CSE-506: Implementing Asynchronous Producer/Consumer Queue

Muhammad Ali Ejaz
Stony Brook University - 110559131

Abstract

Certain processing of data files, such as encryption or compression, consumes a lot of CPU. But it also requires a lot of I/O activity, because the file's data buffers have to be copied between user and kernel levels, producing expensive data copies and context switches.

An alternative is to read the files in the kernel, do all the CPU-heavy processing on them, and then write it back out directly. This can eliminate expensive user/kernel data copies and context switches.

Another problem is that such processing takes a long time, no matter whether you do it in the kernel or user level. We don't want user programs to wait too long, and we don't want to wait for system calls that may take a long time to finish either.

So an additional solution is to perform this processing asynchronously. The user program will "submit" a request to the kernel to process a file, say, encrypt or compress it. The user program will NOT wait for the request to be processed, however. The kernel will accept the request and queue it for processing, for example using a FCFS policy or any other priority based policy. That way, the user program doesn't need to block waiting for the request to finish.

This project is to implement such a mechanism in the kernel along with using netlink to propagate errors and job status to the userland.

1 Introduction

A new system call *submitjob(void *)* has been implemented in Linux 4.0.9+ Kernel of CentOS release 5.11 (Final) Operating System. The system call has been implemented as a loadable kernel module and is called mod_submitjob.

This system call takes in the job requested by the user, performs some basic sanity tests on user inputs and then puts the job in Workqueue based on priority. The user can choose to wait for the result of the job requested or decide to look at the result later in dmesg.

The jobs are run asynchronously and the result is communicated back to user program via netlink. The netlink wait happens in a thread as shown in Fig 1 and as such the user program can continue doing it's work and is not blocked waiting on the jobs result.

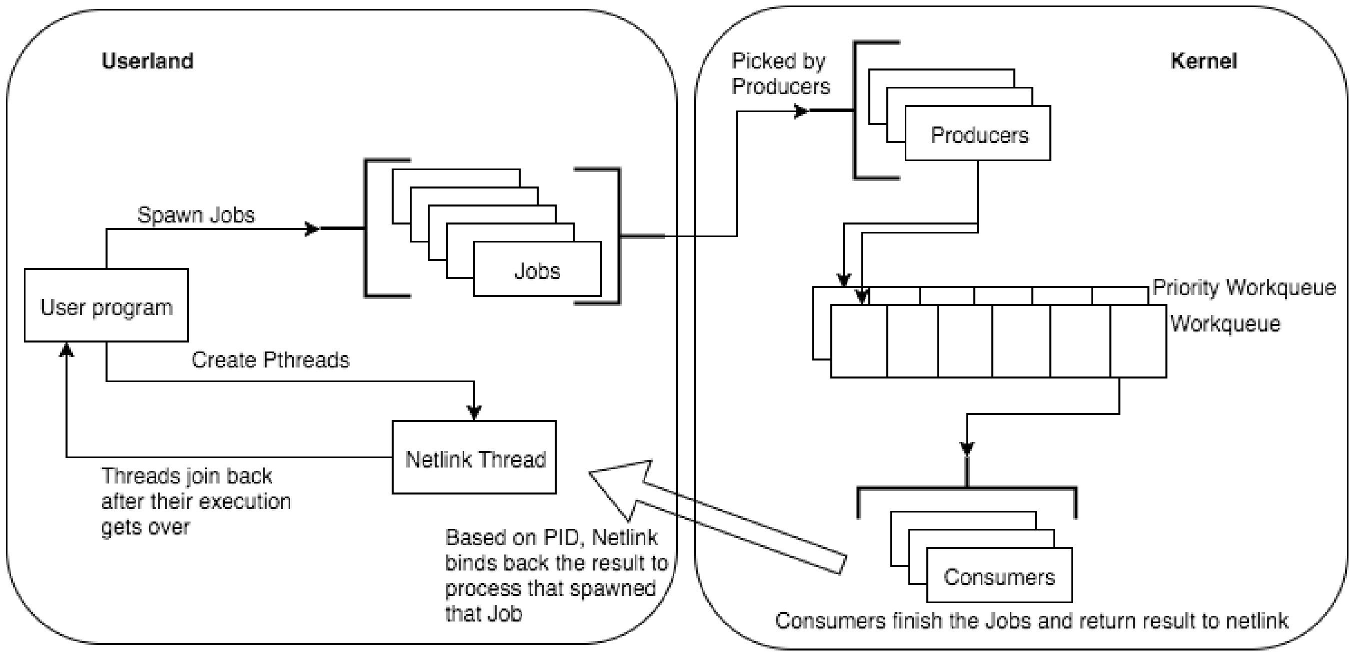
Userland
Spawn Jobs
Jobs
User program
Create Pthreads
Netlink Thread
Threads join back after their execution gets over
Based on PID, Netlink binds back the result to process that spawned that Job
Picked by Producers
Producers
Kernel
Priority Workqueue
Workqueue
Consumers
Consumers finish the Jobs and return result to netlink

