README

1. Purpose

Loadable Kernel Module to perform jobs asynchronously using kernel's work queues.

2. Project Description

In this project, I designed an in-kernel queueing system that performs various operations asynchronously and more efficiently. I have used the inbuilt work queues APIs to perform the below-stated operations asynchronously and also support various operations on the queue.

File operations supported:

1. delete multiple files
2. rename multiple files
3. stat() multiple files
4. concatenate 2 or more files onto a new one
5. compute and return a hash for a file
6. encrypt or decrypt a file

Queue Operations supported:

1. Submit a new job.
2. Job Status.
3. Poll results of a job.
4. Delete a pending job.
5. Reorder a job.
6. List all jobs.

Design:

Set up a basic char device as a module (followed this) and used ioctl as a communication mechanism between the user level and kernel module that implements the queue operations.

Kernel Queue Design:

I am using the kernel's workqueue to perform operations.
A global hash table is maintained to perform lookups on job status, list jobs, to delete jobs. Etc on user request.

Asynchronous behavior:

Every job is split into subtasks since there can be multiple files in a request i.e multiple operations that can be run concurrently. (Like do all 3 files delete together)
Each sub task is pushed into the queue at the same time.
Workqueue’s threads will parallely run them ensuring asynchrousity.

Jobs like Concatenation even when it has multiple input files, it will be sent as only one subtasks as it should happen in order and synchronously. (Enc/Dec is also single subtask)

Queuing order is **timed FIFO**, I push requests as and when they come with a delay. This delay time is for the queue to execute already present tasks.
I am using the kernels **delayed_work_struct** and **delayed workqueue APIs** for this specific purpose.

Implementation: ,

For every queue operation on the workqueue APIs , **hashtable** APIs are used. (along with some atomic variables.)

I use GFP_ATOMIC in kmalloc’s to fasten the return time of the UPPERHALF (before submitting the job to the queue)

I don't allocate memory to all the subtasks to contain all the details. I only send the subtask id with the work and a pointer to its main task. So that parameter accessing is easier and more efficient.

In the bottom half, Once every job is in the work handler it accessed the parameters using its main task's arguments.

Main **JOB STRUCT**:

```
       struct job_struct {
          int jobid;
          int uid;                 // for permission checking
          job_type job_type;       // job type
          int job_priority;               // priority

          char ** input_files;    // array of input files for subtask to access
with id
          int input_files_count;

          char ** output_files;   // array of output files for subtask to access
with id

          int subtask_count;      // number of subtask for this job
          char *enc_key;          // for encryption decryption type jobs

          char * result_file;     // if user wants the results written to a file

          void ** subtask_delayed_work_structs; // pointers to queued work
structures
       int * subtask_status; // Integer status of all the subtasks.
       };
```

Work struct of each subtasks:

```
       struct delayed_work_struct {
          struct delayed_work dwork;     // kernel's work api struct
          int subtask_index;             // subtask index to access params from main
job
          struct job_struct * main_task_js;     // pointer to main task
       };
```

So once every job completes I can easily update its status from the subtasks' index and the subtaks_status array of the parent.

Submit Job:

Once user send a struct representing the job,

I allocate memory for the job struct and construct it,

assign a job id

Push the job into the queue (based on priority, more about priority below.)

Returns a JOBID.

I use atomic jobid and use this job id as the key of the hashtable , locking is avoided as the jobid is lock protected by definition.

I dont push the job if the queue is doing many operations parallely and has more pending jobs to be executed. (I set the max not completed jobs at a time can be 14 - Throttling)

LISTING:

I iterate the hashtable and return the details from the hash table.

I populate the result into user address received on request for all output expected jobs (using ioctl)

DELETE:

On delete request I get the job struct from hashtable.

Job struct has pointers to the array of queued subtask work structs. (as mentioned above)

I use the pointers to deque the respective workqueue using **cancel_delayed_work_sync** API.

I only cancel the pending works and not the running / completed ones.

REORDER:

I maintain **two queues.**

One is the normal queue.

Second one is Higher priority one. (flags : WQ_HIGHPRI WQ_CPU_INTENSIVE)

When a user sends a request to reorder the work is cancelled from one of the queues and pushed into the appropriate queue.

