OS HW3 — DESIGN DOCUMENT

Group ID: 15
Group Members: Aditi Raj Kandoi, Swetang Finviya, Yash Jetwani

AIM: To perform several operations inside the kernel asynchronously.

GIT BRANCH : origin/master-submission

FILES AND FOLDERS:

1. Makefile
2. install_module.sh
3. kernel.config
4. async_ops_module.c
5. async_ops_module.h
6. async_ops.c
7. async_ops.h
8. test-scripts
9. run_all_tests.sh

Makefile
Contains commands to build and compile module as well as user level code.

install_module.sh
Shell script to insert newly compiled async_ops_module.ko module and remove previously installed version of the module, if any.

kernel.config
Contains kernel configuration used for booting the kernel

async_ops_module.c
Contains kernel side workqueue APIs and operation code

async_ops_module.h
Header file for async_ops_module.c

async_ops.c
Contains user level code for handling command line arguments and communicating with the kernel module async_ops_module

async_ops.h
Header file for async_ops.c, shared by kernel module to refer input and return structs

test_scripts
Folder containing all the unit and integrating tests

run_all__tests
Bash script to run all the test scripts present in test_scripts folder

Steps to build and compile async_ops_module (kernel module) and async_ops (user level code)

1. Go to the folder /usr/src/hw3-cse506g15/CSE-506
2. Run the following command: sudo make
3. Run the following command: sudo sh install_module.sh

PILLARS OF OUR DESIGN:

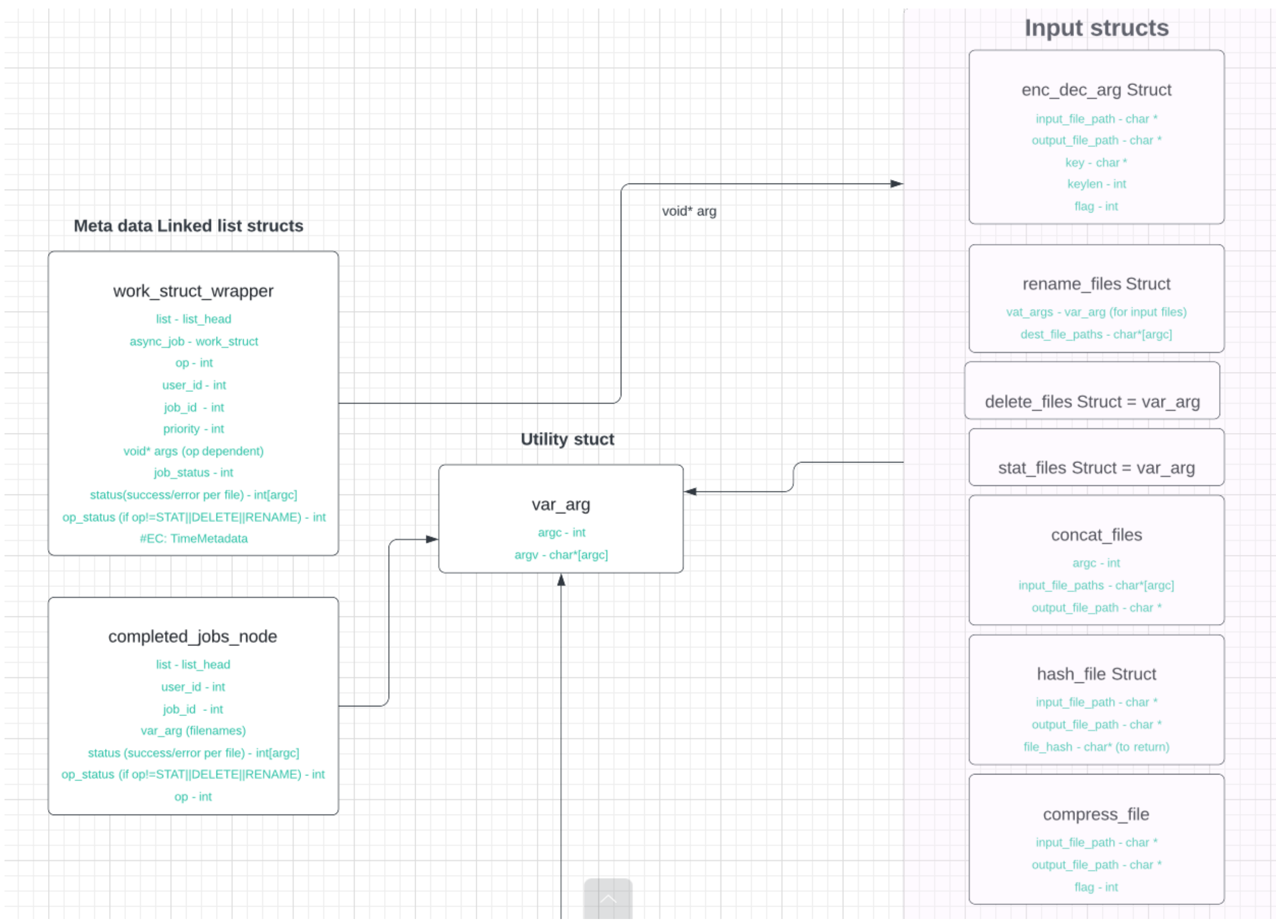
1. **Workqueues**
   We're using linux work queues as the base to carry out queueing operations. The workqueue APIs have served us well for the most part of the project. We wrote additional wrapper structures to carry extra metadata to carry out functions not provided by Workqueues natively.

