CSE-506 (Spring 2018) Homework Assignment #3

Linux kernel-based system to support process-based queues for handling file protection, compression

Team Member - Rahul Sihag (111462160)

Table of Contents

1. About

The purpose of this assignment is to develop a Linux kernel-based system to support process based queues for handling file protection, compression, etc.

2. Introduction

In Unix, when a file is deleted, it's gone forever and very hard to recover, even if a user mistakenly deleted a file. That is why most OSs today have a "**trashbin**" folder. That way, users have a way to undo/recover a lost file deleted by mistake. But, files in a **trashbin** take up space, so we need a way to compress and delete them after a time. Also, those files sit there for a long time and may enable attackers to steal the data, so we'd like to be able to encrypt them. Still, compression and encryption take time, so we'd like to do those asynchronously.

3. Functionalities

Please find below the complete list of functionalities implemented:

3.1 Encryption

Encryption enhances the security of a file by scrambling the content. The main purpose for encrypting the data stored in your computer and devices - even if you have created back-ups or secure passwords - is to ensure your privacy, protect your data, and secure intellectual property. For encryption, I am using **AES** (Advanced Encryption Standard) algorithm. During unlink time if the process is setup with **CLONE_PROT_ENC** flag, the file to be deleted will be encrypted and moved to the trashbin folder. The encrypted file will have an extension ".enc". Users can set their encryption keys or passphrases for encryption using a special IOCTL. If the encryption key for any user is not set, the system will encrypt using a default key set at the mount time. If encryption key at mount time is not specified it will use another key configured in the system.

3.2 Compression

File compression reduces the amount of space needed to store data. For compression, I am using **deflate** algorithm. During unlink time if the process is setup with **CLONE_PROT_ZIP** flag, the file to be deleted will be compresses and moved to the trashbin folder. The encrypted file will have an extension ".comp". If both encryption and compression are set I do the both in one go. Read the file once page by page compress it and then encrypt it.

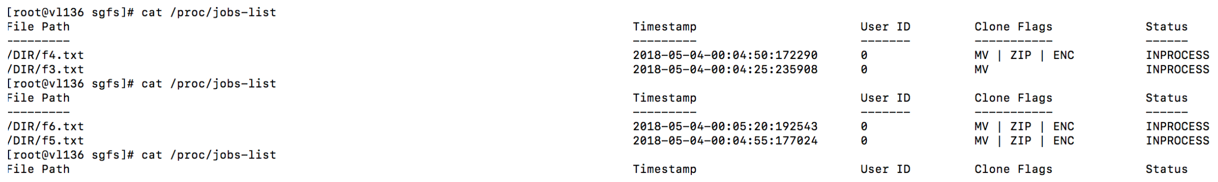
3.2 Move to trashbin

File compression reduces the amount of space needed to store data. For compression, I am using **deflate** algorithm. During unlink time if the process is setup with **CLONE_PROT_ZIP** flag, the file to be deleted will be compresses and moved to the trashbin folder. The encrypted file will have an extension ".comp".

3.2 List

Users want to know which files are currently queued for processing. This is also useful for you for debugging. Users can see the current status of the job in /proc/jobslist. It shows the name of the file, job submission time, Process ID of submitter, operations to be done and job status.

Example below:

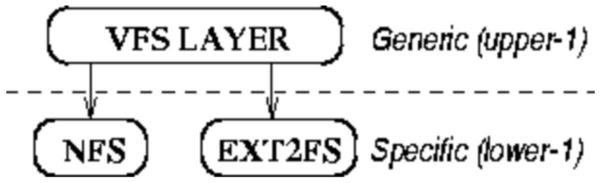


```
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                   Timestamp                     User ID         Clone Flags          Status
---------                                                   ---------                     -------         -----------          ------
/DIR/f4.txt                                                 2018-05-04-00:04:50:172290    0               MV | ZIP | ENC       INPROCESS
/DIR/f3.txt                                                 2018-05-04-00:04:25:235908    0               MV                   INPROCESS
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                   Timestamp                     User ID         Clone Flags          Status
---------                                                   ---------                     -------         -----------          ------
/DIR/f6.txt                                                 2018-05-04-00:05:20:192543    0               MV | ZIP | ENC       INPROCESS
/DIR/f5.txt                                                 2018-05-04-00:04:55:177024    0               MV | ZIP | ENC       INPROCESS
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                   Timestamp                     User ID         Clone Flags          Status
```

