Async Queue Implementation

CSE 506: OS, HW3

GROUP 05:

Sai Harish Avula	- 113276530
Jaideep Penikalapati	- 113220647
Venkata Sai Srikanth Ketepalli	- 113277520

1. Project Description and Requirements:

Designing an in-kernel queueing system that performs various operations asynchronously and more efficiently. Implement the Queue and develop code to perform operations inside the kernel asynchronously

Project Requirements:

A. Design a kernel queue that is initialized when kernel module is loaded. The queue should have a length limit as discussed in class, and trying to submit to a full queue should block (i.e., throttle) the submitter

B. Design how to access the queue from the user land. The Queue should support at least 5 job types out those listed

- Delete multiple files
- Rename multiple files
- Stat multiple files
- Concatenate 2 or more files into new one
- Compute and return a hash of a file
- Encrypt/Decrypt a file
- Compress/Decompress a file

C. The queue should pick jobs in some order (e.g., FCFS), perform them, collect status (e.g., success or failure), and be able to return results to the users upon request.

D. Support Queue functionality for the users such as

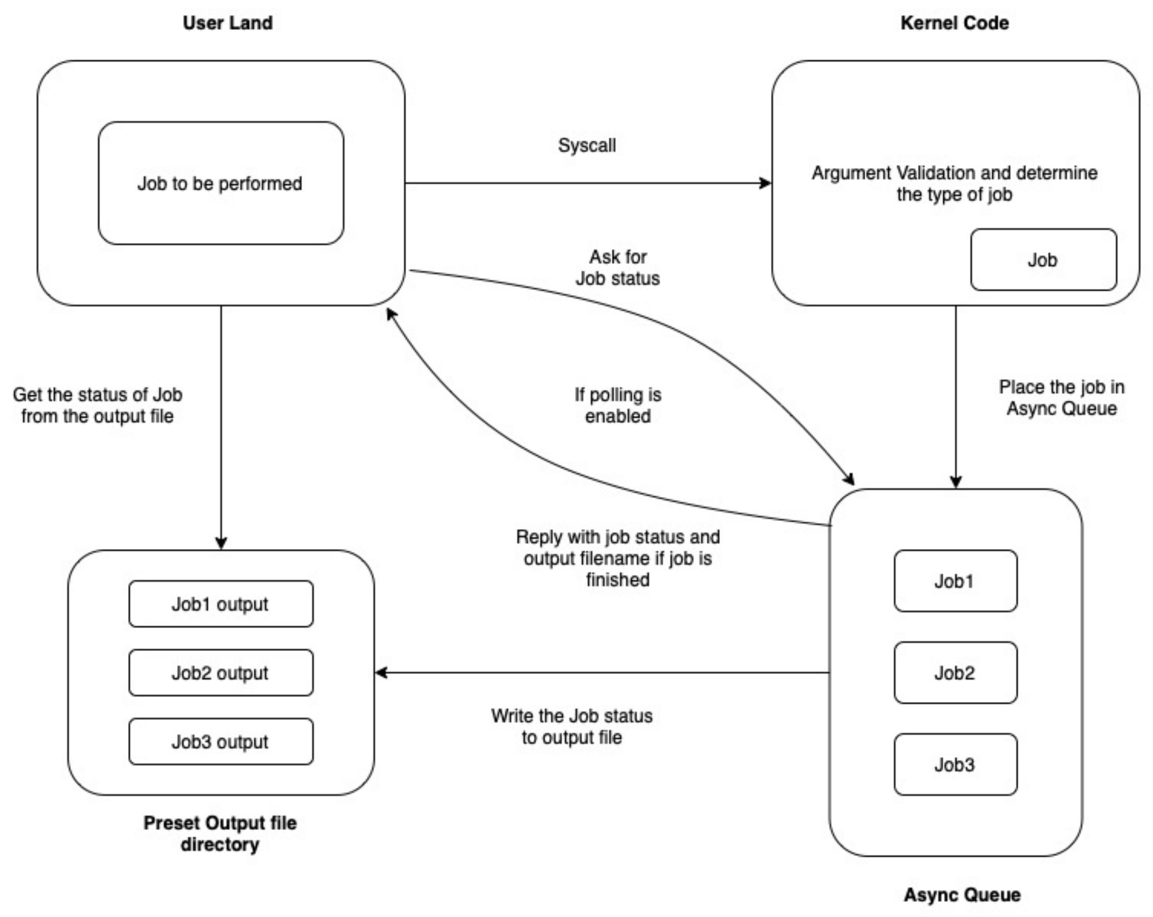
- Submit a new job with all of its args
- Check on the status of a pending job
- Poll to retrieve the results and status (ok/fail) of any job
- Delete a pending job in the queue or one that's running
- Reorder a job (e.g., to give it a higher or lower priority)
- List all jobs (or all jobs on behalf of a user)

