Operating Systems(CSE-506)

Asynchronous Producer Consumer Utility

ver 1.0
Dec 03, 2015

Design Document

hw3-cse506g10

Aditya Prakash (110350983)
Vishal Sahu (110310091)
Romeyo D'souza (110559645)
Kumar Sasmit (110308698)

1 Contents

1. INTRODUCTION

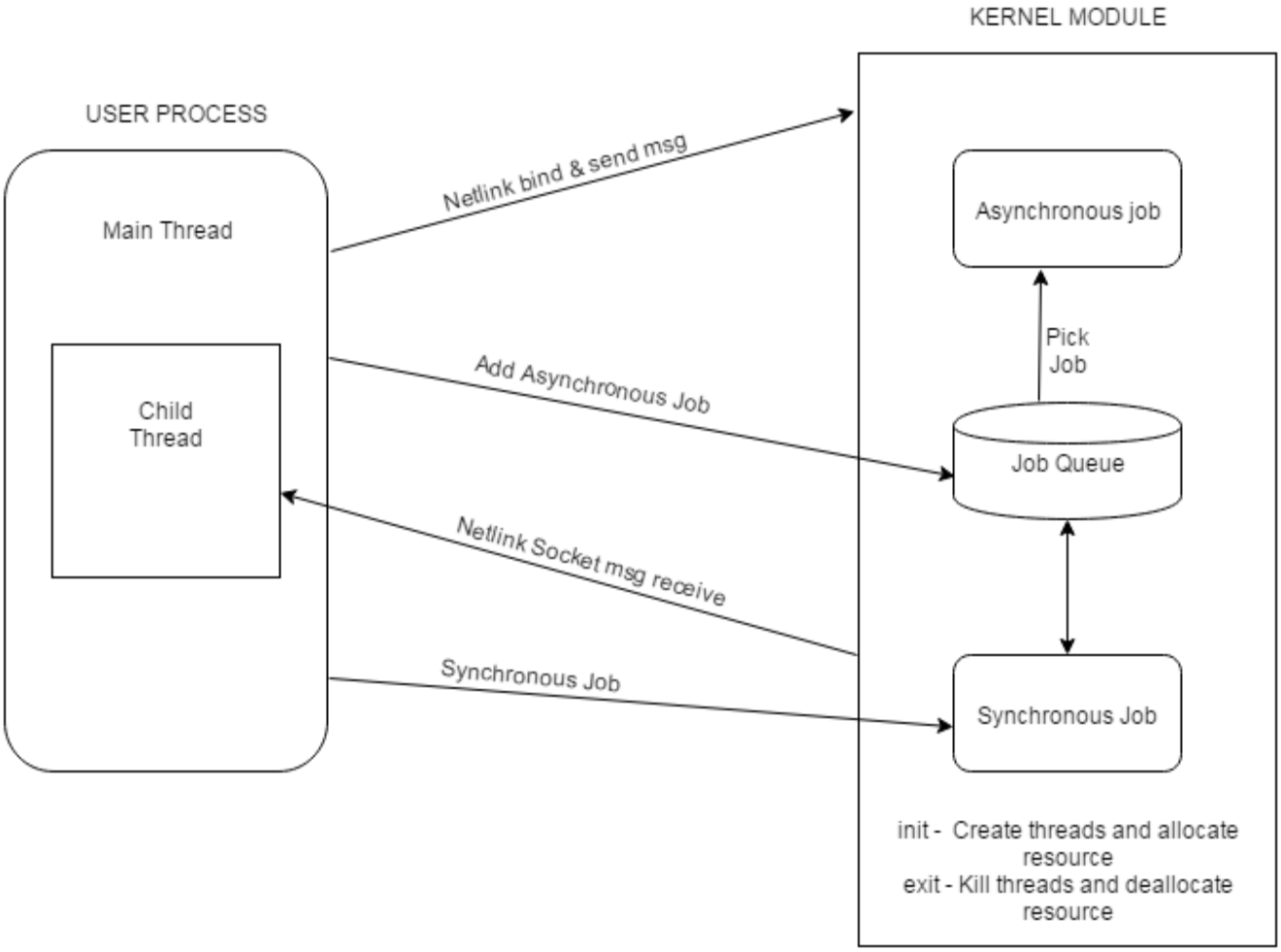
This document explains the design of asynchronous utility module for Linux kernel performing various operations e.g., encryption, compression, checksum etc. It implements producer-consumer job queue supporting priority and fair scheduling. In multi-threaded execution scenario, the asynchrony is handled using various kernel locks. This module is very useful for file systems requiring these utility operations on specific files concurrently.

2. MOTIVATION

This design is motivated by the curiosity to understand asynchronous behavior in multi threaded operations in Linux kernel. Conventional methods to communicate between userland and kernel such as system calls, IPCs, procfs etc. are blocking methods which are not efficient from performance point of view.

We intended to design a mechanism such that user submits the job and goes on to continue other important tasks. Later when the submitted request gets served by kernel, user is interrupted via netlink callback.