2. **Active and Completed Lists** [Kernel linked lists]
   We created two lists to maintain our jobs itinerary. They facilitate the abstract movement of jobs during their lifecycle. The active lists have a struct type of work_struct wrapper. This structure contains metadata information of the job like its job_id, user_id, priority etc.
   Two important fields of this structure are:
   Async_job -> This field contains the work_struct which we actually load and unload on our linux workqueues. The function provided to the work_struct depends on the operation in context.
   args -> This is void* field. We use the techniques taught in class to cast this field into a relevant structure depending on the operation in context. For example, in case the operation is encryption, we cast this field to enc_dec_arg stuct.

   The structure of the second list i.e. the completed list is called completed_jobs_node. This structure is an overall optimization to reduce storage space.
   The rationale: As soon as the job is completed, it no longer requires the input args, which can be big char arrays. We do not want to waste kernel memory to store all these input args after completion. But the user can still poll for the status of operation. And hence, we need some kind of metadata to maintain. And thus comes our completed list. As the operation is completed, we mark the enclosing job as complete.
   This job is then collected by our "job_swapper_function" and moved to the completed_list. While moving we dealloc its most memory intensive fields like input char arrays, and only maintain the metadata which is required to respond to user's results poll

query. We explain more about job_swapper_function and its lazy "garbage collection" like behavior next.