4. Stackable File Systems

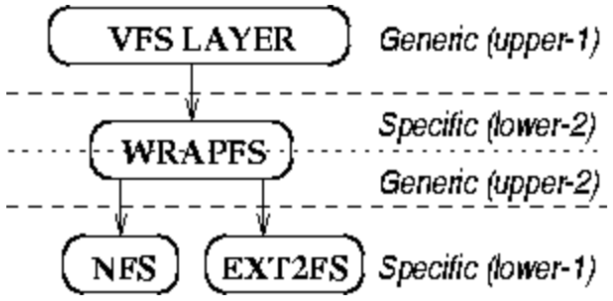
In a stackable file system, each VFS-based object at the stackable file system (e.g., in Wrapfs) has a link to one other object on the lower file system (sometimes called the "hidden" object). We identify this symbolically as X->X' where "X" is an object at the upper layer, and X' is an object on the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

Without stackable file system support, the divisions between file system specific code and the more general (upper) code are relatively clear, as depicted in figure below.



Normal File System Boundaries

When a stackable file system such as Wrapfs is added to the kernel, these boundaries are obscured, as seen in figure below.



File System Boundaries with Stackable File System (WRAPFS in fig. above)

5. Design

1. **Unlinking** – If a user calls sgfs_unlink either on doing a rm or deleting a file using an ioctl, the unlinking functionality first checks for the CLONE flags set to the process unlinking.
   If **CLONE_PROT_MV** is not set – the file will be deleted permanently.
   If **CLONE_PROT_MV** is set – the file will be compressed/ encrypted on the basis of CLONE_PROT_ZIP/ CLONE_PROT_ENC set or unset.
   If any of the CLONE_PROT_ZIP/ CLONE_PROT_ENC is set user program has to explicitly set the CLONE_PROT_MV, otherwise the file gets deleted permanently.

   In case both ZIP and ENC are set the file is first compressed and then encrypted. This results in less data to write to the new file and is also better from security point of view.
   If encryption is done and then compression then file size is more – compression depends on properties of file and encrypting first loses those properties (encryption first increases the entropy of the file and thus compression is not at the best). Also, reverse engineering

can yield patterns in case of encrypt-then-compress. Thus compress-then-encrypt works better from both security and file size point of view.

2. **Clone Flags –** Three new flags have been added to the clone system call to track the MV, ZIP and ENC status of a process being cloned.

3. **Unlinking Small Files –** If the file size is less than or equal to 4KB the job is processed synchronously. File name is modified to include timestamp/ user id and comp/enc information.

4. **Unlinking Large Files –** If the file size is more than 4KB the job is submitted asynchronously to a producer thread. If the work queue is full (work queue size set by a proc entry) producer thread waits and the user program is blocked until the queue size goes less than the threshold. After the producer thread runs it inserts the job in asynchronous queue, update the list asynchronous proc entry and wakes up a consumer thread. Proper locking mechanism is used to access the shared asynchronous queue data structure. A consumer thread will take up this job and perform it asynchronously and will update a proc entry about the status of the job. File name is modified to include timestamp/ user id and comp/enc information.

5. **Procfs Entries –** Different proc entries have been created as explained below

6. **Recovery** – A file can be recovered back using a user program using ioctls. The file is decrypted/ uncompressed back (if applicable) and moved to its original location in its original name.

7. **Trashbin Cleaning** – A thread periodically cleans trashbin folder and deletes the older files more than the threshold number set by a proc entry. If nothing is set default number is 10.