2 MODULE ARCHITECTURE



*__Asynchronous Jobs__ - Encryption, Decryption, Compression, Decompression, Checksum
*__Synchronous Jobs__- list, delete, flush, modify, change priority

3 USAGE DIRECTIONS

3.1 Building and inserting module

3.1.1 Compilation

We use the Makefile in order to create the various objects files, executables and kernel module object. Run **`make`** - this will create:-

sys_submitjob.ko:- LKM object file which will be used for the loading and unloading of the LKM.

producer: User program which will act as the producer and submits the job to the kernel module.

producer_thread: User program which supports the threaded producer.

Run **`make clean`** to clean up the executable and object files.

3.1.2 Module Load and Unload

We use the shell script file our module load and unload. Please run **`install_module.sh`**

`./install_module.sh` : This will do the following things:

unload the previously loaded module using **`rmmod sys_submitjob`**

Loads up the newly generated module using **`insmod sys_submitjob.ko`**

3.2 Execution

3.2.1 Running the Script

commands:- We have the following commands used for various combinations of the job submitted by the users.

1) **`./producer -l :`** listing the jobs in queue

2) **`./producer -s <infile> :`** Creating Checksum of input file

3) **`./producer -e -p<password> <infile> <outfile> :`** encrypt the plain input file and into encrypted cipher file using password.

4) **`./producer -d -p<password> <infile> <outfile> :`** decrypts the encrypted input file into decrypted output file.

5) **`./producer -f :`** Flush all the jobs in the job queue

6) **`./producer -r <job-Id> :`** Removes the job corresponding to job-Id from the job queue.

7) **`./producer -m <job-Id> -n <low|high> :`** Modifies the priority of the job corresponding to job-Id from high to low or vice versa.

8) **`./producer -n <high|low> <e|d|s|c|u> <options needed as per job> :`** Submits the job with the priority as high or low .

4 CODE STRUCTURE

4.1 producer.c

This file consists of the all the functionalities of the producer for the asynchronous and concurrent system. It also provides the common data structure which will be used to transfer the request from user process to kernel. Moreover, we have here, a number of routines which help in parsing the user commands and validating the job/task/request submitted. Followings are the main APIs and Data Structures along with their description:-

4.1.1 Data Structure:

- **_USER_INPUT**:- This structure will be used to compact the user inputs and send to the kernel for processing.

```
typedef struct _USER_INPUT
{
      int  encrypt;      // set if encryption is requested.
      int  decrypt;      // set if decryption is requested.
      int  checksum;     // set if checksum is requested.
      int  compress;     // set if compress is requested.
      int  decompress;   // set if decompress is requested.
      int  list;         // set if list all jobs
      int  delete;       // set if delete a job
      int  flush;        //  set if flush all jobs
      int  modify;       // set if modify a job
      int  new_priority; // set if new_priority a job
      int  weak_validation; // set if we want to differ validation
      int  job_id;  // store job id to delete/modify
      int  algoSpecified; // set if algo type is passed else //default.
      int  passwordSpecified; // set if password is passed.
      char *algoType; // the name of the cipher requested.
      char  *password;     //  the  human  readable  password  string.  use
      //getpass(3) in future to read passwords.
      char *priority;  // new priority
      char *inputFile;  // inputFile name.
      char *outputFile; // outputFile name.
} USER_INPUT;
```

- **net_link_ret**: The structure to pack the netlink information to the kernel side. This will be used by the kernel side netlink socket to return the job's response by filling the fields.

```
struct net_link_ret {
      int id;
      int job_type;
      int type; /* 0-int, 1- string*/
      char result[80];
```

```
        int size;
} nls_ret;
```

4.1.2 Functions:

- **int parse_input(USER_INPUT *userInput, int argc, char **argv)**: This function uses getopt() and parses the given options. It fills up the options structure(**userInput**) which is then used to invoke the system call.

- **int validate_input(USER_INPUT *userInput):** This function is used to validate the user input. We can add several input validation as requirement changes as per the requirements.

- **int main(int argc, char *argv[]):** This is the main routine of the user process which covers the following tasks:-
  - Parsing User inputs from the command and filling the _USER_INPUT structure to be sent to the kernel.
  - Validating the User Input before passing the structure in the syscall.
  - Creates the hash of the key provided by the user using libssl's SHA1 cryptographic API.

4.2 sys_submitjob.c

The Linux Kernel Module which serves the job requests sent from the user level using the "submitjob" syscall. It is the main body in the kernel which receives the requests from the producers, checks for the consistency/integrity and finally dispatches the jobs according to its job type. Here, we create the consumer threads which keep on waiting in the queue and when the job arrives, they are woken up and eventually starts serving the requests. The consumer threads are the main entities which makes the system asynchronous and therefore makes out system capable of serving asynchronous and concurrent requests.

4.2.1 Functions:

- **int do_work(SYS_JOB *p_job):** Each consumer thread will pick the job from the job queue and dispatches the job requests corresponding to the job type for the subsequent processing. It checks the validity of the picked job's fields and eventually performs the async. operations like encryption, decryption,checksum, compression and de-compression.

- **int consume (void *arg):** This is the consumer thread main routine which is called by the consumer thread as soon as the job is submitted to the job queue. It performs the following tasks:

- Consumer keeps on waiting/listeing for the job submission in the job queue. once the job is submitted the thread is woken up.
- picks the job from the job queue.
- dispatch the job for processing by calling do_work API
- and finally when the job is processed the netlink response is sent to the user process.