The diagram below shows the structures and their relationships —



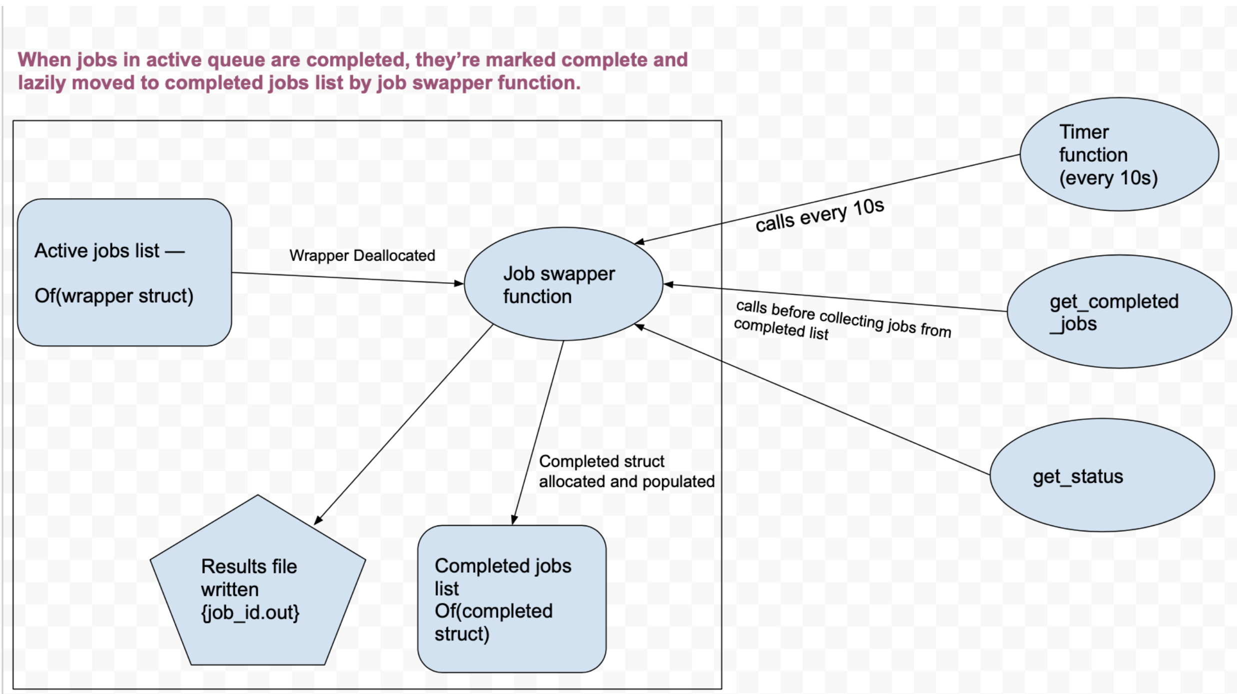
3. **Job swapper function**

The job swapper function works almost like a "garbage collector", the difference being the objects it collects are COMPLETED job wrapper structs. As described above, a job once completed is marked COMPLETE. It is then collected by our "job_swapper_function" and moved to the completed_list. While moving we dealloc its most memory intensive fields like input char arrays, and only maintain the metadata which is required to respond to user's results poll query.

When is the job swapper function called - Is it synchronous? No.

We figured calling a job swapper function for just one complete task would be too much resource consumption. Instead we let the job complete list grow and collect the completed jobs from active lists in one go.

So who calls job swapper function —
We explain that in the diagram below:



`Figure: Callers, input, output of job_swapper_function`

So basically job_swapper_function is called by any function which requires the most updated completed_jobs list.
There's a caveat: Since our job_swapper_function also write results file to the disk. We cannot wait for such a function (one which requires the most updated completed_jobs list) to be called before writing results to the disk. The user might keep waiting expecting its results.
Thus we created a timer function. This function invokes job_swapper_function every 10s. This time is completely customizable.

How did we implement this timer function: We did our research. We learned about busy waiting, linux timers, and tasklets.
(https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch07.html)
And as it turns out the most convenient and stable way to implement a timer function was to use, guess what: Workqueues, again.
Workqueues as it turns out linux timer api to provide a delayed work api, whereby one can mention time in jiffies, and the given function executes after given jiffies.

We embraced this with open arms and created yet another workqueue, appropriately naming it: cleanup_queue

How we use cleanup_queue: At the module init, we init this queue and add a delayed_work struct with a 10s delay. This delayed_work is initialized with the function job_list_swapper_timer_thread. This function calls job_swapper_function and immediately after queues the exact same delayed_work in the cleanup_queue with again a 10s delay.
We learned this in kernel sources, here:
https://elixir.bootlin.com/linux/latest/source/arch/s390/crypto/arch_random.c#L104