8. **List queue content** – Current queue's content can be looked using cating a proc entry as explained below.

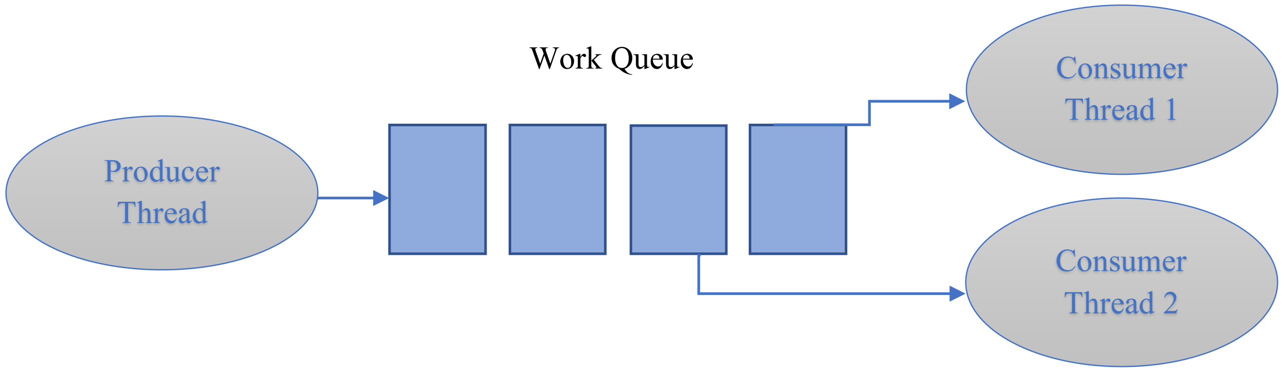
9. **Set password** – Users can set/modify their encryption keys and these keys are encrypted and stored persistently in a read only file. Users can't modify the file.

10. **Purge Trashbin** – Users can purge the trashbin completely using an ioctl. Purge trashbin can only be done by the root user or we need to set the .trashbin permission to RWX for the user . Currently the user has only RW permission to the trashbin. Normal user can delete its file.

11. **List jobs completed and status** – Last 5 completed jobs status can be seen from jobs-completed proc entry.

6. Asynchronous Queuing Subsystem

The basic idea of asynchronous queue system is shown below:



Work queue have a maximum size and if the queue is maxed out, the unlink block until the queue length is under the max (throttling the heavy writer).

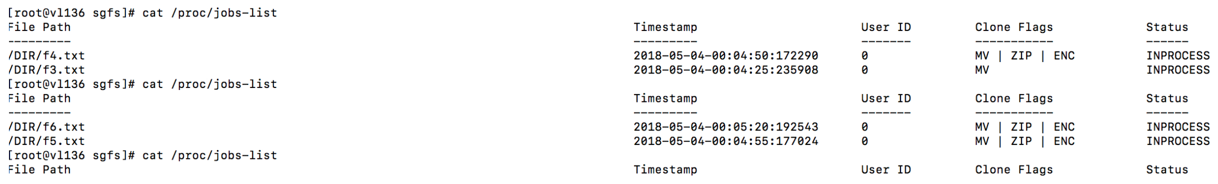
Locking Mechanism – Locking is very important when it comes to accessing shared queue due to concurreny. Here the job queue is shared between the producer/ consumer and the main thread (for listing jobs in proc entries) and thus doing any operation on the queue requires proper locking mechanisms. So, I have used a mutex lock on the job queue which makes insert/ remove/ list without any concurrency issues.

If the queue is maxed out unlink will block and will only return when it has some free space to accomodat any new producer. This way heavy writers are throttled.

7. Procfs Entries

1. **kernel-queue** - Used to set the kernel's asynchronous queue length. Kernel queue have a default max length of 10. Reading this config file using cat /proc/kernel-queue displays the current max queue length. Writing to this config file (echo "20" > /proc/kernel-queue) will set the new max limit of kernel queue (20 in the example aforementioned). Setting the limit to 0 turns off the asynchronous processing and all the jobs will be processed synchronously.