- **asmlinkage long submitjob(void *arg):** It is the system call service routine for "**__NR_submitjob**" system call which is called by the user processes. Following are the tasks done here:
  - Validates the jobs submitted by the users.
  - copies the user inputs to the kernel local copies.
  - checks if the job type is synchronous for the job submitted by the user process then do them right away. The job packets requesting these tasks must be freed; whether it is failure or success, it never goes to the job queue. The jobs we are talking about are LIST, CHANGE_PRIORITY, DELETE, FLUSH.
  - adds the asynchronous jobs into job queue which will be later picked by the consumer threads.
  - And eventually acknowledges the user process about the job submission.

- **void kill_consumers(void):** When the module is unloaded then this routine waked up the sleeping consumers and kills them.
- **__init init_sys_submitjob():** Initializes all the static data structures, waitqueues and mutexes. Also spins consumer threads that will consume the job packets. It also initializes the netlink.
- **__exit exit_sys_submitjob():** Releases resources, calls kill_consumers() to flush the waitqueues. Sets appropriate flags for bust consumers to refer after they finish.

4.3 operations.h

This file defines all functions related to operations performed by this producer-consumer asynchronous system. Besides, it has few of the utility APIs which are used for prior handling of the validation and file operations.

4.3.1 Functions:

- **long validate_job(void *arg):** Function to validate the job sent by the user.

- **struct file *open_file_to_read(char *filename):** Utility function to open the file specified by the filename in read mode and eventually returns the file pointer.

- **struct file *open_file_to_write(char *filename, umode_t mode):** Utility function to open the file specified by the filename in read mode and eventually returns the file pointer. It created the file if that doesn't exist.

- **long check_file_validity(char *inputfile, char *outputfile, int validate_outfile):** This function checks the validity of input and output file like if there are directory or the same. Returns 0 on success, else appropriate errno is returned.

- **long copy_userInput_to_kernel(SYS_JOB *user_data, SYS_JOB *kernel_data):** This function copies the user input to kernel space.

- **int compress_file(char *infile, char *outfile):** This is the service routine which serves the compression requests from the user process. It takes the input file and runs the cryptographic method called "crypto_comp_compress" using "deflate" algorithm and outputs the generated compressed file into file specified by the "outfile".

- **int decompress_file(char *infile, char *outfile):** Service routine for the de-compression job type. It takes the compressed file and runs cryptographic methods called "crypto_comp_decompress" using "deflate" algorithm and outputs the decompressed file in the file specified by the outfile.

- **int decrypt_file(char *infile, char *outfile, char *key, int algo):** Service routine for the decryption of the input file using the key provided by the user process. It uses the kernel APIs specified in the section 6.3. If the key provided by the user is not the same as that was used for encryption, then decryption fails.

- **int encrypt_file(char *infile, char *outfile, char *key, int algo):** Service routine which is used for the encryption of the input file using the algo and key provided by the users. It uses the kernel's cryptographic APIs to set key and encrypt the file.

- **int checksum(char *infile):** Service routine for calculating the checksum of the input file specified by infile. The checksum if written to the file with same name as infile but with .md5 extension. We have just used MD5 checksum algorithm for the same.

4.4 sys_job.h

This is main header file of the module defining all the parameters and macros. Below are some important parameters defined in it:

```
/* This defines capacity of job queue */
#define MAX_QUEUE_LENGTH 10

/**
 * This defines the job_id difference above which HIGH_PRIORITY gets
 * priority over HIGH_PRIORITY to prevent starvation
 */
#define WAIT_THRESHOLD 4
```

4.4.1 Data Structure(s)

```
typedef struct _SYS_JOB {
      pid_t process_id;       /* process id */
      int   job_type;         /* type of job */
      int   algo;             /* subtype or algorithm for job */
      int   job_id;           /* job_id */
      int   priority;         /* priority of job, default 0 */
      int   weak_validation;  /* level or strictness of file validations */
```

```
        char  key[DIGEST_SIZE]; /* passkey for encryption */
        int   keyLength;        /* length of key hash */
        char  *infile;          /* input file name */
        char  *outfile;         /* output file name */
    } SYS_JOB;
```

4.5 resource.h

All job queue related operations are defined in this file.

```
/**
 * Frees job_packet in case of deletion/ completion of service request.
 * @param kernel_data: pointer to packet to be freed
 */
void free_job(SYS_JOB *kernel_data)

/**
 * Frees job_packet in case of deletion/ completion of service request.
 * @param kernel_data: pointer to packet to be freed
 */
long list_jobs(SYS_JOB *arg)

/* Flush the job queue */
int flush_job_queue(struct list_head *q)

/**
 * adds low priority job at the tail of queue and high priority job at
 * local_high_head of queue. If there are no low_priority jobs in queue,
 * local_high_head points to head of the queue.
 * Returns job_id on success and appropriate errno is set in case of
 * failure.
 */
long add_job(struct list_head *q, SYS_JOB *j)

/**
 * removes and returns highest priority job from queue to caller. In case if
 * a low priority job has been waiting for longer than certain threashold,
 * it gets picked. This is to ensure fairness and to prevent starvation.
 * Mainly consumer gets served through this call. The job is picked from
 * either of the 2 heads (misleading but we maintained queue head/ local high
 * head to manage priorities in single queue).
 */
SYS_JOB *pick_job(struct list_head *q)

/**
 * Remove a job from a queue with specific job_id. Returns error number
 * if requested job is not found in the queue. On success, returns 0
 */
int remove_job_id(struct list_head *q, int job_id)

/**
```

```
 * Change the priority of given job. On success, returns 0, otherwise -1.
 * In case if job has the same priority what is requested, no changes are done.
 */
long change_job_priority(struct list_head *q, int job_id, int new_priority)
```