Status and Polling:

Since I maintain the hashtable , I access the table get the job struct and the subtask_status will have the status and the results of a job.

Optionally users can send a result file on submit job which will be populated with statuses when a job completes.

Queue Throttling:

I have lock protected atomic variables to maintain the number of pending and running processes , and when a new job is about to be enqueued I check if the already running and pending count does not exceed the threshold. If it exceeds then I cancel the subtask.

File Operations and Usage:

I have 2 files at the user level. One for submitting the jobs to the queue with the respective options from the command line and one for sending queries on queue operations with the respective arguments. Both uses IOCTL commands to communicate with the kernel module.

Files:

create_jobs_userland:

Submitting a job :

This file is responsible for parsing the input command and getting the command type, input files, output files, password, job priority, output/result file etc,.

In this first I am parsing the input command and parsing it to find the functionality requested. Based on the functionality, I am then getting the corresponding input and output files. I am then verifying if the functionality requires extra optional arguments like result file, password, job priority.

Finally I am checking if all the input arguments are proper by running checks on the input files, verifying whether the password matches the minimum criteria (if provided), relative/absolute path conversion. I will create the key of length 16 using the given password.

Once all the necessary checks are performed and input arguments are validated, I will form the user structure which will be passed to the kernel. In our implementation, the communication will happen using IOCTL calls.

I have registered necessary ioctl commands at the user and kernel level. Once the IOCTL call is made, the kernel will implement the functionality asynchronously. It immediately returns the jobID for this task to the user which can be later used for performing operations on queue.

queue_ops_userland:

This file is responsible for the user to communicate with the kernel regarding queries on the queue related to the jobs they have submitted using the above file. In this file, I am collecting the input command and parsing it to find the type of query and get the relevant data as well. Once input is parsed, it will then populate the corresponding structure and will make the IOCTL call to the kernel where the queue is maintained(workqueues in our case).

After the kernel processes this request it will send back the output in the form of an integer. This return value represents the status of the query. Zero means success and negative means failure occurred processing the query and corresponding error messages or error value will be returned.

I am maintaining all the common structure and definitions in the workqueue.h header file. All the operations are implemented in file_operations.h .

Usage and Implementation:

For submitting a new job to the queue, I will be using the./create_jobs_userland binary generated by compiling the create_jobs_userland.c file. The input command should be of the following format.

```
./create_jobs_userland [-j] [-o] [-p] [-excdmshd] <list of files>
```

j and o are optional arguments which represents job priority and output/result file

Encryption / Decryption:

To encrypt a file, the following parameters/options must be passed.
-p -> password
-e/x -> encryption/decryption
< inputfile outputfile>

Requirements:

Password must be of minimum 6 characters. If the password is not passed, the command will return to the user with an error message.

Input File must be existing in the current directory if relative path is passed or absolute path must be passed(file location).

Output file may or may not be present. If given, the input file is encrypted to the output file else the output file will be generated in the current directory.

Ex:

```
./create_jobs_userland -p “password” -e input.txt output.txt
./create_jobs_userland -p “password” -e /a/b/c/input.txt output.txt
./create_jobs_userland -p “password” -x input.txt /a/b/c/output.txt
```

For this functionality, I am using **crypto APIs** to encrypt/decrypt the file content. I am encrypting/decrypting the whole file in blocks. Each block is of size **PAGE_SIZE**. I am also encrypting/decrypting the remaining content if it is less than the PAGE_SIZE. This is done using the key which is passed to the kernel which is created at the user level using password and crypto api to hash it.(**MD5**).

Hash:

-h -> hashing
<input file>

Ex:

```
./create_jobs_userland -h input.txt
```

The only requirement is for the input file to be present and the hash of the file will be pasted to the file of format "input_file.hash". Input file size should be less than or equal to 16*PAGE_SIZE.

Hashing is implemented like encryption/decryption instead of using a key, I am hashing the file content directly for each block and appending it to a buffer and finally once all the blocks are hashed, I am hashing it one final time to form the **16 bit hash.**

Concatenation:

-m -> concatenate
<input1 input2 … output>

All the input files must be present and the output file may or may not exist..