E. Design user level tools to access and manage the queue, and scripts to demo and test your work

2. Kernel Queue Design:

Our Kernel Queue design is based on workqueue present in the kernel (based on the functionalities present in the file /kernel/workqueue.c)

Basic Design:



User Land
Kernel Code
Job to be performed
Syscall
Argument Validation and determine the type of job
Job
Ask for Job status
Get the status of Job from the output file
If polling is enabled
Place the job in Async Queue
Reply with job status and output filename if job is finished
Job1 output
Job2 output
Job3 output
Job1
Job2
Job3
Write the Job status to output file
Preset Output file directory
Async Queue

Async Queue Design

Struct/object Implementations used:

1. Implementations imported from kernel code:

- Actual Queue is based on the struct <workqueue_struct> from workqueue.c
  -> create_workqueue(): To initialize a new workqueue
  -> queue_delayed_work() and queue_work(): to insert a work item into the queue
- The Work item inside the queue is based on the struct <delayed_work> and <work_struct>. The operations this struct supports are
  -> INIT_DELAYED_WORK(): To initialize the delayed_work struct (actual work item)
  -> this function takes a call-back function as argument, which will be used for the work item to work on when its dequeued from the workqueue_struct
  -> cancel_delayed_work(): To remove a queued work item from the queue

2. Custom Implementation to support other functionality:

- HashTable: (include/linux/hashtable.h) to store the work items based on the key (the job_id of that particular job). This would help us with following operations
  -> hash_add(): to insert a work item into the map (enqueue in workqueue)
  -> hash_del(): to delete a work item present in the map (dequeue in workqueue)
  -> hash_for_each_possible(): to retrieve a work item based on job_id
  -> hash_for_each(): to get all the stored work items in hash table (list all jobs)

3. Implementation for throttling producers:

- Wait queue is implemented based on struct <wait_queue_head>
  -> wait_event_interruptible(): interrupt to place in the wait queue until the workqueue has space to accept new work items. (With a max limit on waiters)
  -> wake_up_all(): interrupt to wake up all the producers waiting in the wait queue. (The work items that we put to wait in the earlier method)

4. Locking:

- Locking is implemented using mutex lock that's available in the kernel struct <mutex>
  -> mutex_lock() and mutex_unlock(): to lock and unlock the mutex respectively
- The counters (for size of queue and producers that are waiting) are maintained using atomic counters <atomic_t>

Design:

All the queue operations are implemented as follows:

1. Insert a new job in the queue: After validation and populating all the arguments required for a specific job, the job is inserted into the queue using queue_delayed_work() with a delay depending on the priority of the job. The work item reference is also inserted into the hash map for reference hashed by its job_id. If the queue is full, we place the job in the producers wait queue until wake_up_all() interrupt wakes items in producer wait queue. If the producer wait queue is also full, we return error to the user (synchronously) that no more space is available for new jobs.

2. Working on a queued job: The workqueue implementation will call the callback function (set while initializing the work_struct item) when it's time to be worked on, which are picked FCFS after the initial delay set during enqueue. We remove this work item from the map first and then appropriate job is performed based on the arguments. As the work item is removed from queue we call wake_up_all() interrupt to enqueue any work items in the producer wait queue.

3. Deleting a queued job: Once the delete job sys call is received, we check the map to see if the job id still in the waiting queue or completed/in-progress. If the job is still in waiting state, then the job is removed from the map and then deleted from the queue using work item reference in the map. (*Design Choice*: If the job to be deleted is

currently processing or completed processing, then the job will not be completed and the user will be notified accordingly)

4. Changing priority of a queued job: Similar to delete, first we check if the job is still waiting in the queue from the map and then compare its priority with the new priority. In case the priorities differ (currently 3 levels of priority are supported), We delete the job from the workqueue_struct and re-insert into the queue based on new priority.

Low priority	-> insert with delay 10s
Default priority (medium)	-> insert with delay 5s
High priority	-> insert with delay 0s (immediate processing)

This scheme also ensures there is no starvation in the queue for default and low priority jobs. Example: When a user tries to add a high priority job continuously with a low or a medium priority job waiting in the queue, These jobs will not be starved immediately. Once the initial delay during insert is completed, the jobs are picked FCFS so a waiting low priority job with 0 secs remaining delay will be picked before a new high priority job.
Note: The initial delay values are configurable in the constants.h file

5. List all queued jobs: To list all the queued jobs, we iterate over the items in the hash map and list the jobs (job_id and job_type), comparing the uid value of user submitting the list job with the uid value of queued jobs. (This validation is bypassed for a root user, i.e, root user will be able to list all the queued jobs)

6. Locking and accessing shared values: mutex is used to lock the workqueue in the following scenarios.

- when the incrementing/decrementing the workqueue size
- enqueuing a job into the workqueue
- deleting a job from the workqueue
- changing the priority of the job in the workqueue

The workqueue size is the shared variable used to increment/decrement queue size accordingly.

