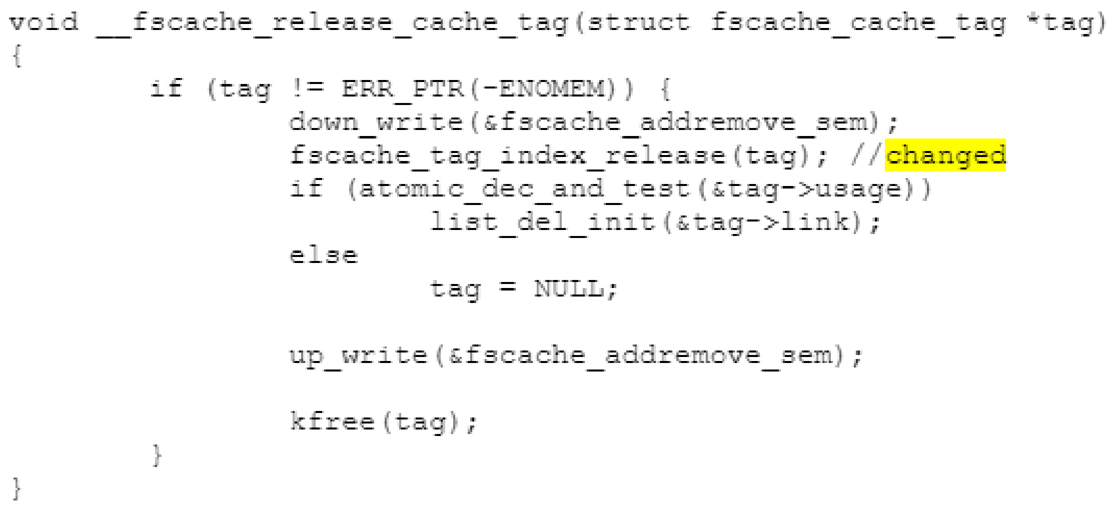
We used these 2 functions in __fscache_lookup_cache_tag and __fscache_release_cache_tag functions.

In __fscache_lookup_cache_tag, we modified it to use fscache_tag_index_init function that was implemented. We started by defining a tag_index object and then, we check every entry in the fscache_tag_index_list. If it is in the list, we return the index of the tag index. If not, we search through the fscache_cache_tag_list and find a matching tag with the name that we are looking. It works as a cache for cache tag. Then, we call our implemented function as fscache_tag_index_init(tag). The rest is the original cache.c.

```
struct fscache_cache_tag *__fscache_lookup_cache_tag(const char *name)
{
        struct fscache_cache_tag *tag, *xtag;
        struct tag_index *index; //changed

        /* firstly check for the existence of the tag under read lock */
        down_read(&fscache_addremove_sem);

        /* changed */
        list_for_each_entry(index, &fscache_tag_index_list, link)
        {
                if (strcmp(index->index->name,name) == 0) {
                        atomic_inc(&index->index->usage);
                        up_read(&fscache_addremove_sem);
                        return index->index;
                }
        }


        list_for_each_entry(tag, &fscache_cache_tag_list, link) {
                if (strcmp(tag->name, name) == 0) {
                        atomic_inc(&tag->usage);
                        fscache_tag_index_init(tag); //changed
                        up_read(&fscache_addremove_sem);
                        return tag;
                }
        }

        up_read(&fscache_addremove_sem);

        /* the tag does not exist - create a candidate */
        xtag = kzalloc(sizeof(*xtag) + strlen(name) + 1, GFP_KERNEL);
        if (!xtag)
                /* return a dummy tag if out of memory */
                return ERR_PTR(-ENOMEM);

        atomic_set(&xtag->usage, 1);
        strcpy(xtag->name, name);

        /* write lock, search again and add if still not present */
        down_write(&fscache_addremove_sem);

        list_for_each_entry(tag, &fscache_cache_tag_list, link) {
                if (strcmp(tag->name, name) == 0) {
                        atomic_inc(&tag->usage);
                        up_write(&fscache_addremove_sem);
                        kfree(xtag);
                        return tag;
                }
        }

        list_add_tail(&xtag->link, &fscache_cache_tag_list);
        up_write(&fscache_addremove_sem);
        return xtag;
}
```

