Asynchronous Job Scheduler

CSE-506
Operating Systems
Spring 2022

Homework 3 Report

Kushaal Hulsoor(114776851)
Rama Aishwarya Kondeti (114777531)

1. Overview

We implemented a Loadable Kernel Module (LKM) which, on being installed, would set up two workqueues (normal and high priority), and one system call named “async_sys” which would allow users to perform various operations on the queues. Users can also submit jobs of different types, each of which would perform a specific task asynchronously. An option can be provided to either poll the job end status or write it into predefined files. Further details regarding design and implementation are discussed in the following sections.

2. Design and Implementation

Firstly, when the module is installed, we allocate two workqueues (normal and high priority). Linux’s native workqueue API has been used for this. High priority workqueue was created simply by sending the WQ_HIGHPRI flag. More details on how this works can be found here(https://www.kernel.org/doc/html/latest/core-api/workqueue.html#flags). On a simpler note, this flag allows the jobs to be queued to hipri work-pool and run with elevated *nice* levels.

Adding to this, we also create a netlink socket to send job execution status from kernel to userland. A directory “/async_job_outs/” is created if it doesn’t exist, to store all the files that store job execution status.

The design can broadly be divided into three components:

1. Userland (Fig. 2.1)
2. Syscall in kernel (Fig. 2.2)
3. Workqueue in kernel (Fig. 2.3)

Let us discuss each of them.

2.1 Userland

Userland component is responsible for taking input from the user as command line arguments (all options and semantics discussed later in the document), parse the input, make necessary validations and send them to the kernel using the *async_sys* system call that we created. The arguments are sent as a part of a struct named *job_item_user*. Before sending filenames to the kernel, we pass them through the realpath() function first, to convert relative paths to absolute paths. This is done because the workqueue are asynchronously run in a different context and have no access to the current working directory path from where the job was triggered.

For submitting a new job, we use the current process id which is set as the job id and displayed to the user. This id can be used later to perform operations on queue.

If the user chose to poll the job completion status and return values, a user level thread is spawned before making the system call. The thread’s job is to wait for the job execution status and display it on the output console. On the other hand, if the user chooses file-out as the out-mode, the system call is triggered and the user process is terminated without waiting for the job execution status.

2.2 Syscall in Kernel

To start with, the syscall first translates userland memory addresses to kernel addresses and performs necessary validations on it.

This component is mainly responsible for performing different operations on the queue. Operations include the following:

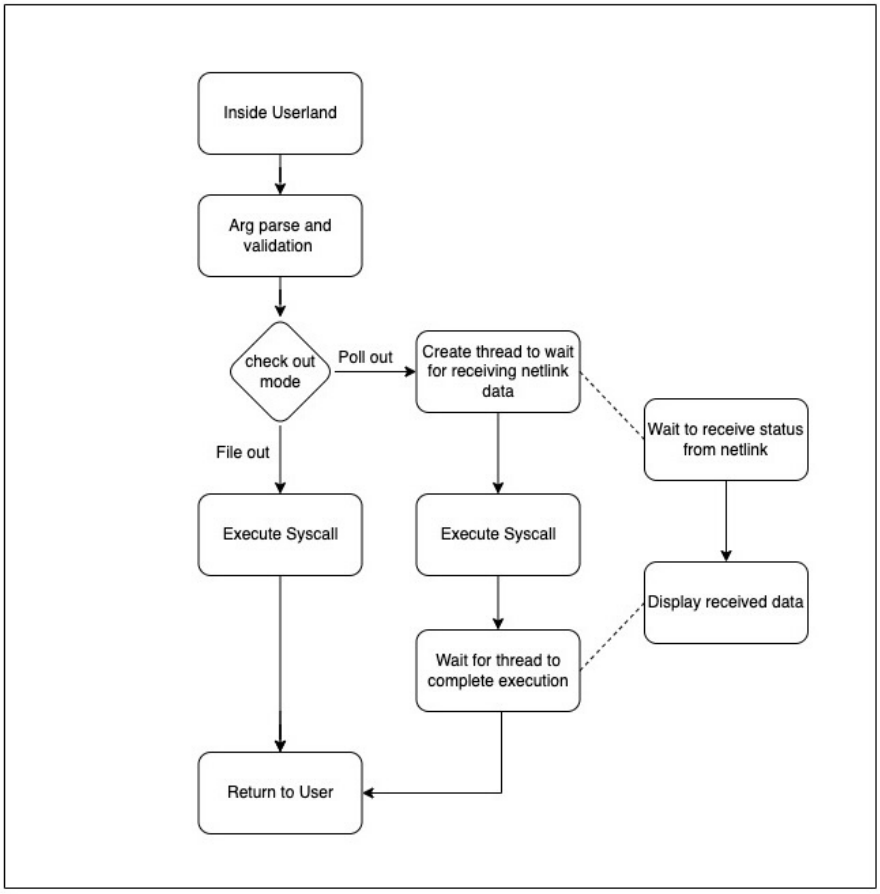

Fig. 2.1 Userland Component Flow

a. Insert job
b. List jobs
c. Delete job
d. Switch job priority
e. Show job status