Ex:

```
./create_jobs_userland -m input1.txt input2.txt input3.txt input4.txt output.txt
/create_jobs_userland -m input1.txt output.txt
```

For concatenation, I am again following the above mentioned approach of block by block. I will copy the file content block by block from each file in the given order to the output file.

Delete:

-d -> deletion
<input1 input2 … input>
All the input files must be existing.

Ex:

```
./create_jobs_userland -d a.txt b.txt /usr/src/c.txt
./create_jobs_userland -d a.txt
```

For deletion, I am using the inbuilt kernel function called **do_unlinkat2()** by exporting it before building the kernel(make). (custom defined in fs/namei.c)

Stat:

-s -> stat
<input1 input2 … input>
All the input files must be existing. The output for each input file will be copied to the corresponding output file of format “inputfile.stat”

Ex:

```
./create_jobs_userland -s a.txt b.txt /usr/src/c.txt
./create_jobs_userland -s a.txt
```

For Stat also I am using the existing kernel function vfs_stat and copying the stat result to the file.

Rename:

-r -> rename
<input1 output1 input2 output2 … inputk outputk>

All the input files must be existing and the output file may or may not be existing.

Ex:

```
./create_jobs_userland -r a.txt a1.txt a2.txt b.txt /usr/src/c.txt c1.txt
```

```
./create_jobs_userland -r a.txt a1.txt
```

For rename, I am using the inbuilt kernel function called **do_renameat2()** by exporting it during the build process of the kernel.

Workflow:

User will input the command with necessary options and arguments and is returned the job id. The create_jobs_userland file will then call the kernel with the corresponding structure populated with the input args using IOCTL command. Device driver is created and the work queue is initiated.

I will form the kernel job structure from the user job structure and identify the number of sub jobs based on the functionality. I will then create the required subtasks structure and populate it using kernel job struct and then enqueue it to the queue.

The jobs will then be dequeued processed by scheduler by calling the necessary functions in the work_handler function. The result (job id) will then be returned to the user. All the functionalities are handled in the kernel space.

Justification:

All the operations are performed asynchronously. The user will be returned just the job id and can perform other operations while the task is waiting/running in the background. Throttling is implemented as well along with the queue operations.

Limitations of design:

Hash table cleanup of old completed jobs (relatively) using a background thread.

Test Cases

All the test scripts can be run using the **sh test$.sh** command from the CSE-506 folder.

Each test script is designed in such a way that one functionality implementation is clearly tested end to end.

Below are the test cases implemented with respect to file operations and queue operations:

1. Encryption and decryptions
   Success cases:
      Same key
      Different key
   Failure case
      Password length less than 6.
      Input file not present.
2. Hash of the file.
3. Delete
   a. Multiple files
   b. Single file
4. Rename files
   a. Multiple files
   b. Single file
5. Stat
   a. multiple files
   b. Single file
6. Concatenate files

7. Poll operation on queue
8. List jobs from queue
9. Reorder a job in queue by passing the priority
10. Throttling of queue
11. Cancel the job
12. Submit the job
13. Permissions check on queue ops

References:

https://embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-kernel/

https://www.geeksforgeeks.org/getopt-function-in-c-to-parse-command-line-arguments/

https://elixir.bootlin.com/linux/v5.4.3/source/fs/read_write.c#L432

https://github.com/Embetronicx/Tutorials/tree/master/Linux/Device_Driver/IOCTL

https://embetronicx.com/tutorials/linux/device-drivers/ioctl-tutorial-in-linux/

https://elixir.bootlin.com/linux/v5.4.3/source/fs/read_write.c#L432

https://www.kernel.org/doc/html/v4.18/crypto/api-samples.html
https://elixir.bootlin.com/linux/v4.7/source/include/uapi/linux/time.h
https://elixir.bootlin.com/linux/v4.7/source/include
https://elixir.bootlin.com/linux/v5.4.3/source/include/linux/workqueue.h
https://elixir.bootlin.com/linux/v5.4.3/source/kernel/workqueue.c
https://elixir.bootlin.com/linux/v5.4.3/source/include/linux/hashtable.h