DESIGN DOCUMENT

Asynchronous implementation of cryptographic API based file operations

Akhilesh Chaganti, Nafees Ahmed, Raghavendra Rao Suvvari
109648086, 109595182, 109596686

1. Introduction

Heavy activities involving file operations consumes lot of CPU and data IO along with context switches between user and kernel space. This work implements a classic producer/consumer queue which takes certain jobs, performs them asynchronously and sends back result to the user.

The list of jobs that can be performed include encryption/decryption, compression/decompression and checksum computation over files along with support for special functions like listing current jobs and removing a specific job from the queue. The following sections give more details of design and implementation, along with the testing and certain limitations. This code can be installed as a loadable kernel module in vanilla Linux kernel 3.2.2.

2. Design & Implementation

The code when installed as a kernel module supports a system call which acts as producer accepting jobs from the user space. This module also initializes a work queue which is used by the system call to queue the jobs. The module creates certain kernel threads which act as the consumer processes de-queuing the jobs from the work queue and processing them. We will go through detailed design of work queue creation and maintenance with lock infrastructure, followed by design of producer and consumer.

(a) Workqueue:

This workqueue is used for asynchronous development of certain file operations. The first thing we have to keep in mind is that they take significant memory because of which have to constrain its length. The length of this queue is a compile time configurable parameter. Also, this queue is used independently by different producers and consumers. Hence we protect the queue with a mutex lock. There are various cases where IO needs to block in the worst case in the critical section of producer and consumer (like kmalloc). Hence, the mutex lock is taken leaving behind the spin and complicated RCU locks. So every operation that handles queue structures like the head, tail and its length should take a lock before accessing it. For example any producer/consumer which wants to add /remove a queue structure it needs to take the mutex lock.

(b) Producer:

The system call "sys_xjob" becomes actives as soon as the module is loaded into kernel. This system call acts as a producer which loads jobs to the work queue. This system call mainly performs three tasks: 1) Encryption/Decryption of files, 2) Compression/Decompression of files and 3) Checksum for the files. Along with these features, the system call performs two special actions: 1) Listing the current jobs and 2) Removing as specific job from the queue. This system call accepts a structure "job" as input. The structure from user contains the following elements.

1) Input file for processing
2) Output file for writing the result (Optional based on the type of task)

3) PID of the process submitting the job
4) Type of the job
5) Algorithm/API used to perform the job.
6) Cipher key used in encryption
7) An extra int variable for any other options in future. (Presently used to refer the job to be removed from queue in "REMOVE" task)

The kernel side structure for representing a job contains all the above elements along with an id which uniquely identifies every job. Initially the system call performs all the preliminary checks to verify legal memory access to the structure and all its elements. Then it checks the conditions that need to be met for specific jobs. For example, input files must be present for all types of jobs except LIST and REMOVE operations. Similarly, output file must be present for encryption/decryption and compression/decompression. PID is mandatory for every job as it is needed while are using asynchronous callback to communicate result back to user. Job type should be specified based on 7 integers which represent 7 different jobs supported by the system call, else an -EINVAL is returned back. For every job specified, details of API must be given based on which the job is computed. Special arguments like cipher key is required for encryption/decryption. This structure accommodates an extra int which is used to pass the job id of the process to be removed. The last elements cipher (char*) and special option (int) can be used to store any future support to increase number of jobs that can be performed.

After all these checks are performed the global mutex lock is taken before assigning the job id. If the job is LIST or REMOVE, queue is iterated over to store all jobs or to remove a specific job in queue. The result of these operations is communicated back using the callback which is discussed later. For other jobs, length of queue is checked first. If the length goes beyond queue length constraints, we don't return any busy. Instead we maintain wait queue which preempts the current process and places it in wait queue i.e. the process in forced to sleep while the queue is not empty. If the length constraint is satisfied, a new queue structure is added with present job into the work queue. Following this, the system call checks if there are any waiting consumers in its respective wait queue and wakes them up to process the jobs. Finally this system call either returns an error number or unique id for the job to the user. The lock is released in after the queuing is done or before forcing the process to sleep (in this case all operations from queue length check are performed again along with holding a lock).