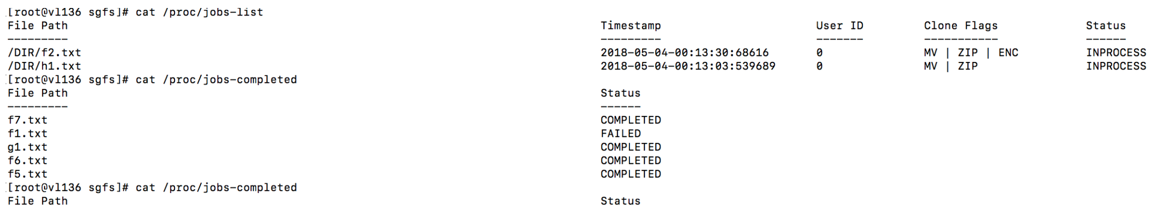
2. **jobs-list** - Used to know which files are currently in the kernel queue queued for processing. If a user bin cat (cat /proc/jobs-list) this proc entry it will list the queue's content with details like file name, job submission time, process ID of submitter, options set for the job and current status of the job.



```
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                                               Timestamp                        User ID         Clone Flags             Status
---------                                                                               ---------                        -------         -----------             ------
/DIR/f4.txt                                                                             2018-05-04-00:04:50:172290       0               MV | ZIP | ENC          INPROCESS
/DIR/f3.txt                                                                             2018-05-04-00:04:25:235908       0               MV                      INPROCESS
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                                               Timestamp                        User ID         Clone Flags             Status
---------                                                                               ---------                        -------         -----------             ------
/DIR/f6.txt                                                                             2018-05-04-00:05:20:192543       0               MV | ZIP | ENC          INPROCESS
/DIR/f5.txt                                                                             2018-05-04-00:04:55:177024       0               MV | ZIP | ENC          INPROCESS
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                                               Timestamp                        User ID         Clone Flags             Status
```

3. **trashbin-max** - Used to set the maximum number of files in the trashbin folder. If the trashbin folder has more then the maximum number of files, older files are deleted by a kernel thread that runs periodically (every 60 secs). Read/Write to this proc entry using cat/echo**.**

4. **jobs-completed** - Used to know which jobs have been processed and what is the status. Since the jobs are processed asynchronously user has no way to determine if the job has been succeeded or failed. User can check this proc entry and check the status. This entry has a buffer size of 5. i.e. it shows the status of last 5 jobs processed.



```
[root@vl136 sgfs]# cat /proc/jobs-list
File Path                                                                Timestamp                    User ID       Clone Flags          Status
---------                                                                ---------                    -------       -----------          ------
/DIR/f2.txt                                                              2018-05-04-00:13:30:68616    0             MV | ZIP | ENC       INPROCESS
/DIR/h1.txt                                                              2018-05-04-00:13:03:539689   0             MV | ZIP             INPROCESS
[root@vl136 sgfs]# cat /proc/jobs-completed
File Path                                                                Status
---------                                                                ------
f7.txt                                                                   COMPLETED
f1.txt                                                                   FAILED
g1.txt                                                                   COMPLETED
f6.txt                                                                   COMPLETED
f5.txt                                                                   COMPLETED
[root@vl136 sgfs]# cat /proc/jobs-completed
File Path                                                                Status
```

8. APIs added/modified

The list of APIs added/modified as part of the code are given below:

A. **procmodules** - directory containing the code base for different proc modules created as part of the assignment.

1. procmodules/jobs-list: contains the code base for jobs-list proc module.
2. procmodules/kernel-queue: contains the code base for kernel-queue proc module.
3. procmodules/trashbin-max: contains the code base for trashbin-max proc module.