Figure 5: The code part 4

In __fscache_release_cache_tag, we modified our code such that if there is no error, we call fscache_tag_index_release(tag). The rest is the same as original cache.c. This way, we made sure that our newly implemented 2 functions can be executed when a cache is released or a cache is looked up.



```
void __fscache_release_cache_tag(struct fscache_cache_tag *tag)
{
	if (tag != ERR_PTR(-ENOMEM)) {
		down_write(&fscache_addremove_sem);
		fscache_tag_index_release(tag); //changed
		if (atomic_dec_and_test(&tag->usage))
			list_del_init(&tag->link);
		else
			tag = NULL;

		up_write(&fscache_addremove_sem);

		kfree(tag);
	}
}
```

Figure 6: The code part 5

The diagram of our new cache.c:

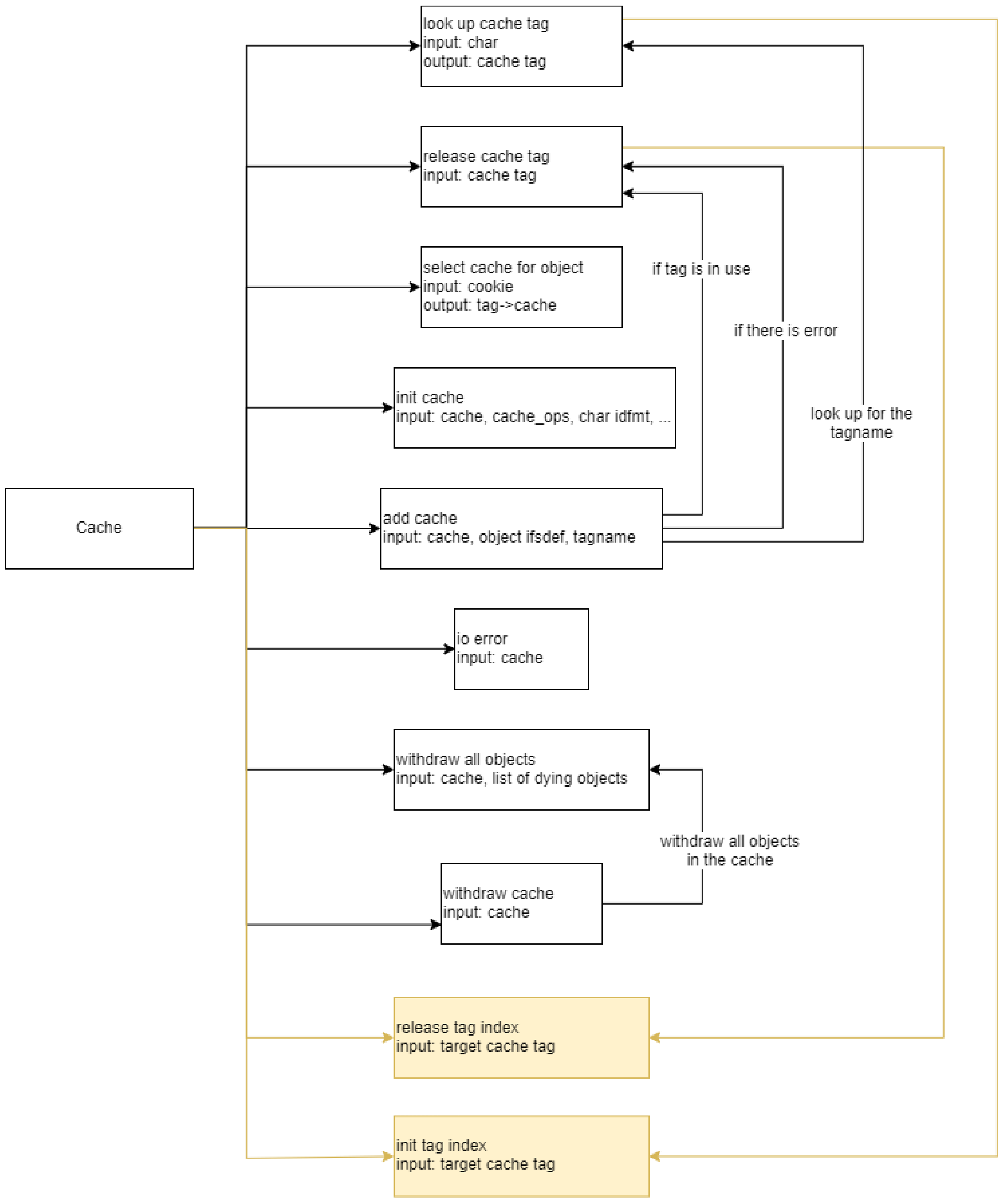
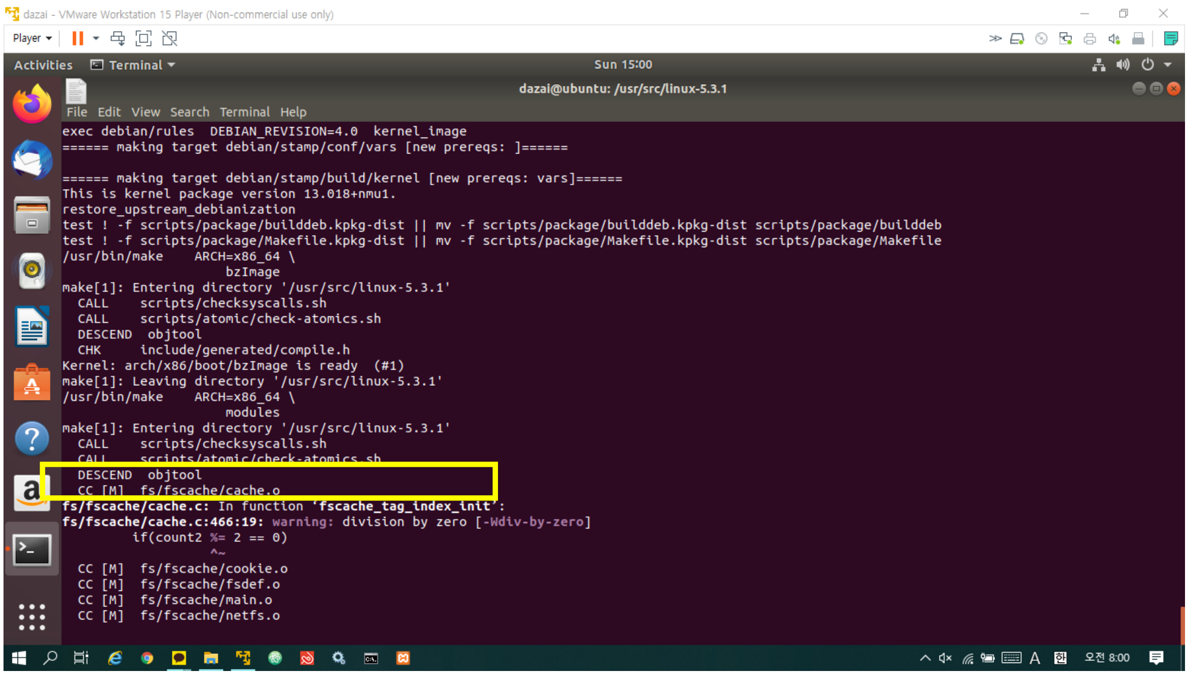