Fig 1: Flow of control for the asynchronous Job execution

If the user program finishes earlier than the Netlink thread, the user program must wait for the thread's execution to get over. If not, the thread will die with the main program's exit. To ensure that all spawned threads are complete, a pthread_join(...) is called at the end of user program. This call waits for the mentioned thread to get over and then exits from the user program.

If the user program didn't have any other work to do and didn't care for instant job results, the user might not want to wait at all for the results. In such scenarios, the user can set the -w flag. This flag when set will turn off the wait flag and the user program will not wait for the job result. The job result can be checked in dmesg whenever required.

On the kernel side, some sanity checks are done on the user input and then the jobs are put on producer/consumer queue for processing. There are two queues, a normal or low priority queue and a high priority queue. Based on Jobs priority, the jobs are placed on respective queues.

The consumer, picks up the Jobs from the queue and calls in appropriate methods based on the job type. After the respective methods get over, the consumer puts in the job result in dmesg and sends a unicast message to the netlink user waiting for that job result. The decision, on who gets the unicast message is decided by the PID of the user program that initiated the system call for the given job.

2 Important File Details

All the sources files are present in the location hw3-mejaz/hw3. Please see below for the file description:

submitjob.h This is a header file shared between the userland and kernel space.

jobs.h This is a header file shared among the files in kernel space.

post_job.c User Level Program which acts a driver program for the system call and calls *submit(void *)* system call for testing purposes.

netlink.c This file contains user level code for netlink. This has methods to bind to a socket and receive messages from kernel.

sys_submitjob.c This is the Kernel Module Program which implements the system call *submitjob(void *)*. This file contains the producer/consumer code.

shared.c This file contains the common code like validity checks on an opened file and is shared by all kernel code files.

checksum.c This file contains the code to generate checksum for the given file. When the producer gets a job to compute checksum, it calls methods defined in this file.

concat.c This file contains the code to concatenate the given input files. When the producer gets a job to concatenate files, it calls methods defined in this file.

xcrypt.c This file contains the code to encrypt/decrypt the given input file. When the producer gets a job to encrypt/decrypt files, it calls methods defined in this file.

xpress.c This file contains the code to compress/decompress the given input file. When the producer gets a job to compress/decompress files, it calls methods defined in this file.

Makefile Makefile is used to 'make' the program and generate the executables. The make command compiles both kernel code and the user-level program.

install_module.sh This file removes and installs this loadable kernel module. This file only executes the commands to remove previous module and install the new one. This file doesn't check for any errors.

kernel.config The kernel that has been used for the project has first been minimized. The resulting Kernel image obtained through this config is 2.7MB. The image can be found in hw3-mejaz/arch/x86/boot/bzImage.