Although we use workqueues to push the jobs for execution, we maintain a separate queue for each workqueue, to store details about each job, which would help us with performing all the queue operations. Proper **locking** is done whenever we manipulate this queue as this acts as a critical section that is accessed by all the worker threads.

All job operations except insert first check if the job belongs to the user that triggered it using the **uid** that is stored with every job when it is inserted. Other than root user, no user can be able to view or operate on the jobs that are owned by other users.

To **insert** a job, we allocate a node to store the job details and enqueue it to our queue right before pushing it to the workqueue based on the user chosen priority. We also store the work_struct of the job within our queue to help us cancel the work if required later on.

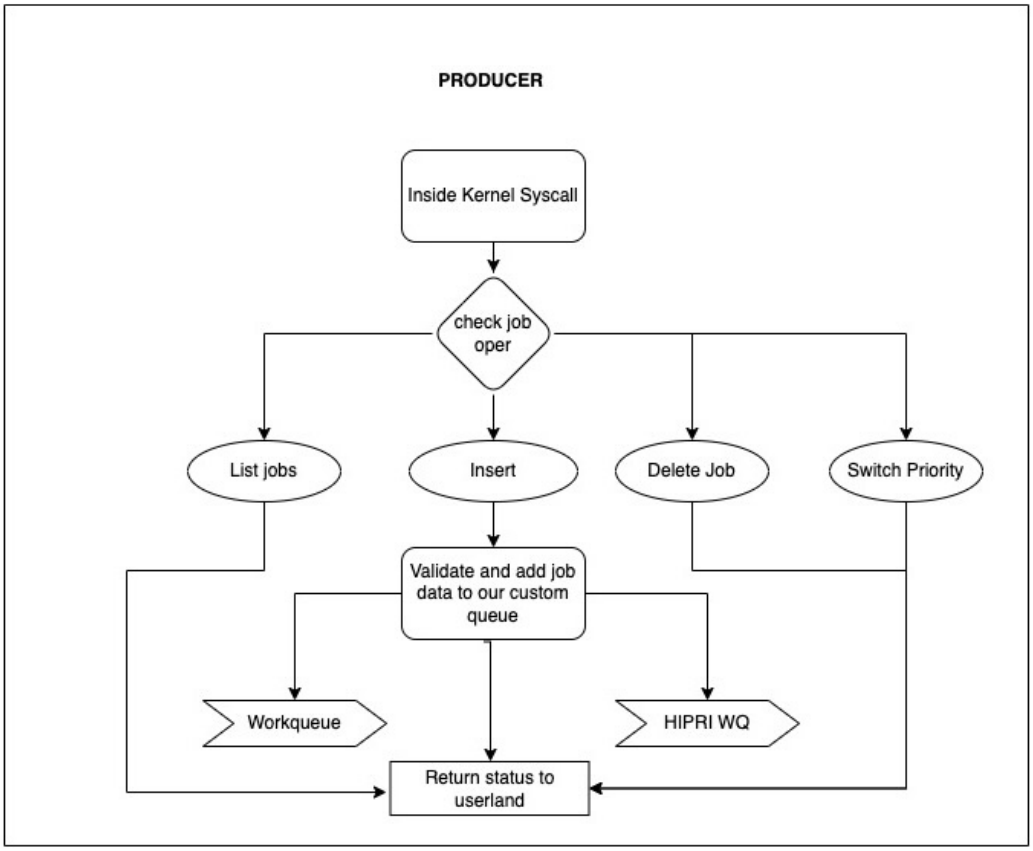

Fig. 2.2 Syscall in Kernel Flow

Regarding **throttling**, linux workqueue's documentation(https://www.kernel.org/doc/html/latest/core-api/workqueue.html#max-active) states that throttling has been handled and controlled internally. We can manipulate the max number of workers that can run at once in a cpu by changing the *max_active* parameter while allocating a workqueue. We have chosen to go with the default value.

To **list** existing jobs, we iterate over our queue and store the nodes contiguously in the user memory using copy_to_user(). We also require the priority type (low or high) from the user, based on which, we choose which of our queues to iterate.