Diagram 2: The Modified Module

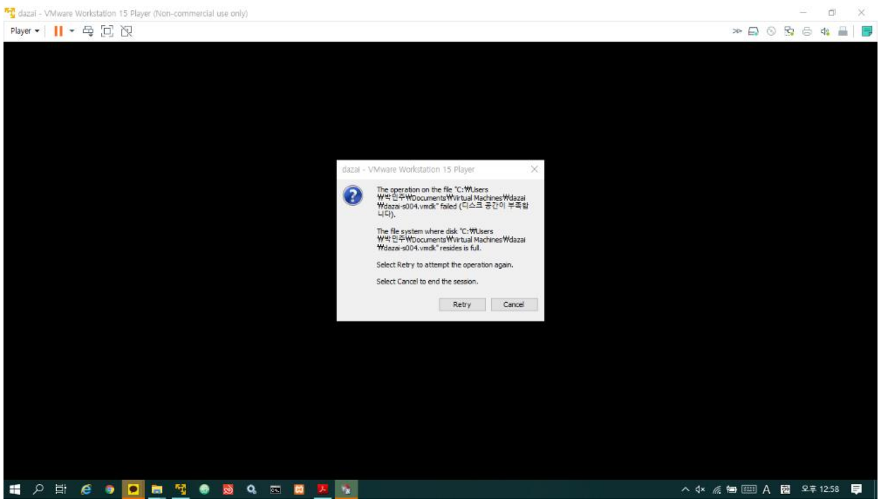
Comparison& Implementation



```
exec debian/rules  DEBIAN_REVISION=4.0  kernel_image
====== making target debian/stamp/conf/vars [new prereqs: ]======

====== making target debian/stamp/build/kernel [new prereqs: vars]======
This is kernel package version 13.018+nmu1.
restore_upstream_debianization
test ! -f scripts/package/builddeb.kpkg-dist || mv -f scripts/package/builddeb.kpkg-dist scripts/package/builddeb
test ! -f scripts/package/Makefile.kpkg-dist || mv -f scripts/package/Makefile.kpkg-dist scripts/package/Makefile
/usr/bin/make    ARCH=x86_64 \
                 bzImage
make[1]: Entering directory '/usr/src/linux-5.3.1'
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
  DESCEND  objtool
  CHK     include/generated/compile.h
Kernel: arch/x86/boot/bzImage is ready  (#1)
make[1]: Leaving directory '/usr/src/linux-5.3.1'
/usr/bin/make    ARCH=x86_64 \
                 modules
make[1]: Entering directory '/usr/src/linux-5.3.1'
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
  DESCEND  objtool
  CC [M]  fs/fscache/cache.o
fs/fscache/cache.c: In function 'fscache_tag_index_init':
fs/fscache/cache.c:466:19: warning: division by zero [-Wdiv-by-zero]
         if(count2 %= 2 == 0)
                   ^~

  CC [M]  fs/fscache/cookie.o
  CC [M]  fs/fscache/fsdef.o
  CC [M]  fs/fscache/main.o
  CC [M]  fs/fscache/netfs.o
```