5 FUNCTIONING

5.1 Locking semantics

5.1.1 Mutexes

We have used a single mutex (queue_lock) to protect the job queue as well as the curr_queue_length. This mutex is to be acquired whenever an operation related to the queue is to be done. It is also used to protect special signaling variables (stop_consumption and flush_producers) which are used to convey various events.

5.1.2 Wait Queue and Spin locks

We have used Wait Queues for producers to throttle heavy writers. In addition to this, we have limited the number of producers that can be added to the Wait Queue (NUM_CONSUMERS * 5). After that we start rejecting any new request. We have used spinlocks on variables that are used to change variables on which the wait_event_interruptible() condition is dependant. Finally, the Wait Queue is flushed whenever rmmod is executed.

5.2 Queuing mechanism

5.2.1 Synchronous queue operations:

These operations are meant to be served synchronously by kernel without putting them in the job queue. User submits these requests to kernel and system call returns appropriate value after completion. There is no callback initiated by kernel for these requests. All of these operations are performed atomically.

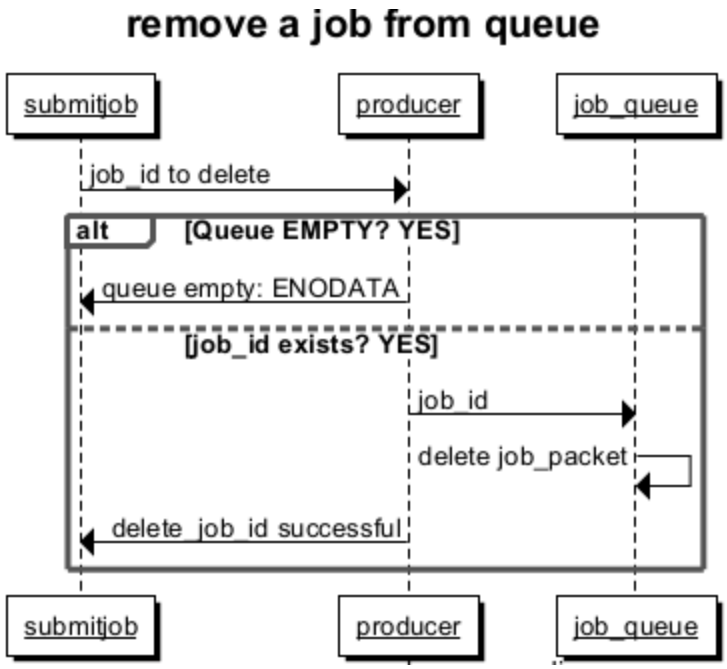
5.2.1.1 List jobs

This option lets user see the current status of job queue. It returns the list of jobs which are currently residing in job queue waiting for consumer. One such instance of list_job query is shown below:

```
number of jobs in queue: 8
| Job Id: 106 |Process id: 7644 |Job Type: COMPRESS   |Job Priority: HIGH_PRIORITY |
| Job Id: 108 |Process id: 7645 |Job Type: ENCRYPT    |Job Priority: HIGH_PRIORITY |
| Job Id: 109 |Process id: 7646 |Job Type: ENCRYPT    |Job Priority: HIGH_PRIORITY |
| Job Id: 107 |Process id: 7647 |Job Type: DECRYPT    |Job Priority: LOW_PRIORITY  |
| Job Id: 110 |Process id: 7637 |Job Type: ENCRYPT    |Job Priority: LOW_PRIORITY  |
| Job Id: 111 |Process id: 7648 |Job Type: DECRYPT    |Job Priority: LOW_PRIORITY  |
| Job Id: 112 |Process id: 7649 |Job Type: DECRYPT    |Job Priority: LOW_PRIORITY  |
| Job Id: 113 |Process id: 7640 |Job Type: CHECKSUM   |Job Priority: LOW_PRIORITY  |
-bash-3.2#
```

5.2.1.2 *Remove a particular job*

This option lets process remove a particular job submitted earlier which is still in job queue waiting for consumer. If any user A removes job submitted by user B, user B gets notified via callback about this removal. Since this is potential security vulnerability, we have added the authentication feature by verifying the parent process ID.



5.2.1.3 *Change priority of a given job*

This options lets user change the priority of particular job submitted earlier which is still in job queue waiting for consumer. If user A changes priority of job submitted by user B, user B gets notified via callback about this priority change. We ensure that user A has privilege to change priority of performing this task.

USAGE: #**./producer -m <job_id> -n <new_priority>**

5.2.1.4 *Flush the job queue*

This requested is considered as cleanup task when user wants to flush the entire job queue and start afresh. This differs from hard remove module(**rmmod**) in that only jobs in queue currently waiting for consumer get flushed and list head and mutex remain intact. User is immediately notified if the job is successful or failed. Note that flush operation is performed atomically and mid-way failures are not considered probable.