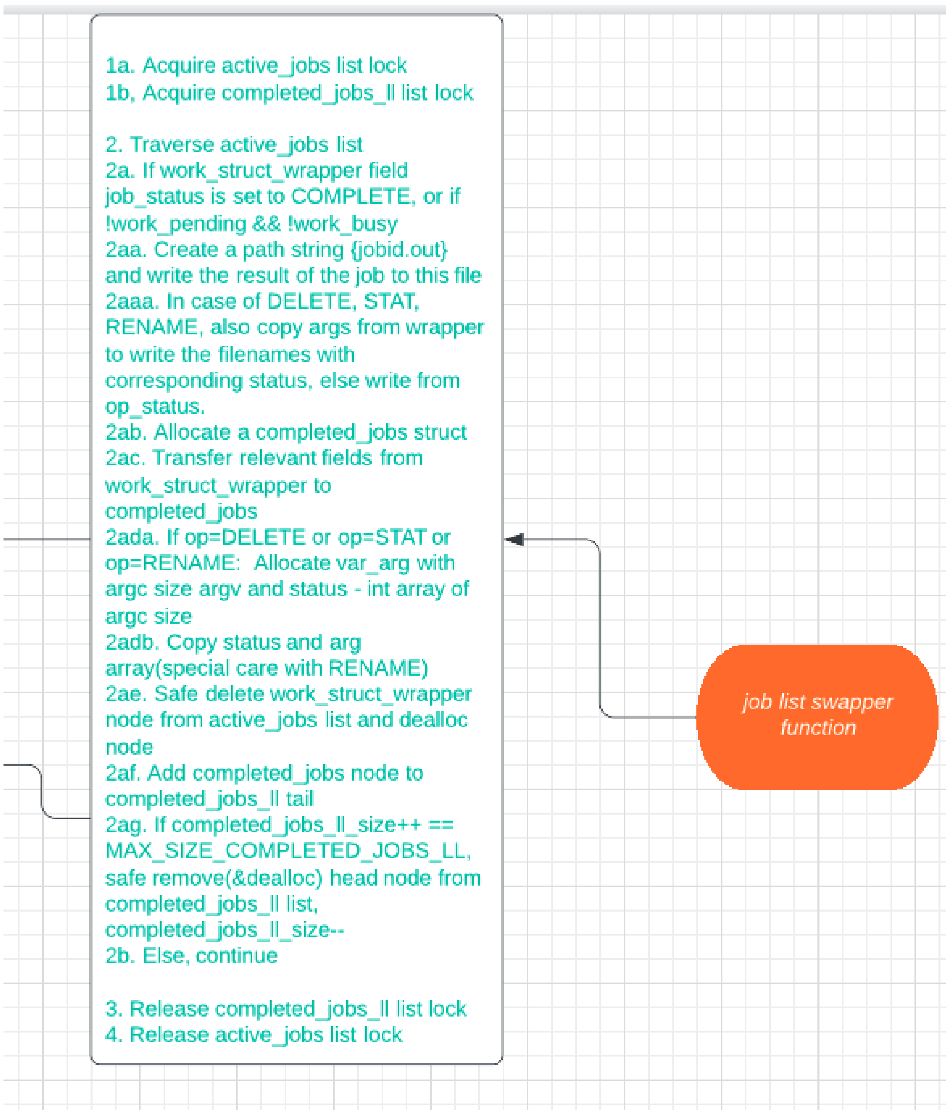


Figure: More about the job_list_swapper function flow

LOCKING MECHANISM USED:

Two locks called active list lock and completed list lock are maintained for preventing simultaneous access to active jobs and completed jobs respectively. Both these locks are mutex locks. We avoid using spinlocks as manipulation of both active list lock and completed list lock exceeds the limited time bounds that favor spinlocks. Also, we maintain that when both the locks are required together, active list lock is acquired first and then the completed list lock. This is done to avoid a deadlock situation.

JOB TYPES SUPPORTED BY THE WORK QUEUE:

1. **Delete multiple files**
   This command will delete all the files provided by the user or return an appropriate error.

   Usage: ./async_ops delete -n N -i file1 file2.. fileN

   Flags: -n: number of files
   -i: the files to be deleted

2. **Rename multiple files**
   This command will rename the files to the file names given by the user after the -o flag.

   Usage: ./async_ops rename -n N -i file1 file2.. fileN -o output1 output2.. outputN

   Flags: -n: number of files
   -i: the files to be renamed
   -o: new names of the files

3. **Concatenate multiple files**
   This command will concatenate the files and store the content in the output file. If any of the input files does not exist, the operation fails and no output file is created.

   Usage: ./async_ops concatenate -n N -i file1 file2.. fileN -o output

   Flags: -n: number of files
   -i: the files to be renamed
   -o: the concatenated file

4. **Getting the stat of multiple files**

This command will generate the stat for all the files provided and store their stat to their respective output files.

Usage: ./async_ops stat -n N -i file1 file2.. fileN -o output1 output2.. outputN

Flags: -n: number of files
-i: the files to be renamed
-o: output files which will store the st*at*

5. **Calculate hash of a file:**
This command will calculate the hash of file.txt and store it in result.txt. md5 algorithm is used for computing the hash. The value of the hash stored in result.txt will be the same as result of following command: md5sum file.txt.

