CSE 506 : Operating Systems
Homework : 03
Asynchronous and Concurrent Processing

Gaurav Dugar

May 12, 2014

1 Introduction

This homework includes implementation of Asynchronous and Concurrent Processing, Kernel Locking, Efficiency, and Callbacks. I have implemented a workqueue structure which stores all submitted jobs (such as Checksum, Encryption or Decryption of a file), and processes them as possible. When the CPU (consumer) is busy, then the jobs are added to the queue. The user is notified accordingly. The details of implementation are in sub-sections below.

2 Make and Installation

The syscall and user programs are compiled using make utility in hw3. The two optional arguments that CAN be passed are:

1. Consumers: It can set the number of consumers processing the jobs.
2. Queue_size: It is the queue size tp hold the jobs.

```
make ARGS='-DCONSUMERS=3 -DQUEUE_SIZE=1024'
```

These are optional parameters.

To submit a checksum job,

```
./doCheck <filename> <algo_name>
such as
./doCheck 123 md5
```

To submit a encryption job,

```
./doCrypt <algo> <filename> <key> <flag for enc/dec>
./doCrypt 'cbc(aes)' 123 key_with_size_16 1
```

3 Workqueue

The workqueue has been implemented in sioq.c file. The basic functionalities are available in linux/workqueue.h. Some detailed implementation is also available in unionfs code.

The basic idea is to use an array of fixed size and insert jobs in it. Use it as circular queue (by providing jobs next available location in a circular manner). The workqueue also allows us to set the value of maximum consumers. These consumers can do parallel processing (simultaneous job execution).

```
struct sioq_args {
        struct completion comp;
        struct work_struct work; //work structure
        int err; //error code
        int pid; //pid of work
        int id; // work id of work
        void *ret;
        struct path pwd; //path of infile
        int complete; //flag to indicate completion of work
        int type; //type of work
        union {
                struct checksum_args *checksum_arg; //checksum arguments
                struct crypt_args *crypt_arg; //encryption/decryption arguments
        }; };
```

When a job is submitted, it is assigned a work-id, which is basically the index of array location to which that work is linked. If there is no available space in the queue, then that user request is put in running state, where it eventually gets an id, as soon as any job is finished.

Every job has some structure which stores work_type, job_type and void pointer to store configuration.

```
struct jobs {
        int work_type; //SUBMIT, LIST, CANCEL
        int type; //CHECKSUM, DECRYPT, ENCRYPT
        void *config; };
```

4 Checksum

A checksum or hash sum is a small-size datum computed from an arbitrary block of digital data for the purpose of detecting errors which may have been introduced during its transmission or storage.

Here, we read the content of input file and produce the checksum of it. The checksum is displayed back to the user using Netlink. (explained later)

The structure used for storing arguments of checksum is as follows:

```
struct checksum_args {
        char *algo; //algorithm for checksum
        char *file; //input file
};
```

To calculate checksum, many algorithm can be used such as MD5, SHA-1, SHA-2, Simple file verification, SYSV checksum, Damm Checksum etc. This implementation required user to enter his choice of algorithm with the input file.

5 Encryption/Decryption

Encryption is the process of encoding messages or information in such a way that only authorized parties can read it. For encryption, we need a key, algorithm, input file and a flag (to decide whether to encrypt or decrypt).

The structure to store the crypt arguments is as follows:

```
struct crypt_args {
       int keysize; //size of key
       char *file; //input file
       char *algo; //algorithm to be used
       char *key; //the key
};
```

The cryptoAPI has many inbuilt functions that can be used for such implementation. The crypto.c and crypto.h are the files where actual processing is being done. We calculate the encryption using calc_hash() function and then store it in same file. To avoid any intermediate error, I have created a temporary file to store output and after successful execution the content is copied back.

The encryption is done in blocks. The size of block depends upon the algorithm used. So, the encryption/decryption has to be done in such blocks. If required padding is added to the block to fill it.