3 Implementation Details and Design Decisions

3.1 Userland

The user program posts jobs asynchronously to the kernel for processing. Currently Encryption, Decryption, Compression, Decompression, Checksum computation, Concatenation of multiple files, Listing of Jobs on queue, Removing a job from queue and Swapping of Job priority is supported through the following flags:

-e encrypt the given file

-d decrypt the given encrypted file

-s shorten or compress the given file

-r restore or decompress the given compressed file

-c compute checksum of the given file

-m merge or concatenate the given input files

-l list the queued jobs waiting to be processed

-u undo or remove a previously put job

-t tweak or swap priority of previously put job

The above functionalities might require parameters like algorithm to be used, password, etc. These should be passed from the user using the following flags:

-a algorithm to be used for encryption, decryption, compression, decompression and checksum computation

-p password to be used for encryption and decryption. Password should be at least 6 characters long

-i ID of the Job to be deleted or whose priority needs to be changed.

-w do not wait for the job response

-P priority to be used as high for the given job default priority is 'no priority' for all the jobs except checksum computation whose priority is always high

-h display this help and exit

All these options for parameters are not necessary to be provided and have some *default values*. Default values are provided to make it easier for users who might not be sure of the values to be set for these options. Currently, the algorithm for encryption/decryption defaults to *'aes'*, for compression/decompression it defaults to *'deflate'* and for checksum it defaults to *'md5'*.

The following algorithms are supported by the kernel module as of now:

encryption/decryption *aes, des, blowfish, twofish, des3_ede, seed, anubis, khazad, xeta, xtea, tea, arc4, cast5, cast6, camellia, tnepres, fcrypt* and *serpent*.

compression/decompression *deflate, lzo, lz4* and *lz4hc*.

checksum *md5, md4, sha1, crc32, crc32c, rmd128, rmd160* and *rmd256*.

The jobs are run asynchronously at a later point in time and might not run in the location where the user issued the task from. As such the relative paths provided by user would result in a File Not Found error. One way to fix this would be to provide absolute paths for the task. But this might get cumbersome and the dependency for the correct path depends on the user. As such, the solution that I chose was to allow users to give relative/absolute paths (whichever is preferred by user) and instead the driver program takes care of computing the real path of the file before the path is passed to the system call. While computing the realpath(...), the input files' existence is also checked. The existence of output file is not checked as the output file might not be existing and might have to be created with the name provided by the user at a later point in time.

By default all the user jobs' results are printed on stdout/stderr. The user requests for the job and then continues doing his/her work. This is implemented with the help of netlink.

The decision to choose netlink for the communication between userland and kernel space was directed by the following points:

1. It is a nontrivial task to add system calls, ioctls or proc files for new features; we risk polluting the kernel and damaging the stability of the system.
2. Netlink socket is simple to implement and the kernel module and application can talk using socket-style APIs.
3. Netlink is asynchronous because, as with any other socket API, it provides a socket queue to smooth the burst of messages.
4. The system call for sending a netlink message queues the message to the receiver's netlink queue and then invokes the receiver's reception handler.
5. The receiver, within the reception handler's context, can decide whether to process the message immediately or leave the message in the queue and process it later in a different context.
6. Unlike netlink, system calls require synchronous processing. Therefore, if we use a system call to pass a message from user space to the kernel, the kernel scheduling granularity may be affected if the time to process that message is long.
7. NetLink sockets can be used for Multicast although for this assignment I have used only Unicast.

Before issuing the system call, the user program binds to the netlink socket by using its PID. This PID would be used on the kernel side to send back the response. After the system call is made a new *Pthread* named *rcv_msg_thread* is created. This thread listens to the socket and when it gets a message for itself (identified by the PID) it prints the message on stdout/stderr. In the meanwhile the user program continues to do some other task. Once both the user program and this thread's execution gets over the user program exits. To ensure the main user program doesn't exits before the *rcv_msg_thread's* execution is over, a *pthread_join* call is made before exiting the main user program. This call ensures that this spawned thread's execution is complete.

Having said that, the user can decide to not wait for the job result. This might be useful in case when the user just wants to issue the job and doesn't care for the result at that point in time. The user can issue the job and exit the program and later check the result in dmesg.