Usage: ./async_ops hash file.txt result.txt

6. **Encrypt or decrypt a file:**
This command will encrypt/decrypt the text present in file1.txt and store the encrypted/decrypted text in file2.txt. The key used for encryption/decryption is derived from the password phrase.

For decryption, if the wrong key is used then the error -EKEYREJECTED will be reported.

Usage: ./async_ops crypt {-e|-d} -p “password phrase” file1.txt file2.txt

Flags: -e: for encryption
-d: for decryption
-p: for password

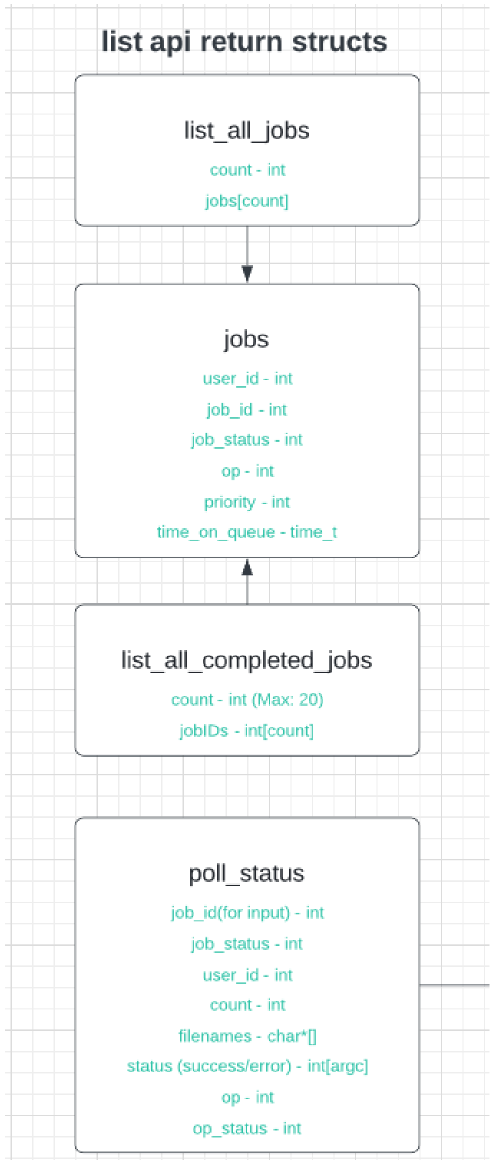
7. **Compress/decompress a file**:
This command will compress/decompress the content present in file1.txt and store the compressed/decompressed content in file2.txt. lz4 algorithm is used for this purpose**.**

Usage: ./async_ops comp {-c|-d} file1.txt file2.txt

Flags: -c: for compression
-d: for decryption

OPERATIONS SUPPORTED BY WORKQUEUE:

Our API response structs —



1. ### Submitting a job

   A job gets submitted to the workqueue by default when any of the above operations is requested to be processed by the module.

Whenever a job is submitted, the program returns a job id as output. This job id can later be used by the job owner to track progress of the job.

To submit a job with high priority use the **-P flag.**

Example to submit a hashing operation having high priority:

./async_ops hash file.txt file.txt -P

Return: A job submitted success message, appropriate error otherwise

2. **Lists all the pending/running jobs**

   All the pending and running jobs in the workqueue are listed using the list jobs operation. Except for root, all other users can only see the jobs submitted by them.

   Usage: ./async_ops list_jobs

   A sample output of this function is shown below:

JOB_ID	OPERATION	STATUS	PRIORITY	TIMEonQ(s)	USER_ID
8	HASH	RUNNING	normal	9	0
9	DELETE	PENDING	normal	7	0
10	HASH	RUNNING	normal	5	0
11	HASH	PENDING	normal	4	0

   This operations shows following information about the job:

   Job_ID: Id of the job
   Operation: Operation that the job has to perform
   Status: Whether the job is running/pending
   Priority: Priority of the job
   TIMEonQ: Time elapsed in seconds since the job has been submitted
   User_ID: Id fo the user who submitted the job

   Implementation:

1. Acquire active_jobs list lock

2. Traverse Active Jobs List
2a. Check job_status field of wrapper struct, if complete, move on
2b. Call WQ job status(pending and busy) Api for every work_struct
3. Keep updating an list_all_jobs struct => Add relevant fields for those Jobs which are not completed

4. Release lock
5. Return list_all_jobs struct

3. **Get status of a job/ poll result of a job**

   Functionality of fetching status of a job or polling its result has been combined into a single operation called get_status.