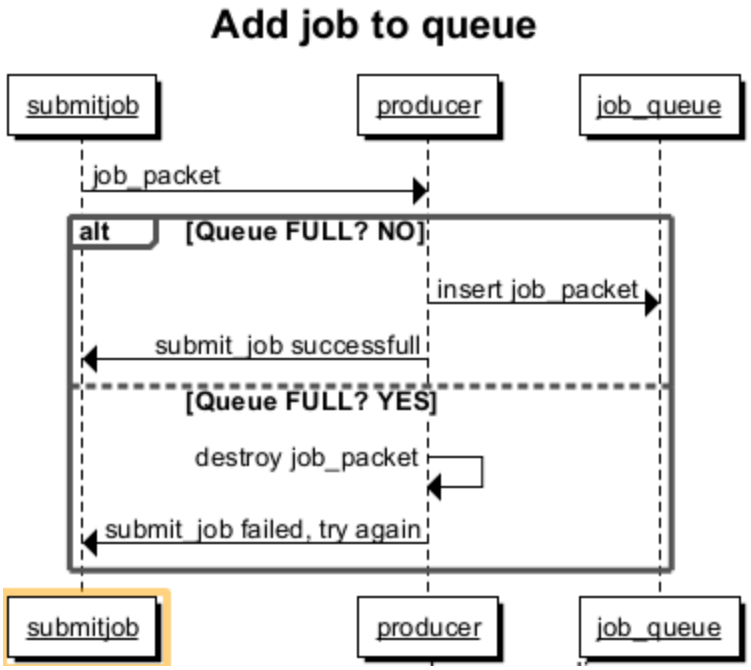
USAGE: #**./producer -f**

5.2.2 Asynchronous queue operations:

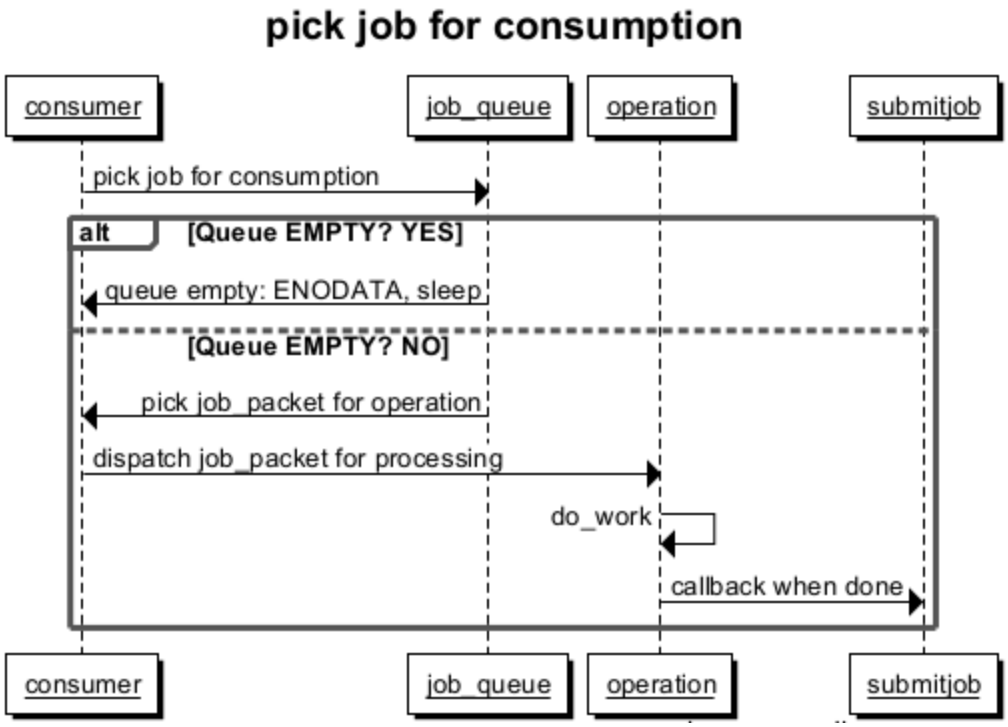
These operations spend time in job queue waiting for consumer. Producer submits the job for processing and then goes on working some other useful task. In mean time available consumer picks the appropriate job from queue, processes it and callbacks user process.

5.2.2.1 *Add job in consumer wait queue*

This is performed by the producer by inserting job_packet received from user process into job queue. Successful insertion into the queue notifies the consumer wait queue.



5.2.2.2 *pick job for consumption*



This operation is performed by consumer when consumer wait queue receives interrupt from add job process which implies that there is a valid job packet in queue for processing. One of the consumer picks the job packet and dispatches it for processing. It also interrupts producer wait queue for inserting new job requests in job queue.

5.2.3 Fairness policy:

- To deal with starvation of lower priority/ long ago added jobs.
- This occurs when higher priority jobs continuously coming in, keeping the lower priority job waiting in queue for consumer.
- Each job is assigned unique job ID and is classified either as HIGH or LOW priority. If the low priority job ID differ by more than certain threshold, it gets consumer instead of the one with HIGH priority.

