CSE 506 - Spring 2014
Homework Assignment 3
Group 19 - Vinay Jain, Utpal Thakar

General Workflow of the system

1. User program makes a syscall sys_xjob passing the arguments for the job Eg: filenames for checksum computation and encryption.
2. Handling of sys_call by kernel
   a. Validate the user arguments
   b. Create an internal job structure from the request sent by user program.
   c. Call producer to submit the job to producer-consumer work queue
   d. Wait for consumer to pick the job and process the request
   e. Save the result of operation in result queue and then intimate the user program about availability of the result.
   f. Return the result of processing to the user via asynchronous callback mechanisms.
3. Read the result from the syscall as well as from asynchronous callback mechanisms.
4. Exit

Design of Producer consumer work queue

Work Queue

Work queue is implemented using singly linked list of work element.Each work element has a field to store a job object. We also maintain a unique work queue id which is used to identify a job in the work queue.User program can use this work queue id to retrieve information about the job.

Producer and consumers

Producer is responsible for converting the user request into a job and submitting it to work queue.

Consumer is responsible for fetching a job from the work queue and processing the user request.

Starting and stopping consumer threads

We create three consumer threads when the kernel module is first loaded into memory. The threads are destroyed when module exits.If module is removed while there are jobs in the queue then before stopping the consumer thread we discard all the jobs in the queue..

Throttling the heavy producer

A producer must not busy wait if there isn't a slot for the new job. Therefore it is put to sleep on the producer wait queue. In this way it is effectively throttled until a consumer consumes a job and makes slot available. Once consumer picks up a job for processing it wakes up a producer waiting on the wait queue..

Consumer Idle scenario

A consumer is put to sleep when idle.When a job is available the producer will wake up a consumer waiting on the wait queue to pick up the jobs.

Storing results

```
struct result {
        int queue_id;   /* Work queue ID. */
        int pid;        /* PID of the process which submitted the job. */
        char *res;      /* Result of the operation(String). */
        int reslen;     /* Length of the result. */
        struct result *next; /* Pointer to the next result. */
};
```

Functions:

```
add_result /* For adding results to the result queue. */
char *enumerate_one_result(int id); /* returns a char* for result of the given queue_id */
char *enumerate_results(void); /* return a char* for all the results */
```

Similar to the work queue we also have a result queue. Whenever a job chooses the result be saved in the memory it calls *add_result* with all the required info. There is a limit on number of results that can be stored. If the limit is reached and a *add_result* call is made it removes the oldest result (result at the tail) and then adds the new result provided by the caller of *add_result.*

From the admin module the user can get the result by providing the queue_id. The user can also retrieve the result by using netlink. For security reasons the queue_id and pid of the caller will be validated before returning the result.

Locking mechanisms

Since work queue is a shared data structure, mutual exclusion is necessary. We used mutex locks to achieve it. Mutex Locks are more efficient in this scenario as Critical section involves memory allocation which can trigger an I/O operation which takes lot of time to complete.

We use the same mutex lock for synchronizing add_result and enumerate_one_result, enumerate_results.

Asynchronous callback implementation

We are using below mechanisms to communicate with user program.

Signal

Signalling is used to intimate the user program of completion of job.
we write signal handlers which calls a function that sets the flag on which the user program sleep waits for the completion of the job. Signal can also carry 32 byte of information with it.
In encryption operation we use signals which intimate about the completion of the job and gives the number of successfully encrypted/decrypted files.

Netlink Socket

User program creates a netlink socket connection to kernel and requests to send the result to it via socket (similar to querying a database). At kernel side , a netlink socket is created when kernel module is initialized and released when it is removed.Kernel module processes the user request and writes back the results to the socket.User program reads the result from the socket.

In admin operations we use netlink socket to retrieve all the job ids waiting in the work queue as well as retrieving the results.In checksum operation we use it to send the result to user program.

Encryption/Decryption process

Convert password to key

Our algorithm requires a 32 byte key for encryption. In the function generate_key the password from the user is passed. User should enter a password with characters between 8 and 15. We have two predefined string called SaltL and SaltR. Two temp buffers with length 16 made from the SaltL and SaltR using the password.The temp buffers are hashed(using sha1) independently and then concatenated to form a 32 byte key.

This 32 byte key is used with the AES for encrypting/decrypting the file.

Issue with padding and its solution

Problem: The AES algorithm takes 16 bytes of data block at a time and encrypts it. When we reach the end of the file and there are less than 16 bytes we have to do padding to make it 16 byte block. And during decryption we need to take care of the removing the extra bits that were padded.

Solution: Say k bytes (k < 16) are left in the last iteration.

Case: Encryption

We encrypt a block (16 bytes) one by one. Last chunk of the input file is day k bytes (k<16) pad (16 -k) ‘0’ to the string to make it to the block size. Encrypt the block. Write the block to the result file. Now additionally write k byte of the block to the result file.

Case: Decryption

We decrypt a block (16 bytes) one by one but we delay the writing to the file by one block. When we are the last chunk of the input file(encrypted) , the k bytes from the last decrypted block will be written to the output file and the rest will be ignored.