B. util.h – file containing helper functions used by many APIs

```
1. int string_to_int(char *num) - convert string to integer
2. int ceph_aes_encrypt(const void *key, int key_len,
                void *dst, size_t *dst_len,
         const void *src, size_t src_len) - API used for encryption

3. int ceph_aes_decrypt(const void *key, int key_len,
                void *dst, size_t *dst_len,
         const void *src, size_t src_len) - API used for decryption

4. int list_asyncqueue(void ) - writes the kernel queue job list to jobs-list proc
   entry
5. int insert(struct job *job) - inserts a new job to the kernel queue
6. struct work *remove(void ) - removes a new job from the tail of kernel queue (FCFS)
7. int process(struct job *job) - process current job
8. int producer(void *data) - callback function used by producer thread, wakes up a
   consumer kernel thread
```

9. int xcrypt_read(struct file *file, unsigned long long offset, unsigned char *data, unsigned int size) – file read
10. int xcrypt_write(struct file *file, unsigned long long offset, unsigned char *data, unsigned int size) – file write
11. int xcrypt_open(const char *path, int flags, int rights, struct file **fileptr) – file open
12. void xcrypt_close(struct file **fileptr) – file close
13. void print_md5(unsigned char *key_hash) - print md5 hash
14. int calculate_md5(char *dst, char *src, int len) – calculates md5 hash of a key
15. char *get_user_key(unsigned int uid) – get the key of the user from keys file
16. int set_user_key(char *key, unsigned int uid) – set the key of the user in keys file
17. void swap_file_details(struct file_details *a, struct file_details *b) - helper function for bubbleSort API
18. void bubbleSort(struct file_details *start) - API for bubble sorting a linked list
19. void push(struct file_details **head_ref, struct timespec file_time, struct dentry *file_dentry) - push node to the head of linked list
20. int filldir_one(struct dir_context *ctx, const char *name, int len, loff_t pos, u64 ino, unsigned int d_type) – callback to iterate dir. fills DS in linked list
21. int consumer_callback(void *data) – callback function for consumer thread
22. int callback(void *data) – callback funtion for background cleaning thread

C. file.c – modified/addes APIs for IOCTLs/ purging trashbin completely/ list only user specific files

1. void purge_trashbin(void) – purge trashbin completely
2. int action_fcn(struct dir_context *ctx, const char *name, int len, loff_t offset, u64 a, unsigned b) – callback used by readdir. calls check_user to display user specific files only
3. static long sgfs_unlocked_ioctl(struct file *file, unsigned int cmd,
   unsigned long arg) – added new ioctls

D. inode.c – modified/addes APIs for IOCTLs/ purging trashbin completely/ list only user specific files

1. int sgfs_unlink_util(struct inode *dir, struct dentry *dentry) – original unlink API
2. static int decompressed_bytes(struct file *input_file, int *count) – helper for encryption/ decryption/ compression/ decompression. Extracts data length to read from file. Data saved as – len(len(data)len(data)data e.g. abc stored as 13abc
3. int get_file_name_from_dentry(struct dentry *dentry, char *file_name) – given dentry get filename
4. int sgfs_unlink(struct inode *dir, struct dentry *dentry) – modified sgfs_unlink
5. int decrypt_decompress_file(struct file *input_file, struct file *output_file, char *key, int key_len, int write, int read_offset) – decrypt and decompress file in one go. reads only once. writes only once

6. `int encrypt_decompress_file(struct file *input_file, struct file *output_file, char *key, int key_len, int write, int read_offset) - encrypt and compress file in one go. reads only once. writes only once`
7. `int xcrypt_file(struct file *input_file, struct file *output_file, char *key, int key_len, int flag, int write, int read_offset) - api to encrypt/decrypt file on basis of flag`
8. `int compress_file(struct file *input_file, struct file *output_file, int write) - API to compress a file`
9. `int decompress_file(struct file *input_file, struct file *output_file, int read) - API to decompress a file`
10. `char *filename_to_trashfilename(char *uid2str, char *inputfile, int compressed, int encrypted) - api to add details like timestamp, uid, comp/enc info to filename`
11. `char *setup_xcrypt_key(unsigned int uid) - - setup user key or mount point key or default key in key buffer`
12. `int sgfs_unlink_(struct inode *dir, struct dentry *dentry, struct job *curr_job) - unlink helper`
13. `int sgfs_restore(char *filename, struct super_block *sb) - restore a file. called by ioctl`