The netlink message is a structure containing a message and a return code. This was done to print human readable messages sent by kernel on to the stdout/stderr and at the same time use perror to print the errors corresponding to error numbers in case of failures.

3.2 Kernel Space

The producer/consumer behavior is implemented using Workqueue. There are two queues, one is a normal or no priority queue while the other is a high priority queue. This is done to enable executing a high priority job earlier than all the queued jobs. If however, there are jobs in high priority queue waiting for the CPU, the job would need to wait. The jobs are processed in FCFS order. The process behind Workqueue is shown in Fig2.

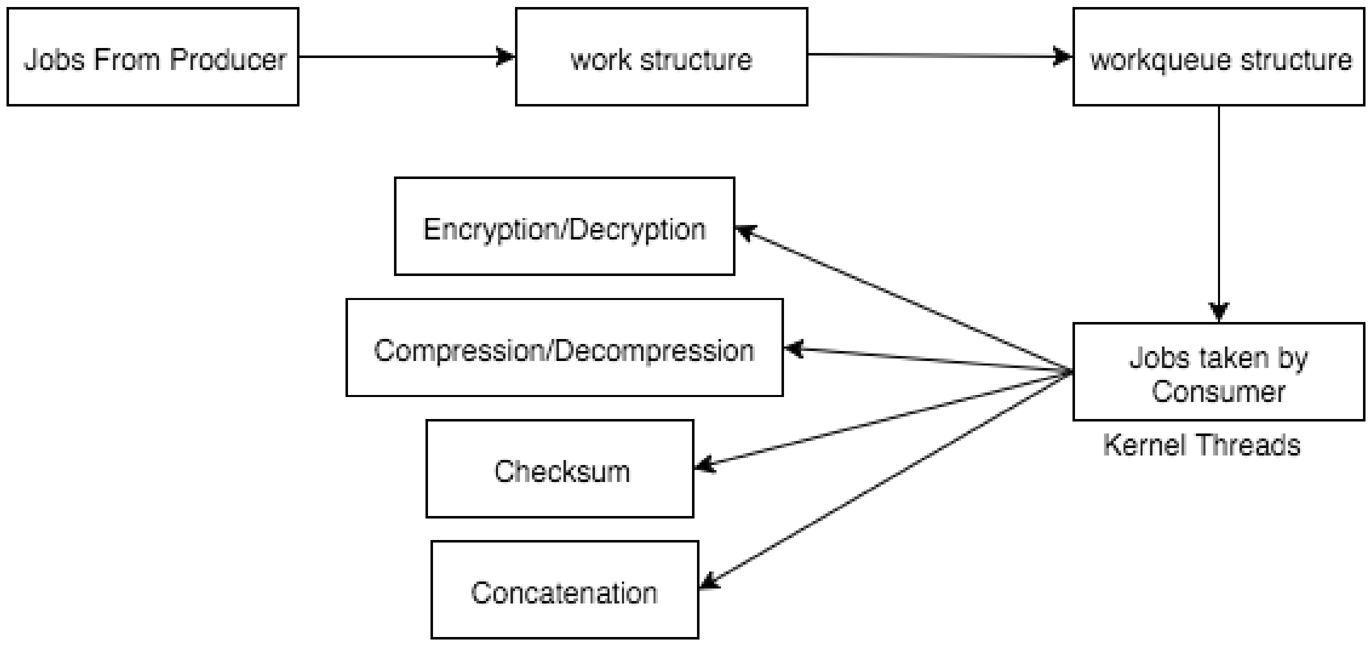

Fig 2: The process behind work queues

Both the normal and high priority workqueues are created and destroyed when the kernel module is loaded and unloaded respectively. At the same time two atomic counters one for each of the queues is initialized to 0. These counters are used to track the size of the workqueue. Also, since it's atomic, the increment/decrement of these values are atomic operations.