(c) Consumer:

Certain kernel threads are created which acts as consumers when the module is 'insmod'ed into the kernel. This number is a compile time configurable parameter. Initially all the threads that are created are added to wait queue waiting (sleeping) for any "wake up" instructions from the producer. Whenever a job is added, the producer tries to wake up one job present in the wait queue. Any active consumer thread first takes a lock to check the queue length. If the queue is empty the thread is added to wait queue again. In case if there is any existing job in queue, it removed from queue, following which a "wake up" is issued to the wait queue of the producers to wake up one producer if the wait queue is not empty. Following this, the specific task is performed over the job, based the parameters in the structure. Here the

result for each job is sent back to user using the callback structure. Following this the consumer checks if it has been asked to be destroyed (in case of rmmod) to exit the consumer thread. Else it will be repeat the same process waiting for jobs.

(d) Asynchronous call back mechanism:

Due to the asynchronous nature of the producer/consumer infrastructure, we need to maintain a callback mechanism which communicates with the user process. This communication usually carries the data/return values (success or failure) of job submitted by the user. After giving a thought for efficiency, support for carrying complex information (i.e. different structures of data) and ease of implementation we have zeroed on netlink based communication between kernel and user. This implementation is very similar to TCP/IP socket communication except for the fact the end points of communication lie in user space and kernel. A socket for this communication is created in the kernel at the time of module loading. This socket listens to any user spaces and requests and transmits any message destined for a particular process in userspace. There are several instances where we use this method to transmit data to the user. This callback method for communication accepts job id, any result along with its size, the type of message (list of jobs, int, char*) and the PID of the user process. The call back packs all of the above details in structure (struct job_res in code) and unicasted to the particular PID of the user process. If the PID is not existing, -ESRCH is returned.

(e) Other details:

When we try to remove the module (of course when no user process holds the call to "xjob"). All the jobs which are present in queue are removed and communicated back with an interrupt (-EINTR) error. A flag destroy is set and wakes up any kernel thread present in the wait queue eventually destroying the whole queue and consumer threads. This method is considered for many reasons. Wait for the processes to end turned out to be mediocre implementation as it might take a long time to process all the jobs in queue mean while any additional producers can add tasks to the queue.

Supported Functions by the module and its design:

With the help of Cryptographic API we have implemented Encryption, decryption, compression, decompression and checksum.

Encryption and Decryption:

In general the algorithms supported by this API implement symmetric key encryption, where a plain-text message is encrypted with a key to produce cipher text. Generally, the same key is used to decrypt the cipher text back into the original plain text. It should be easy to encrypt and decrypt messages with the key (which must be kept secret) but difficult to do so without it. Examples of cipher algorithms are Triple DES, Blowfish and AES. There are two types of ciphers: block ciphers operate on fixed-length blocks of data (e.g., 16 bytes), and stream ciphers use a key stream to operate on as little as one bit of data at a time. Ciphers also may operate in a variety of modes, such as Electronic Codebook (ECB), where each block of plain text is simply

encrypted with the key, and Cipher Block Chaining (CBC), where the previously encrypted block is fed into the encryption of the next block.

Implementation strategy for Encryption and Decryption: In our project implementation we have used AES cipher for encryption. As AES is a block cipher, encryption/decryption are done only in respective block sizes. In case of AES implementation the standard AES Block size should be 16 bytes. This standard 16 bytes block size is defined in <openssl/aes.h>.

The cryptographic API mainly deals with two primary objects

- *Algorithm implementations:* kernel modules that contain the underlying algorithm code.
- *Transforms:* objects that instantiate algorithms manage internal state and handle common implementation logic.
  Transforms are managed by crypto_alloc_tfm() and crypto_free_tfm(). A set of API wrappers are provided to
  Simplify transform use and to allow the properties of a transform's underlying algorithm to be queried.

For encryption we are using **crypto_cipher_encrypt_one** () function from cryptographic API and iterating the same function for every 16 bytes of data for the entire file.

But as per the implementation method of above function using AES cipher logic, when the block size is less than 16 bytes, the function automatically pads additional bits to the output buffer (encrypted buffer) which should always be 16 bytes in size. To avoid this we are inserting EOF (end of file) when where when encountered reading buffer less than 16 bytes from input file (most possibly this situation is encountered while reading last part of the input file)

While decrypting the text we are following the same procedure using the counter function **crypto_cipher_decrypt_one**() and if we have EOF in the data buffer we explicitly avoid writing the unnecessary padded bits to output file.