Figure 7: Snapshot of cache.o

We added the modified cache.c (which is same with indexed_cache.c in .zip) into the kernel. We compiled it for several hours. For comparison, we hoped to say that our efficiency is better than the original cache.c. The 2 included functions are not time consuming and the tag lookup function can scan the index tag list first which works like cache of cache tag. In the screenshot above, we see the fs/fscache/cache.o file. Therefore, our compile works successfully.



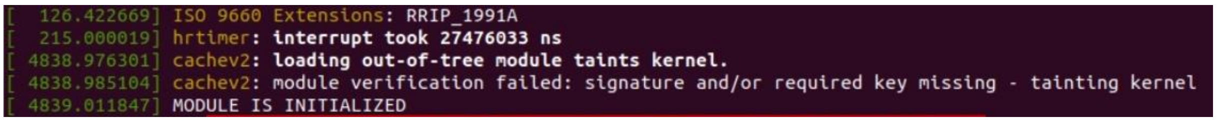
```
dazai - VMware Workstation 15 Player

The operation on the file "C:₩Users
₩박민주₩Documents₩Virtual Machines₩dazai
₩dazai-s004.vmdk" failed (디스크 공간이 부족합
니다).

The file system where disk "C:₩Users
₩박민주₩Documents₩Virtual Machines₩dazai
₩dazai-s004.vmdk" resides is full.

Select Retry to attempt the operation again.

Select Cancel to end the session.

Retry    Cancel
```

Figure 8: Snapshot of cache.o

However, the compilation had been going on for more than 24 hours. And finally, it died. The error said that there is no more space left.

We retrieved our code, the file in the submitted .zip is called indexed_cache.c.

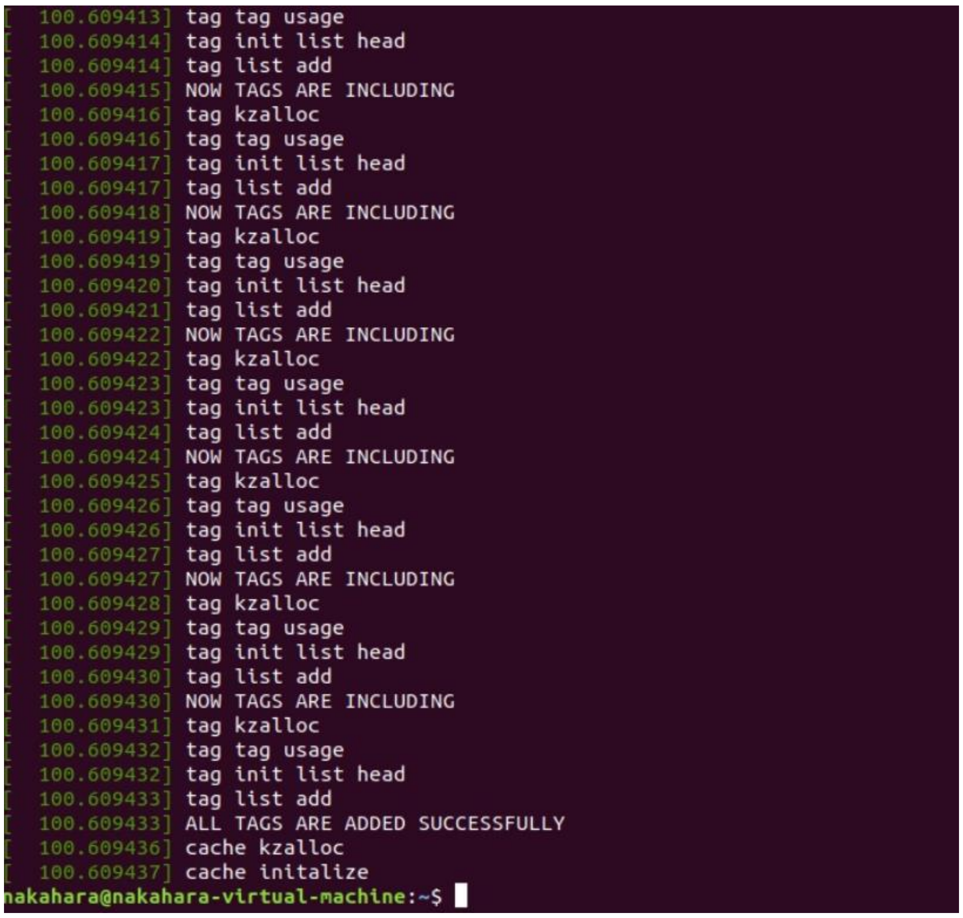
Another approach that we did was making a separate file just like cache.c and we compiled it. The file is called cachev2.c. Here are some screenshots:



```
[  126.422669] ISO 9660 Extensions: RRIP_1991A
[  215.000019] hrtimer: interrupt took 27476033 ns
[ 4838.976301] cachev2: loading out-of-tree module taints kernel.
[ 4838.985104] cachev2: module verification failed: signature and/or required key missing - tainting kernel
[ 4839.011847] MODULE IS INITIALIZED
```