7. Job status and output to user: All the synchronous jobs (List all queued jobs, delete a queued job, change priority of a queued job) provide immediate feedback to the user land. All the other jobs report the job status and the job output in two ways:

A. If the user adds an argument to enable polling, then once the job is submitted to the queue, The user land code will create a new thread which polls every second (at max 300 secs) calling the kernel to check the status of the job. The kernel will send back the status of the job as "WAITING" or "RUNNING" appropriately if the job hasn't completed yet. If the job is completed, kernel will return a filename to read results of the job for the user.
B. If the user did not provide argument to enable polling, then the current code execution ends and returns a job_id to the user. Then the user can use this job_id to manually check the job status and output using the same user program with valid arguments. (Appropriate file access validations are done to prevent access of any user to check status of any job)
C. Default case: Disregarding If the polling is enabled or not, each job writes its output to a designated dir and a filename (job_id), which can be used to retrieved later either by polling or different get status sys call

3. Jobs Supported and outputs:

Non - Queue specific jobs (Asynchronous jobs):

A. Encrypt/Decrypt a file: Encrypt or decrypt a file based on arguments and the filenames. Only the user with RW access to the input file will be allowed to encrypt/decrypt the file. The output written to file is encrypt/decrypt status and delete status of the original file.

B. Delete multiple files: Delete one or more files. Output written to the file is the delete status (success or failure) of each filename in arguments

C. Rename multiple files: Rename one or more files. Output written to the file is the rename status (success or failure) of each filename in arguments

D. Stat multiple files: Get stat for one or more files. Output written to the file is stat status of each file along with the details of stat for each successful stat operation.

E. Compute hash for a given file: Recursively hash each block of file starting with a hardcoded hash value.

Example:

<16 byte hardcoded hash>+<first x bytes of file>=<16 byte hash input for next x bytes>
<previous hash>+<next x bytes of the file>=<16 byte hash for next x bytes>
...
... =<final 16 byte hash of the file>

The output stored in the file is job status and final hash value of the file (if success).

F. Compress/Decompress a file: Compress or Decompress a file depending on the arguments. The output written to file is job status of compression or decompression. The libraries used for compression and decompression are *crypto_comp_compress* and *crypto_comp_decompress* respectively

G. Concatenate two or more files into one: Concatenate multiple files into one file. The output written to file is the job status of concatenation. (*Design choice:* If there is an error with even one file during concatenation, the job is suspended and partially written output file is discarded)

Queue specific jobs (synchronous jobs):

A. List all Jobs: List all pending (queued) jobs in the queue as well as producers waiting in the producer wait queue (throttled producers). The output is sent back to the user in the format of

1. *<Job_id> <job_type> <job_priority>*
2. *<Job_id> <job_type> <job_priority>*

...

B. Delete a queued job: Delete a queued job, based on the job_id, if the job is already completed or processing, the sys call will return ESRCH (no such process) errno to the user or success (0) if the job is successfully removed from the queue. If the User do not have access to the job (If the job is created by a different user), kernel return EACCES status to the user, without performing delete operation

C. Change Priority of a queued job: Change priority of the job based on job_id. Sys call sets success/failure (sets appropriate errno) and returns to the user. if the job is already completed or processing, the sys call will return ESRCH (no such process) errno to the user. If the new priority matches the old priority, then the status EBADRQC will be sent back to the user. If the User do not have access to the job (If the job is created by a different user), kernel return EACCES status to the user, without changing priority of job

D. Get status of a particular job: Get the current status of a job, If the user do not have access to the job, the return value is set to EACCES. If the Job is completed, kernel returns the filename of the job output. In other cases, the output would be the status of the job, I.e., WAITING, IN-PROGRESS, NO SUCH JOB (for invalid job-ids)

4. User Permissions and Access:

Job Specific Permissions:

All jobs where the user do not have read and write access to input files would result in EPERM error status in the output log (job output file).

- Delete/Rename/Stat: only the files that user has access to are processed
- Concat: If any file in the input files is not accessible to the user, the job is suspended and partial output files are deleted and the status would be written as EACCES to job output file
- Encrypt/Compress: Input file will be encrypted/compressed only if the user has access to the file. Output file will be generated and EACCES error will be written to output

- Hash: User only need read access to generate hash of the file. The hash of the file will be written to the job output folder.