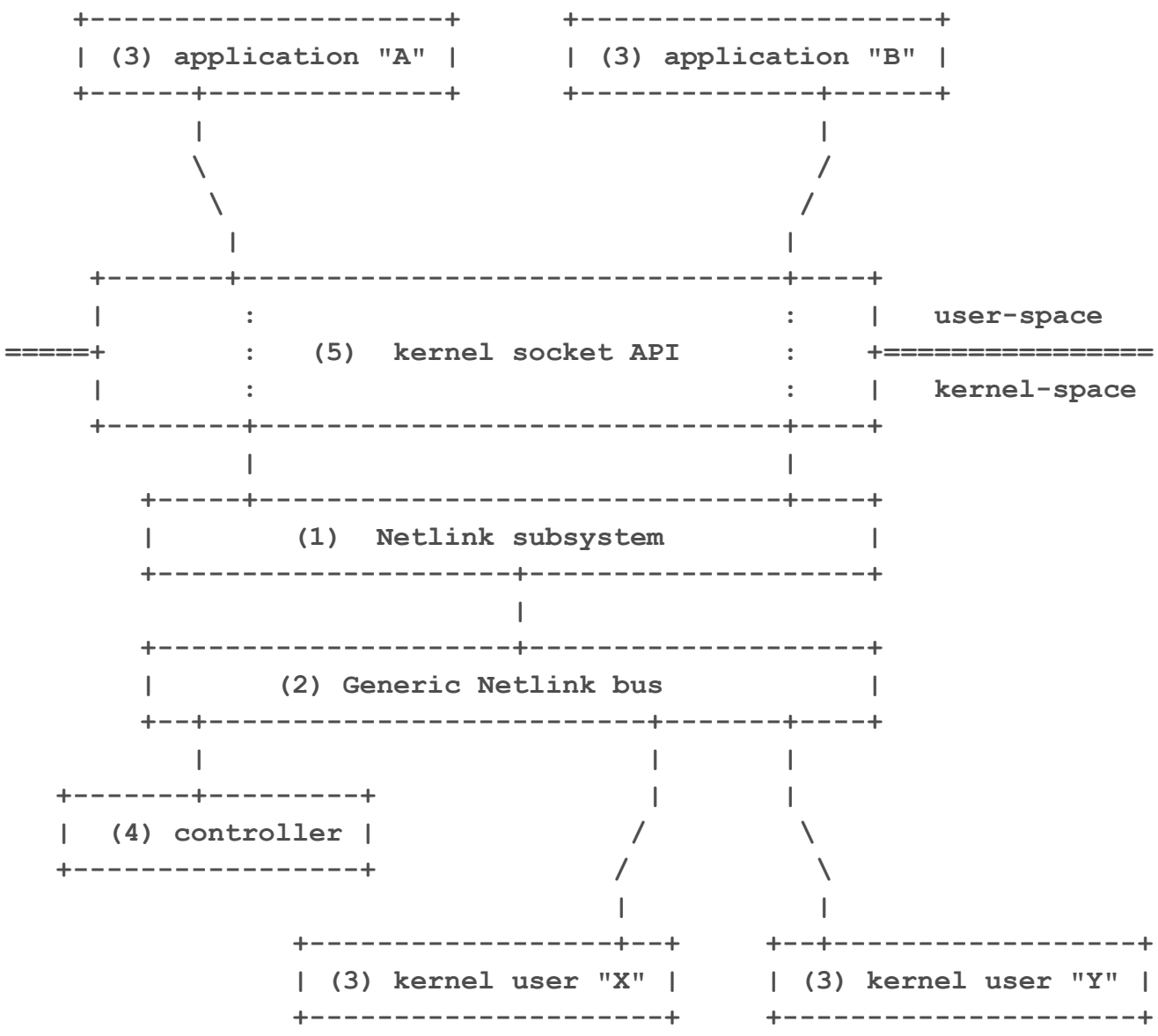
5.3 User call-back

Netlink Socket APIs have been used for interprocess communication between user process and the kernel module. These are similar to the Unix domain sockets. the user process creates the netlink interface and binds to it. On the other side , the kernel module joins the socket at the time when it is loaded and registers for listening and sending messages to the user process through a callback function.

For each operation on the user side where an acknowledgement is required, a message is sent from the kernel module to the user through netlink. The Netlink message from kernel is received in a

separate thread on the user side, to facilitate the user process to perform some other operation, instead of being blocked while waiting for the kernel's reply.

With the present implementation the user process is interrupted and the message is conveyed to the user as soon as the kernel sends it, followed by it, the user process resumes its ongoing task.



Figure[1]: Generic Netlink architecture

5.4 Encryption-decryption

The kernel crypto APIs offers a rich set of APIs for encryption and decryption. We have leveraged those APIs to provide the following services:-

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto

- consumers requesting cryptographic services
- data transformation implementations (typically ciphers) that can be called by consumers using the kernel crypto API.

We have used the Transformation (TFM) objects which maintains the data of the crypt operations. The structure that contains transformation objects may also be referred to as a "cipher handle". Such a cipher handle is always subject to the following phases that are reflected in the API calls applicable to such a cipher handle:

1. Initialization of a cipher handle.
2. Execution of all intended cipher operations applicable for the handle where the cipher handle must be furnished to every API call.
3. Destruction of a cipher handle.