   Except root, only the user who owns a job can fetch status of a job.

   Usage: ./async_ops get_status job_id
   job_id: the id of the job whose status or result has to be seen

   If the job is running or pending, then the output of the command:
   ./async_ops get_status 12

   could like this: **Job ID: 12 - RUNNING**

   If the job has been completed then the output of this operation depends on the nature of the job.

   Scenario 1: For operations like encryption/decryption, compression/decompression, hashing and concatenation that are atomic in nature, output of get_status looks like this:

   Job ID:12 - Completed
   Operation status: -2

   Here operation status is the error code(as per the linux convention) for the job. 0 indicates that the job is successful. In the above example, -2 is returned indicating ENOENT error.

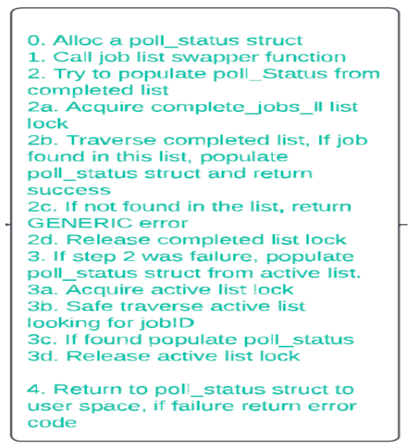
   Scenario 2: For operations like rename, delete, and stat that are executed on multiple files, the output could look like this:

   Job ID: 12 — Completed

/usr/src/hw3-cse506g15/CSE-506/12.out: 0
/usr/src/hw3-cse506g15/CSE-506/13.out: 0
/usr/src/hw3-cse506g15/CSE-506/14.out: -2

Output shown above is the result of a delete job, trying to delete files 12.out, 13.out and 14.out . Status code 0 is returned for files 12.out and 13.out. meaning they were successfully deleted while the file 14.out cou;ld not be deleted as it did not exist in the first place.

Implementation:



4. **Delete a job**

The functionality is used to delete any job currently in the workqueue.

A job gets deleted only when:

- The request to delete a job is submitted by the owner of the job
- The execution of the job has not started i.e. the job is in pending state

Usage: .async_ops delete_job job_id
job_id: the id of the job whose status or result has to be seen

Implementation:

1. Acquire active_jobs List Lock
2a. Safe Traverse Active Jobs List
2b. Check iterativelty for jobID in wrapper struct
2c. If found— Check job_status: Call wq pending api, if success Check priority field of wrapper struct, if high DON'T DO ANYTHING. If low, call wq cancel sync api to remove job from low priority wq.
2d. if api's response successful : Add work_struct to High WQ (using queue work api)
4b. If api response success, update prioirty field of wrapper struct to high
4c. Release active_jobs list lock

5. If JobID not found anywhere, return GENERIC error
6. Return to user space

5. **Priority Boost**

This operation increases the priority of the job from normal to high.

This operation succeeds only when:

- The user is submitting this request is either root or owner of the job
- The job does not already have high priority

Usage: ./async_ops priority_boost job_id

Implementation:

1. Acquire active_jobs List Lock
2a. Safe Traverse Active Jobs List
2b. Check iterativelty for jobID in wrapper struct
2c. If found— Check job_status: Call wq pending api, if success Check priority field of wrapper struct, if high DON'T DO ANYTHING. If low, call wq cancel sync api to remove job from low priority wq.
2d. if api's response successful : Add work_struct to High WQ (using queue work api)
4b. If api response success, update prioirty field of wrapper struct to high
4c. Release active_jobs list lock