To **delete** a job, we search for the job using job id, which if found, is first canceled from the workqueue and then deleted from our custom queue. In case the job is already running, the job cancellation would fail by returning an error.

To **switch** priority, the job is fetched from one queue, canceled and queued into another queue. If the job is already running, an error would be returned.

To **show** job status, the jobs list is iterated and displays the job details if found, else an appropriate message is displayed.

2.3 Workqueue in kernel

Every worker thread is responsible for handling a single job. The job node in each thread is fetched using container_of macro.

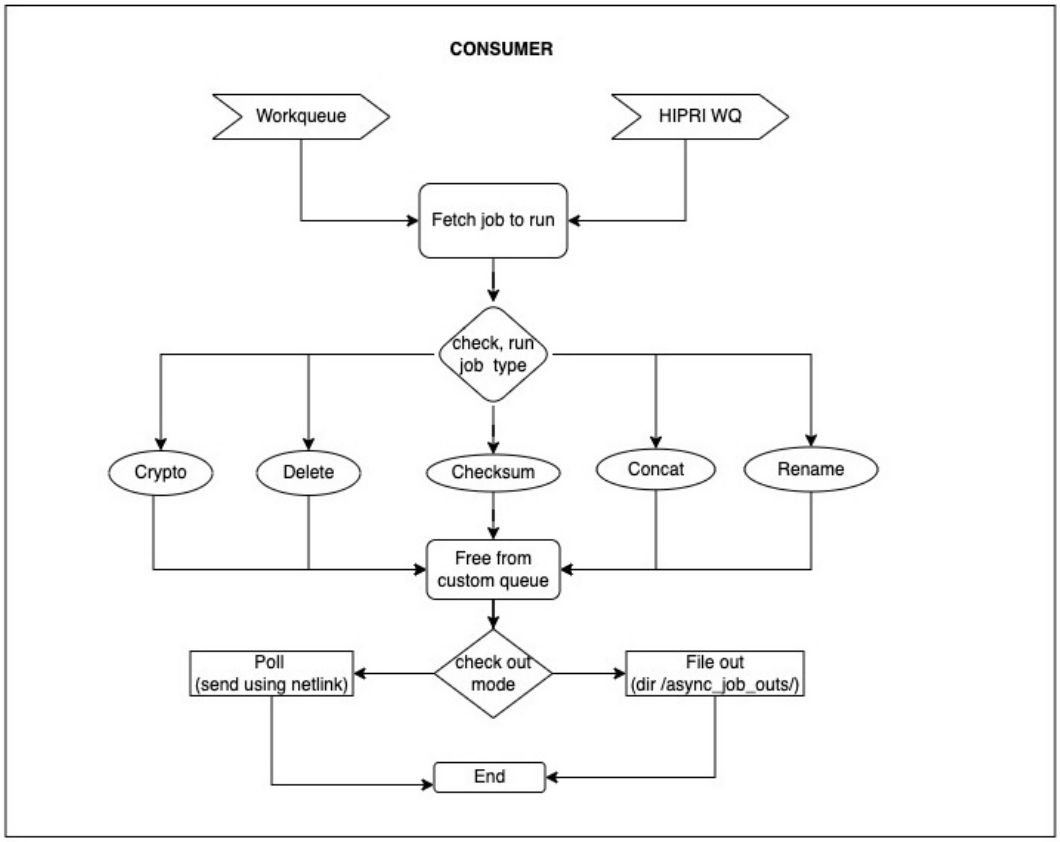

Fig. 2.3 Workqueue in Kernel Flow

This component's main functions are:

a. perform the required job
b. notify the job error status (poll or file-out),
c. dequeue the job from the custom queue,
d. free the memory allocated by the node and exit.

Currently, there are six jobs supported by our module:

a. Encrypt / Decrypt
b. Delete multiple files
c. Rename multiple files
d. Checksum of a file
e. Compression / Decompression
f. Concatenate multiple files

Let us discuss each in detail.

2.3.1 Encrypt /Decrypt

We are using the kernel crypto API's *ctr(aes)* algorithm for encryption and decryption. This is to avoid complexities of padding the data block as required by other algorithms. The input file is opened, encrypted incrementally by allocating a PAGE_SIZE buffer and stored in an output file. The same logic is used for decrypting as well, but the input password is first checked for a match. We store the hashed password in the beginning of the output file to achieve this functionality.

2.3.2 Delete multiple files

We make necessary inode locks for the parent directory, and unlink the file using vfs_unlink, followed by unlocking the parent. This operation is performed for every single file received in the input file list, one after the other.