6 User side programs : Modularity

User side programs to call syscall include submitting a job, cancelling a job and listing all jobs in the queue. To provide modularity, I have created different files for every operation, namely user_cancel_job.c, user_list_job.c, crypt.c and checksum.c.

The file sys_ds.h contains the drfinition of all structures required by user program to store arguments and send to the syscall. The result is returned by syscall and evaluated by the user program.

7 Submitting a Job

To submit a job, I have created some executables which basically link the netlinkuser.o and (function).o. For example, to submit a checksum job, we need doCheck executable.

The arguments are validated at user level and wrapped in the respective structure to be sent to syscall.

7.1 Checksum

The checksum job requires user to input a valid filename and algorithm to be used. The program returns the output as checksum value or appropriate error.

Filename: crypt.c, Executable: doCrypt

7.2 Encrytion/Decryption

The encryption/decryption job requires user to input following information.

1. A valid filename
2. algorithm to be used
3. flag (0 - decrypt, 1 - encrypt)
4. key (its size has to be algorithm specific)

Filename: checksum.c, Executable: doCheck

8 Listing all Jobs

To list all jobs currently in the queue, we have user_list_job.c file. It takes no argument and returns output of process id, work id and Type of the jobs or appropriate error message.

9 Cancel a Job

Cancelling a job requires its ID. To cancel, we have user_cancel_job.c file, which takes one argument as its ID. It returns the output as successful deletion or appropriate error message.

10 NetLink

Netlink is used to transfer information between kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules.

10.1 How It Works

The kernel side code for netlink is in netlinkkernel.c and user side code in in netlinkuser.c files. When the syscall is initialized, we also initialize netlink. Whenever a job is submitted or request from user arrives, the kernel send message using netlink socket via send_msg() function of netlink. At this time, the user program should be listening to that socket using get_msg() function od netlink. The send_msg() takes input as process id of user program and the message.

The method used to send the message in kernel code is as follows:

```c
static void __call_send_msg(struct sioq_args *args, char *msg)
{
	while (!args->complete)
		schedule();
	schedule();
	msleep(100);
	send_msg(args->pid, msg);
}
```

It basically waits for the work to be completed, and then send the message to the user. Similar method is used to send any error message to the user.

10.2 Integration with Submit, List and Cancel

The message returned by the netlink is:

1. SUBMIT: Job successful submission message and Checksum value or appropriate error message.
2. SUBMIT: Job successful submitted and success message.
3. LIST: The pid, work_id and type of all jobs in the workqueue.
4. CANCEL: the successful cancellation message or error message.

11 Features

1. Customizable size work queue, where consumers count can also be set implemented.
2. Listing of all jobs presently in the queue.
3. cancelling a job in the queue.
4. Able to calculate checksum by reading contents of the file and using algorithm as specified by the user.
5. Successful implementation of netlink to pass message from kernel space to user space.
6. Modularity, by having different files for their jobs.
7. All memory leaks patched.
8. Handled all error messages and returned appropriate message tp the user by netlink and by return value.
9. All user data is checked before being used and copied in kernel space. (putname, Verify Read/Write etc)
10. NOTE: Compression type jobs are not implemented.

12 File Operations

The file operations such as open, read, write, close and sync are adde in the file.h and file.c files. This code is taken from homework 1.

13 References

1. http://lxr.free-electrons.com/source/include/linux/workqueue.h?v=3.2
2. http://man7.org/linux/man-pages/man7/netlink.7.html
3. http://www.logix.cz/michal/devel/cryptodev/cryptoapi-demo.c.xp
4. http://www.opensource.apple.com/source/libxslt/libxslt-7/libxslt/libexslt/crypto.c
5. http://stackoverflow.com/a/17606674
6. http://lxr.free-electrons.com/source/fs/fs_struct.c?v=3.2
7. http://stackoverflow.com/questions/15215865/netlink-sockets-in-c-using-the-3-x-linux-kernel