We ensure that the user sends the proper required arguments needed for the complete execution and validation of the requests. Encryption as well as decryption could be done for any size of the file. We made "ctr-generic" mode in aes as the default mode of encryption. However, we have implemented the other algorithms as well which will be accounted for the extra credit. Following are the APIs being used for encryption as well as decryption:

5.4.1 Allocation API

crypto_alloc_blkcipher:- Allocate a cipher handle for a block cipher. The returned struct crypto_blkcipher is the cipher handle that is required for any subsequent API invocation for that block cipher.

struct crypto_blkcipher * crypto_alloc_blkcipher(const char * alg, u32 type, u32 mask);

alg: name of the block cipher
type: type of the cipher
mask: specifies the mask for the cipher

5.4.2 De-Allocation API

crypto_free_blkcipher:- zeroize and free the block cipher handle

void crypto_free_blkcipher(struct crypto_blkcipher *tfm);

tfm: cipher to be freed.

5.4.3 Set Key API

crypto_blkcipher_setkey:- The caller provided key is set for the block cipher referenced by the cipher handle. Returns 0 if the setting of the key was successful; < 0 if an error occurred.

NOTE:- the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

int crypto_blkcipher_setkey(struct crypto_blkcipher *tfm, const u8 * key, unsigned int keylen);

tfm: cipher handle
key: a buffer holding key

keylen: length of key in bytes.

5.4.4 Encryption API

crypto_blkcipher_encrypt:- Encrypts the plaintext data. The blkcipher_desc data structure must be filled by the caller and can reside on the stack. The caller must fill desc as follows: desc.tfm is filled with the block cipher handle; desc.flags is filled with either CRYPTO_TFM_REQ_MAY_SLEEP or 0. Returns 0 if the cipher operation was successful; < 0 if an error occurred.

int **crypto_blkcipher_encrypt(struct blkcipher_desc * desc, struct scatterlist * dst, struct scatterlist * src, unsigned int nbytes);**

desc: reference to the block cipher handle with meta data
dst: scatter/gather list that is filled by the cipher operation with the ciphertext
src: scatter/gather list that holds the plaintext
nbytes: number of bytes of the plaintext to encrypt.

5.4.5 Decryption API

crypto_blkcipher_decrypt: Decrypts the ciphertext data. The blkcipher_desc data structure must be filled by the caller as documented for the crypto_blkcipher_encrypt call above. Returns 0 if the cipher operation was successful; < 0 if an error occurred.

int **crypto_blkcipher_decrypt(struct blkcipher_desc * desc, struct scatterlist * dst, struct scatterlist * src, unsigned int nbytes);**

desc: reference to the block cipher handle with meta data
dst: scatter/gather list that is filled by the cipher operation with the plaintext
src: scatter/gather list that holds the ciphertext
nbytes: number of bytes of the ciphertext to decrypt.

5.5 Compression-decompression

Kernel's Cryptographic APIs for compression and decompression have been used for compressing the uncompressed file and decompressing the compressed file, "**deflate**" being the algorithm used by the transformation object. User can pass any input file (with a size constraint upto PAGE_SIZE = 4kB) for either operation. If the output file does not exist, it is created and if it exists from before, it is truncated and rewritten with the desired contents.

5.5.1 Allocation API

crypto_alloc_comp:- Allocate a cipher handle for a block cipher. The returned struct crypto_comp is the cipher handle that is required for any subsequent API invocation for that block cipher.

struct crypto_comp * crypto_alloc_comp(const char * alg, u32 type, u32 mask);

alg: name of the block cipher, which is deflate
type: type of the cipher
mask: specifies the mask for the cipher

5.5.2 Compression API

crypto_comp_compress:- Compresses the plaintext. Returns 0 if the cipher operation was successful; < 0 if an error occurred.

int **crypto_comp_compress(struct crypto_comp *tfm, char * inbuff, unsigned int insize, char *outbuff, unsigned int *outlen);**

tfm: transformation object
inbuff: plaintext to be compressed
insize: length of the input buffer
outbuff: compressed output buffer.
outlen: length of the output buffer

5.5.3 De-Compression API

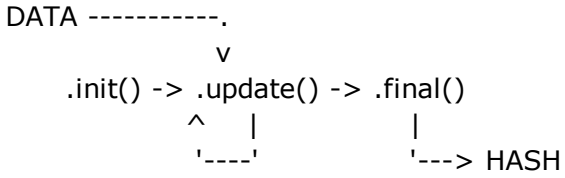
crypto_comp_decompress:- Decompresses the compressed file. Returns 0 if the cipher operation was successful; < 0 if an error occurred.

int **crypto_comp_decompress(struct crypto_comp *tfm, char * inbuff, unsigned int insize, char *outbuff, unsigned int *outlen);**

tfm: transformation object
inbuff: Compressed file to be converted to plaintext.
insize: length of the input buffer
outbuff: de-compressed output buffer.
outlen: length of the output buffer

5.6 Checksum

Kernel crypto APIs provide rich set of the hashing APIs which could be leveraged to calculate the checksum of the input file and corresponding checksum was dumped into output file generated on the fly having ".md5" extension. We used the MD5 cryptographic hashing algorithm to generate the checksum. The generated checksum was 16 byte long for any length of the file. Following if the diagram which completely defines or shows the process of checksum generation.