Along with the two workqueues, a List is maintained to keep track of all the jobs in both the queues. This list is used to perform operations on workqueue. The operations allowed to the user are addition of job, deletion of job, listing of all jobs with priority and swapping of priority of the jobs. These operations are facilitated with the help of this List.

The operations on queue are done with the help of List because it's easier to iterate over the list with just an ID to find the job that the user wants to remove/swap priority of. The listing is of jobs require just iterating over this list. For swapping the priority, the list need not be changed; job reference is taken from the list, removed from the old queue and put on the new queue. Hence using the list facilitated in faster and easier operations.

Workqueues can be used to limit the number of consumer threads and all workqueue operations happen through it's own locking mechanisms. All the operations on List, however, is done through explicit locking mechanism to ensure the List is consistent with the workqueues. Currently I've limited the number of consumer threads to 5 and the total number of jobs allowed on the workqueue to be 20.

There was no need of putting the consumer to sleep or throttling the producer since Workqueue handles these producer/-consumer behavior implicitly.

The above points along with workqueue being a widely used mechanism in kernel influenced the decision behind choosing workqueues.

Apart from locking the List operations, the files being operated on by the consumer threads are also locked. This was done to ensure that no two operations modify the same file. Although Linux kernel ensures writes happen through a lock implicitly, I still chose to implement my own locking mechanism, because the operations I implemented did file operations through page size. And an operation could have possibly returned stale data.

The locking of file is done by opening a file named f1.lock for a file with name f1. Before creating this file, vfs_stat is run on the file and it's output is checked to see the existence of this lock. If the lock is with some other process, the thread is put to sleep for sometime. When the thread wakes up and the lock was still not released, the thread goes back to sleep. Every time the thread goes back to sleep, it sleeps for double the time than it's previous sleep cycle, that is, the sleep time increases exponentially. This is done to prevent too many frequent checks for the lock. I found this to be the advised behavior on many forums.

This obtaining of file lock is done within a mutex lock to avoid another process from taking the lock in the race condition. After checking the vfs_stat, if the status implies that the lock can be taken, another process shouldn't take the lock in between this check and obtaining of lock. However, if the status implies the lock to be taken by some other process, the mytex lock is release and then the process is put to sleep. The code is written to ensure no process goes to sleep while having the lock.

Also, the case of deadlock is avoided by releasing previous locks if the current lock is not available. If a process is able to take a lock on its input files but not on its output file, the process should release the lock on its input file before going to sleep. This ensures that no deadlock scenario arise.

The netlink message from kernel is sent when the consumer finishes its execution or the job gets cancelled. Sending the message on job cancel is required because the Pthread waiting for response will keep on waiting for the result while the job no longer exists.

4 Steps to use this Kernel Module

make: To generate the module and other binaries, make command is used. For the first time, a make on entire hw3-mejaz is recommended. Thereafter, if any code changes are done to this module, a make inside the hw3-mejaz/hw3 would do.

sh install_module.sh: This script located inside the hw3-mejaz/hw3 path can be used to load the generated module.

sh post_job: Thereafter *sh post_job* or *./post_job* with all the options specified before can be used.

References

[1] The format of help is made referring to *'man cat'*.

[2] The netlink usage is inspired from http://amsekharkernel.blogspot.com/2012/01/what-are-ways-of-communication-bw-user.html

[3] The workqueue usage is inspired from http://www.ibm.com/developerworks/library/l-tasklets/ and work_queue.pdf slides from http://gauss.ececs.uc.edu/Courses/c4029/lectures

[4] The encryption/decryption algorithms are inspired from ceph_aes_encrypt(...) and ceph_aes_decrypt(...) in http://lxr.fsl.cs.sunysb.edu/linux/source/net/ceph/crypto.c

[5] Checksum generation algorithm is inspired from symlink_hash(...) in http://lxr.fsl.cs.sunysb.edu/linux/source/fs/cifs/link.c

[6] Compression/Decompression algorithms are inspired from ubifs_compress(...) and ubifs_decompress in http://lxr.fsl.cs.sunysb.edu/linux/source/fs/ubifs/compress.c