D. Data structures

```
struct file_details{
    struct timespec file_time;  /* access / modify / create time */
        struct dentry *file_dentry;
        struct file_details *next;
};

struct getdents_callback_ {
        struct dir_context ctx;
        char *name;             /* name that was found. It already points to a
                                   buffer NAME_MAX+1 is size */
        u64 ino;                /* the inum we are looking for */
        int found;              /* inode matched? */
        int sequence;           /* sequence counter */
        struct dentry *trashbin_dentry;
        struct file_details **file_details_list;        /* list head */
        int *list_len;
};

struct job {
        unsigned int pid;               /* process id of submitter */
        int recover_flags;      /* MV, ENC, COMP... */
        int status;             /* 1 - in process, 0 - success, -1 - failure */
        char timestamp[27];     /* timestamp is 26 bytes */
        struct inode *dir;
        struct dentry *dentry;
};

struct work {          /* work queue */
        struct job *job;
        struct work *next;
};
```

Following modifications have been done to kernel code base:

1. modified _do_fork and copy_process API in fork.c to set MV, ZIP, ENC flags.
2. added macros for CLONE_PROT_MV, CLONE_PROT_ZIP, CLONE_PROT_ENC in sched.h file.

9. Important Features

The complete functionality is built on the top of stackable file system wrapfs. Thus, all the features supported by this module works when working under the mount point. The reason for choosing Wrapfs was it gives an abstraction and we can use these features on a part of the filesystem. And as wrapfs provides abstraction and all the calls are intercepted this design is more extensible.

10. How to Run

User Program

Please find the steps below:

1. @restore]# ./sgctl –t /mnt/sgfs/.trashbin/ -u filename - filename should lie in .trashbin
2. @setenckey]# ./sgctl –t /mnt/sgfs/.trashbin/ -e key123
3. @delete]# ./sgctl –d /full/file/path
4. @purgetrashbin]# ./sgctl –t /mnt/sgfs/.trashbin/ -p “”
5. @deletejob]# ./sgctl -t /mnt/sgfs/.trashbin/ -q filename

sgctl is present in hw3/sgctl folder and need to make to build it.

To set the clone flags we need to make use of the clone program given under clone directory in hw3 folder.
Usage of clone:
./clone –[m][z][e] –c [command] –a[args] where m means set move flag, z means set compress flag and e mean set enc flag
command is the user program – we can use both rm or ./sgctl with –d option. args is the arguments taken by the user program

examples of clone:
@move-zip-enc]# ./clone –mze –c “rm” –a “filename”

or using sgctl
@move-zip-enc]# ./clone –mze –c “./sgctl” –a “-dfile_path”

How to compile/run sgfs module –
```
run sh install_all.sh to install all the modules and sgfs
```

To build the kernel go to 3.10 kernel and do
1. make menuconfig
2. make
3. make modules
4. make modules_install install
5. reboot

11. Test programs/ Scripts
All the features have been tested manually and all of them work fine. Please find the automation scripts in hw3/scripts folder.
For testing I have put timers in the code that be directly configured by changing the values is sleeptimer.h header file.

To run the test cases put the scripts in mount path and execute sh test_suite.sh. It will run ten test cases. Required files for testing are present in testing folder in hw3.

12. References

1. https://www.geeksforgeeks.org
2. http://wrapfs.filesystems.org/docs/linux-stacking/index.html
3. https://elixir.bootlin.com
4. http://www.staroceans.org/projects/beagleboard/net/ceph/crypto.c
5. https://www.kernel.org/doc/htmldocs/kernel-hacking/queues.html
6. procfs – refered from internet blog