In this way, we are successfully encrypting and decrypting a given file.

Compression and Decompression:

Compression—this is often used in conjunction with encryption so that it is more difficult to exploit weaknesses related to the original plain text as well as to speed up encryption (i.e., compressed plain text is shorter). By definition, encrypted data should be difficult to compress, but this adversely affects performance over links that normally utilize compression. Compressing data before encryption helps reduce this performance hit in many cases. Examples of compression algorithms are LZS and Deflate.

Implementation strategy for Compression and Decompression: In our project we are using deflate algorithm. Managing the transforms and registering the respective algorithm is same as the one we have discussed above in implementation of encryption/decryption.

We are using **crypto_comp_compress** function for compression and **crypto_comp_decompress** function for decompression. These methods take transform object, source, destination buffers and respective lengths.

The main challenge that was involved here is that these functions work well for page size (which is in general 4096 bytes) but when a file larger than page size is given, desired outputs are not produced. To enhance the capability to work for files having more than page size we have implemented a different logic.

We need to make sure that number of bytes compressed and number of bytes given as input while decompressing should be same.

For this, while compressing the file data and writing compressed data into output file, we are also writing the buffer having lengths of compressed chunks at the end of file .By doing this, while decompressing the file, we read this buffer from end of file to get the lengths of compressed chunks and hence decompress accordingly to get desired outputs. We have tested with many test cases and our implementation gives desired output most of the times. As the buffer containing lengths has to be predefined, we have created this buffer with 20 bytes and if huge files are given as we cannot accommodate more values into this buffer sometimes we may end up with undesired output.

Checksum:

Checksum is a simple error-detection scheme in which each transmitted message is accompanied by a numerical value based on the number of set bits in the message. The receiving station then applies the same formula to the message and checks to make sure the accompanying numerical value is the same. If not, the receiver can assume that the message has been garbled.

Implementation strategy for Checksum: In our project we have used three different algorithms to find checksums – MD5, SHA1, CRC32 and CRC32C. These algorithms return unique digest value for any given file. Managing the message digest and registering the respective algorithm is same as the one we have discussed above in implementation of encryption/decryption.

The function **crypto_alloc_hash** registers which algorithm to be used for finding the checksum. The functions **crypto_hash_update** and **crypto_hash_final** find the digest values for the data which are sent in page sizes. The digest value is calculated for each page and at the end the collective digest values for all the pages are again processed to find the final value for the whole file.

The MD5 algorithm will give a signature value of size 32 bytes and SHA1 algorithm will return a signature value of size 40 bytes. The digest values are unique for all the files and will depend only on the content of the file. The other two methods to compute checksum is to use Cyclic Redundancy Check (CRC32) Checksum API. Using this method produces an unsigned 32 bit identifier (u32 type) which serves as checksum. The crc32c (u32 crc, data, size) and crc32 (u32 crc, data, size) are two APIs used to compute crc32c and crc32 checksum. Both of these functions compute a checksum for data buffer of given size and previous checksum if any. For the input file passed, each time checksum is computed for the data buffer of size page-size. The checksum is computed iteratively till the entire file is read. This checksum is converted to string and copied to the buffer given by the caller. Later this filled buffer is passed to user with the help of netlink callback.

(f) Supported special functions:

i. LIST:

This function requires no special inputs from the user except the PID of requested user. When the system call sees the "LIST" job, instead of adding the structure to queue, the queue is iterated over and list (struct jlist in code) is constructed containing the job id, PID of the process and type of task for every queue item. This array of jlist if passed over to netlink callback which communicates the result to user.

ii. REMOVE

This function requires the id of the job to be removed from the work queue. When the system call sees the "REMOVE" job, instead of adding the structure to queue, the queue is iterated over to find the queue element with specific id. If the id is found, the element is removed from the queue, following which a netlink message with (-EINTR) is sent to the user process of deleted id. After a successful deletion from the queue, zero is sent to user process which requested the deletion. If the queue element is not found with specific id, then -ENOENT is sent back to the user over a netlink message.