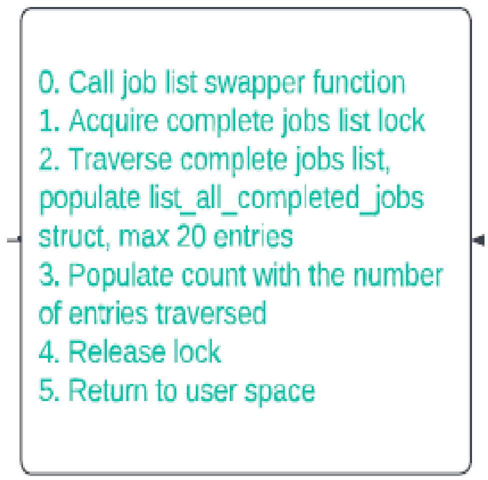
5. If JobID not found anywhere, return GENERIC error
6. Return to user space

6. **Get list of completed jobs**
   The operation gives a list of job_ids of all the jobs that have been completed.

   Root user can see the list of all the jobs that have been completed whereas any other user can only see the completed jobs that are owned by that user.

   Usage: ./async_ops get_completed_jobs

   Implementation:



7. **Giving results asynchronously**
   By default, when a job with job id jid gets completed, its result is stored in the file jid.out . The location of this file is the same as where the kernel module async_ops_module was inserted.

TEST CASES

There are two types of testing performed in the project - integration and unit testing.

1. **Integration Tests**

   a. delete_op.sh: Test to check if one or multiple files can be deleted simultaneously.
   b. rename_op.sh: Test to check if one or multiple files are renamed simultaneously.
   c. concat_op.sh: Test to check if multiple files are concatenated into one.
   d. stat_op.sh: Test to check if stat for multiple files are generated simultaneously.
   e. enc_dec_op.sh: Test to check if encryption and decryption is carried out successfully.
   f. hash_op.sh: Test to check if a hash of a file is generated.

g. comp_dec_op.sh: Test to check if compression and decompression of a file is carried out successfully.
h. delete_job.sh: Test to check if a pending job in the workqueue is successfully deleted.
i. complete_job.sh: Test to check if all the completed jobs in the workqueue are displayed.
j. priority_boost.sh: Test to check if the priority of a job is set to high after calling the priority boost operation.
k. throttling_users.sh: Test to check the limit on the number of jobs that can be added to the workqueue.
l. user_job_access.sh: Test to check user's access to active and completed jobs.

2. **Unit Tests**

a. delete_op.sh: Test to check if the delete operation works for deleting more than **MAX_NUMBER_OF_FILES** files.
b. rename_op.sh: Test to check if correct parameters are provided to rename files.
c. concat_op.sh: Test to check if more than **MAX_NUMBER_OF_FILES** files are concatenated into one and if correct parameters are provided.
d. stat_op.sh: Test to check if stat for more than **MAX_NUMBER_OF_FILES** files are generated simultaneously and if output files are provided to store the stat of the files.
e. enc_dec_op.sh: Test to check if mandatory flags are missing for the encryption and decryption operation.
f. hash_op.sh: Test to check if multiple files can be hashed or an output file is given to store the hash of a file.
g. comp_dec_op.sh: Test to check if correct parameters are provided for performing compression and decompression.
h. delete_job.sh: Test to check if multiple jobs can be deleted at the same time.

REFERENCES

https://kernelnewbies.org/FAQ/LinkedLists

https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch07.html

https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch04s08.html

https://elixir.bootlin.com/linux/latest/source/arch/s390/crypto/arch_random.c#L104

https://lwn.net/Articles/403891/

https://linuxtv.org/downloads/v4l-dvb-internals/device-drivers/ch01s06.html

https://linuxtv.org/downloads/v4l-dvb-internals/device-drivers/API-cancel-work-sync.html

http://www.makelinux.net/ldd3/chp-7-sect-6.shtml

https://stackoverflow.com/questions/43973583/how-to-immediately-cancel-a-work-item-of-a-workqueue-in-a-linux-kernel-module

https://docs.huihoo.com/doxygen/linux/kernel/3.7/structkstat.html#ac88d3ffa03ce961a895d6da7cdd10625

https://opensource.apple.com/source/Libc/Libc-594.9.4/stdlib/FreeBSD/realpath.c.auto.html

https://elixir.bootlin.com/linux/v5.4.3/source/fs/nfsd/nfs4recover.c#L95