Figure 9: Snapshot of cachev2.c working

This is the end of the initialization function. We can see that our code works and some mock tags and tag names are added into the lists using our own implemented functions.



```
[  100.609413] tag tag usage
[  100.609414] tag init list head
[  100.609414] tag list add
[  100.609415] NOW TAGS ARE INCLUDING
[  100.609416] tag kzalloc
[  100.609416] tag tag usage
[  100.609417] tag init list head
[  100.609417] tag list add
[  100.609418] NOW TAGS ARE INCLUDING
[  100.609419] tag kzalloc
[  100.609419] tag tag usage
[  100.609420] tag init list head
[  100.609421] tag list add
[  100.609422] NOW TAGS ARE INCLUDING
[  100.609422] tag kzalloc
[  100.609423] tag tag usage
[  100.609423] tag init list head
[  100.609424] tag list add
[  100.609424] NOW TAGS ARE INCLUDING
[  100.609425] tag kzalloc
[  100.609426] tag tag usage
[  100.609426] tag init list head
[  100.609427] tag list add
[  100.609427] NOW TAGS ARE INCLUDING
[  100.609428] tag kzalloc
[  100.609429] tag tag usage
[  100.609429] tag init list head
[  100.609430] tag list add
[  100.609430] NOW TAGS ARE INCLUDING
[  100.609431] tag kzalloc
[  100.609432] tag tag usage
[  100.609432] tag init list head
[  100.609433] tag list add
[  100.609433] ALL TAGS ARE ADDED SUCCESSFULLY
[  100.609436] cache kzalloc
[  100.609437] cache intialize
nakahara@nakahara-virtual-machine:~$ 
```

Figure 10: Snapshot of cachev2.c initializing

References:

[1]Howells, David. FS-Cache: A Network Filesystem Caching Facility, Red Hat UK Ltd https://people.redhat.com/dhowells/fscache/FS-Cache.pdf

[2] Silberschatz, Abraham. Operating Systems Concepts, 9th Edition, Wiley, 2013.

ISBN: 978-1-118-06333-0

[3] TOMOYO Linux, *NTT DATA CORPORATION*, http://tomoyo.osdn.jp/cgi-bin/lxr/ident

[4] “Linux Kernel: NULLポインタエラーハンドリング(ERR_PTR, IS_ERR, PTR_ERR)”, *debimate*,

https://debimate.jp/2019/03/02/linux-kernel-null%E3%83%9D%E3%82%A4%E3%83%B3%E3%82%BF%E3%82%A8%E3%83%A9%E3%83%BC%E3%83%8F%E3%83%B3%E3%83%89%E3%83%AA%E3%83%B3%E3%82%B0err_ptr-is_err-ptr_err/

[5] The Linux Kernel API,

http://www.cs.bham.ac.uk/~exr/teaching/lectures/opsys/10_11/docs/kernelAPI/index.html

[6] “[Linux][Kernel] 커널크래시란”, *egloos*,

http://egloos.zum.com/rousalome/v/9977539

[7] “KernelNewbies: FAQ / BUG”, *Linux Kernel Newbies*,

https://kernelnewbies.org/FAQ/BUG