Checksum crc32c Logic

We initialize a empty crc to ~0 which makes it all ones.After that we use the crc32 function repeatedly for each 16 byte block.At the end of the last block we get the required crc. When compared with the crc from the cksum syscall our value differs.This is because syscall checksum algorithm is not compatible with crc32.c For validation we make sure that every time we get same crc for given file.

Source code tree organisation

Location: /usr/src/hw3-cse506g19/hw3

install_module.sh

A shell script to remove the module and insert it again and does lsmod in between to see if the operations were successful.

Makefile

Makefile for compiling the user programs as well as the module.

kernel.config

Kernel configuration for building the kernel.

xhw3.h

Header file for all the user programs.

xhw3.c

Generic user program. Not actually used.

admin.c

User program for listing the jobs in the queue, remove one or all jobs and retrieving the results of the jobs is implemented here.

user_checksum.c

User program for submitting checksum jobs.

user_encrypt.c

User program for submitting encryption jobs.

user_compress.c

Stub file. Functionality not implemented.

./src:

common.h

Header file for storing job object structs. This header is shared by both user programs and the module.

sys_xjob.h

Common header files and extern functions are declared in this header file.

sys_xjob.c

Producer, consumer, work queues, wait queues & result queue and their related functionalities are implemented here.

sys_xjob_checksum.c

Checksum(crc32c) operation and converting a checksum job object of user to the job object of the kernel is implemented here.

sys_xjob_encryption.c

Password to key conversion, encryption/decryption operation and converting a encrypt job object of user to the job object of the kernel is implemented here.

sys_xjob_netlink.c

Netlink channel between user and kernel is implementation here. Based on the input message the response (output message) is sent.

Usage Information

Operations are categorized into two modes.

Admin Mode

1. To know status of a particular job - Waiting/Running/Completed
2. To get result of a particular job operation(checksum)
3. To get a list of all jobs in the queue.
4. To get a list of all results for checksum operation.

We use the job id returned by kernel to user program to query about the status of particular job..

User Mode

This mode has operations to perform checksum and encryption of given file names.

Admin Mode Operations syntax

- List a particular job
  ./admin -l [pid]

- List all jobs in the queue
  ./admin -l all

- Remove a particular job from queue
  ./admin -r [pid]

- Remove all jobs in the queue
  ./admin -r all

- List all results
  ./admin -j all

- List result of a particular job
  ./admin -j [pid]

Checksum operations syntax

- Wait for the result from kernel. Result is written by kernel into the socket connection created between it and the user program.

```
./user_checksum  -w  [filename(s)]
```

- Write the result to file. User program exits and doesn't wait for the kernel to return results. Result file is created under /tmp/results/[jobid].chksum
```
./user_checksum  [filename(s)]
```

- Save the result into kernel buffer so that it can be queried by admin later
```
./user_checksum -s [filenames(s)]]
```

Encryption/Decryption operations syntax

Encrypted files are created in directory of the input file with *.encf* extension. Similarly decrypted files are created in directory of encrypted file with *.dncf* extension.

- Encryption without waiting.
```
./user_encryption -e -k [key] [filename(s)]
```

- Decryption without waiting.
```
./user_encryption -d -k [key] [filename(s)]
```

- Encryption with waiting for response
```
./user_encryption -w -e -k [key] [filename(s)]
```

- Decryption with waiting for response
```
./user_encryption -w -d -k [key] [filename(s)]
```

- Waiting mode by default stores the result of the job in the job queue. To submit encryption/decryption job without waiting and still save the result in result queue for future reference we use -s as follows.
```
./user_encryption -s -e -k [key] [filename(s)]
./user_encryption -s -d -k [key] [filename(s)]
```

Problems faced and their solutions

Scenario 1: In the exit code we called wake_up_consumer and then called kernel stop. The consumer wakes up and finds the qlen is still zero and sleeps again. As a result kernel_stop operation is waiting for the consumer thread to exit which is sleeping on the wait queue.

Solution: We set the qlen variable to QMAX+1 and then wake up the consumer. The consumer loops and finds that thread_stop was called and exits cleanly.

Scenario 2: When we introduced two consumer threads, In the exit code we called (wake_up_consumer, thread1_stop) and (wake_up_consumer, thread2_stop). As it is not necessary that on the first wake up the thread1 wake up. By the time kernel stop signal reaches the thread, the thread goes back to sleep and the exit function is stuck waiting.

Solution: We set the qlen variable to QMAX+1 and then wake up the consumer and then make the thread sleep using msleep for 2 seconds.When we introduced a third thread we didn’t have any problems.

Scenario 3: In checksum and encryption if user provided filename with path relative to pwd the kernel thread failed to find the file in its pwd (kernel thread work in / dir).

Solution: We used realpath api to convert the user supplied file names to absolute pathnames before passing them to consumer thread.

Limitations

- The number of results shown in Retrieve results operation of admin are limited by MAX_PAYLOAD size.
- Checksum files are created in /tmp/results/
- If the ID of the job is less than the queue_ID of the head element, the job is assumed to be completed.