2.3.3 Rename multiple files

We make necessary inode locks for the parent directories of both source and destination files, and perform using vfs_rename, followed by unlocking the parent directories. This operation is performed for every single set of files received in the input file list, one after the other.

2.3.4 Checksum of a file

We use linux's native crc32() which computes cyclic redundancy checksum for the file's content.

2.3.5 Compression / Decompression

We have multiple types of algorithms provided by the crypto lib, we have selected the deflate algorithm from the crypto lib. When the consumer receives a job for compression we first allocate the crypto with the "deflate" algorithm which in turn is used to perform compression. Once the bytes are compressed we are ready to write the compressed output to the out file from the job struct. We also store the size of the input file before storing the compressed bytes into the output file.

For decompress job , we first allocate the crypto with the "deflate" algorithm and read the number of bytes in the starting of the file , after reading the compressed bytes we pass them to the *crypto_comp_decompress* which returns decompressed bytes , we then initialize a buffer with the size stored above and write to the output file.

2.3.6 Concatenate multiple files

We receive a list of file names and an output filename. Each input file is opened and written into the output file one after the other, followed by closing all the files and returning.

3. Files and their details

Below is a list of all important files and a brief description about each of them.

a. **Makefile:** compiles and generates binaries for project files.
b. **install_async_sys.sh:** script to make and install our module.
c. **uninstall_async_sys.sh:** script to clean and uninstall our module
d. **async_sys.c**: install modules, syscall and queue operation logic for kernel.
e. **jobmanager.c**: user level logic to operate on queues.
f. **common_symbols.h**: common variables to be used between kernel and userland
g. **checksum.h**: logic for computing checksum.
h. **compresslib.h**: logic for compression and decompression implementation.
i. **concat.h**: logic for concatenating multiple files.
j. **cryptolib.h**: logic for performing encryption and decryption.
k. **delete.h**: logic to unlink files .
l. **fileops.h**: common operations on files.
m. **includes.h**: contains all the libraries and macros and method signatures for jobs.
n. **rename.h**: contains implementations for rename operation.

4. Run the module

First, we must install the module by running command:

sh install_async_sys.sh

This should generate a file **jobmanager**. This would be the binary which takes various operations and allows users to interact with the queue.

The options passed to this executable **strictly restrict the order** in which they are passed.

Below is the option tree that helps us create the right command:

-i insert a job
- **-e** <*in*> <*out*> <*key*> => encrypt
- **-d** <*in*> <*out*> <*key*> => decrypt
- **-c** <*in*> <*out*> => copy
- **-a** <*in_list*> <*out*> => concat
- **-x** <*in_list*> => delete file
- **-s** <*in*> => checksum
- **-r** <*in_out_pair_list*> => rename
- **-p** <*in*> <*out*> => compress file
- **-u** <*in*> <*out*> => decompress file
  - **-h** high pri
  - **-l** low pri
    - **-p** poll
    - **-f** file-out

-l list jobs
- **-h** high pri list
- **-l** low pri list

-g show job details
- **-h** <job_id> => high pri job
- **-l** <job_id> => low pri job

-p switch job priority
- **-h** <job_id> => high pri job
- **-l** <job_id> => low pri job

-d <*job_id*> => delete job

The placeholders enclosed within angular braces (<, >) denote that it must be replaced with a valid input string. For example <in> refers to input filepath. <in_list> refers to a space-separated list of input file paths.

For example, if we want to submit a job into low pri queue to encrypt file *in1.txt* to *out1.txt,* and poll for the result, the command goes as follows:

./jobmanager -i -e in1.txt out1.txt
passwordqwerty -l -p

To uninstall the module:

sh uninstall_async_sys.sh

5. Tests

All tests related to this project are within a directory named “tests/”. They are all shell scripts and can be run using the following command:

sh <file_path>

6. References

[1] Multiple parts of the assignments including but not limited to the encryption logic and system call implementation are used from our own homework 1 and homework 2 repositories

[2]https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html

[3]https://www.kernel.org/doc/Documentation/locking/spinlocks.txt

[4]https://www.kernel.org/doc/html/latest/core-api/workqueue.html#max-active

[5]https://www.kernel.org/doc/html/latest/core-api/workqueue.html#flags

[6]https://elixir.bootlin.com/linux/latest/source/kernel/workqueue.c

[7]https://embetronicx.com/tutorials/linux/device-drivers/work-queue-in-linux-own-workqueue/

[8]http://www.makelinux.net/ldd3/chp-5-sect-5.shtml