We used the cryptographic Transformation object and scatter list as explained in above section. Here we will give details of the three APIs used for the checksum generation.

5.6.1 Allocation API

crypto_alloc_hash:Allocate a cipher handle for hash. The returned struct crypto_hash is the cipher handle that is required for any subsequent API invocation for that hash. allocated cipher handle in case of success; IS_ERR is true in case of an error, PTR_ERR returns the error code.

struct crypto_hash * **crypto_alloc_hash(const char * alg, u32 type, u32 mask)**

alg: name of the block cipher

type: type of the cipher
mask: specifies the mask for the cipher

5.6.2 Initialization API

crypto_hash_init: The call (re-)initializes the message digest referenced by the hash cipher request handle. Any potentially existing state created by previous operations is discarded. Returns 0 if the message digest initialization was successful; < 0 if an error occurred.

int **crypto_hash_init(struct hash_desc * desc)**

desc: cipher request handle that to be filled by caller -- desc.tfm is filled with the hash cipher handle; desc.flags is filled with either CRYPTO_TFM_REQ_MAY_SLEEP or 0.

5.6.3 Update API

crypto_hash_update: Updates the message digest state of the cipher handle pointed to by the hash cipher request handle with the input data pointed to by the scatter/gather list. Returns 0 if the message digest update was successful; < 0 if an error occurred.

int **crypto_hash_update(struct hash_desc * desc, struct scatterlist * sg, unsigned int nbytes);**
desc: cipher request handle
sg: scatter / gather list pointing to the data to be added to the message digest
nbytes: number of bytes to be processed from *sg*

5.6.4 Final API

crypto_hash_final: Finalize the message digest operation and create the message digest based on all data added to the cipher handle. The message digest is placed into the output buffer. Returns 0 if the message digest creation was successful; < 0 if an error occurred.

int **crypto_hash_final(struct hash_desc * desc, u8 * out);**

desc: cipher request handle.
out: message digest output buffer -- The caller must ensure that the out buffer has a sufficient size (e.g. by using the crypto_hash_digestsize function).

6 TESTING

- One producer-one consumer
- One producer-multiple consumer
- Multiple consumer – multiple consumers
- Behavior on **`rmmod`**
- Callback verification
- Types of Producers: Busy wait, Submit and exit, Do work while waiting.
- All types of valid and invalid inputs (commands, input files etc.)

7 EXTENSIBILITY

Our framework allows new type of jobs and job requests to be added very easily due to less coupling and more modular design. For example, we have algoType variable for each type of job request along with jobType.

8 ASSUMPTIONS MADE

8.1 Regarding Compression and De-Compression

We assume that the user inputs the infile with the number of characters in it being less than the PAGE_SIZE(which is 4096 bytes for our server), also the file to be decompressed has the same limit. This is because, as per our implementation, the linux kernel compression and decompression crypto APIs are called only once on the input buffer which reads at max PAGE_SIZE at one time.

For the operations to work on larger size, we need to maintain a record which will contain the number of bytes generated by the compression API on the input buffer during the compression operation. This record will be required during decompression operation on the compressed file. During decompression operation decompression APIs will be called repeatedly and each time the number of bytes read from the compressed file will be equal to the information obtained from the record as mentioned earlier. We have not implemented this feature currently but the feature can be implemented by maintaining another file which will be meta-data file for the compressed file, maintaining the record of number of bytes generated during each call of crypto compression API.

9 EXTRA CREDIT WORK

9.1 Process authentication in case of job deletion and change priority

It ensures that only parent process can delete or modify jobs submitted earlier. This is to prevent unauthorized job removal by other processes waiting for job_queue.

9.2 Starvation prevention

Normal job queues don't address the problem of starvation of lower priority jobs. Suppose there are higher priority jobs continuously coming in, the lower priority will keep waiting in queue for consumer. To deal with this issue, we maintain 2 priorities as HIGH/LOW and unique job ID for each job. If the low priority job ID differ by more than certain threshold, it gets consumer instead of the one with HIGH priority.

9.3 Relative path support in kernel

We resolve all the paths (relative or absolute) that are passed from userland to correct absolute paths in the kernel for the kernel threads to use. This necessary as the kernel threads do not have the context of the process that had submitted the job.

9.4 Rmmod handlings

If a rmmod request comes while we are still processing then we gracefully unload the module. We also inform all the producers of jobs in the queue about the queue flush. We also do not kill currently running consumers so that we do not leave incomplete jobs. We notify the running consumers that rmmod has happened and to exit on finishing the current job.

9.5 Weak validation for pipelining dependent jobs using -w

To handle jobs that are dependent on output of jobs ahead in the queue, we have added support to pass a flag for weak validation during submission. The validation for dependent job packet will be deferred until it is picked for processing. By default, the validation is strict to improve queue efficiency.

10 REFERENCE(S)

1. https://en.wikipedia.org/wiki/Asynchronous_I/O
2. https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem
3. http://www.makelinux.net/ldd3/chp-6-sect-2
4. http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto
5. http://www.linuxjournal.com/article/7356
6. https://stackoverflow.com/questions/15215865/netlink-sockets-in-c-using-the-3-x-linux-kernel?lq=1