Other Permissions:

Other permission restrictions on users implemented are:

- User can only list queued jobs that are inserted by the same user. (List all jobs output will only list down jobs queued by the same user)
- User can only check the status/output of the job(s) queued by same user.
- Deleting a queued job can only be done by the same user who has queued it.
- Only root user will be able to list all jobs. Check status and delete job restrictions still apply for a root user. (Root user cannot delete or check status of other users jobs)

5. Race Conditions and Safe Unlinks/Deletes:

Unlink (deleting files):

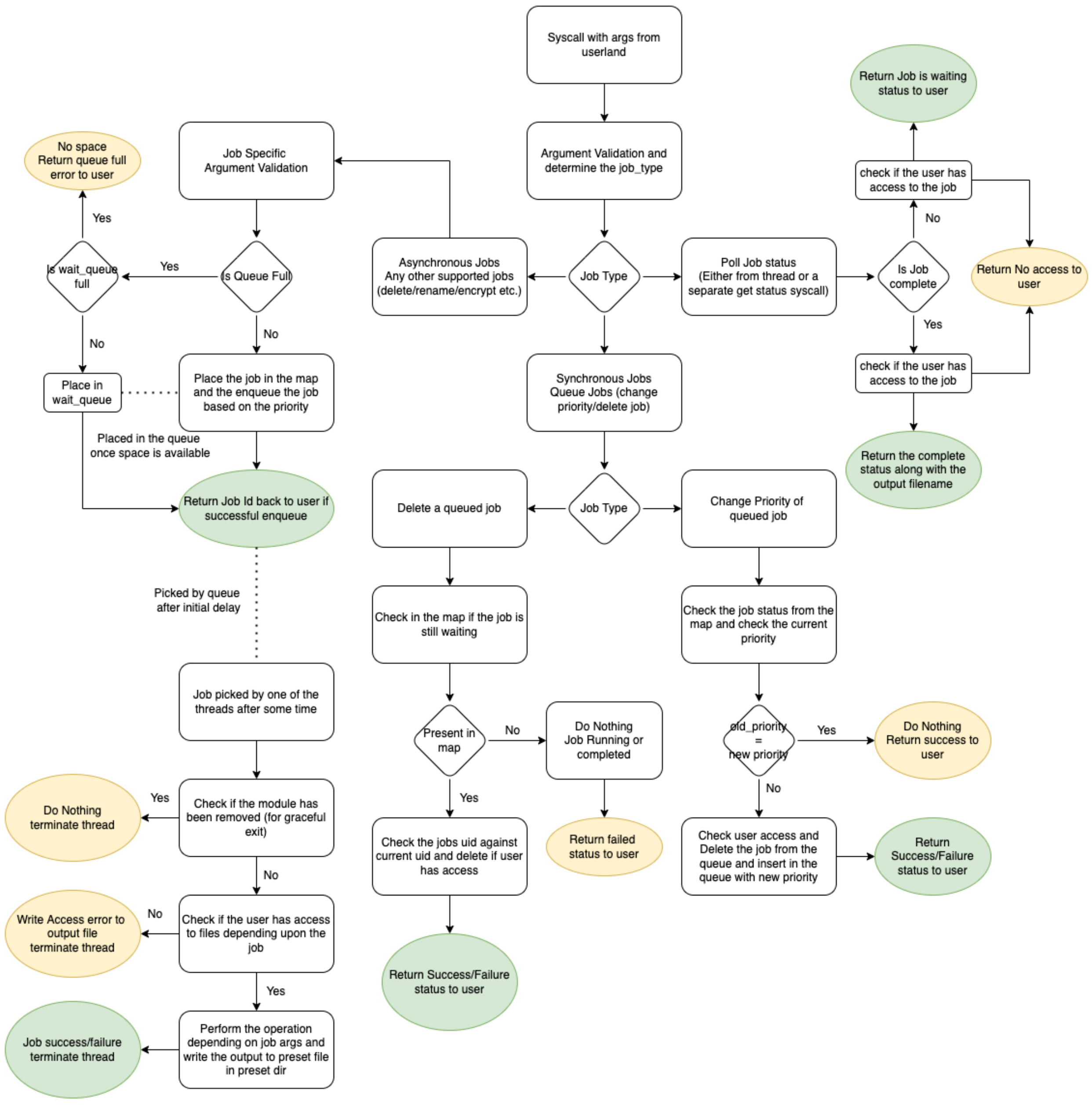
We have created utility for safe_unlink which makes sure to lock the parent directory inode before vsfs_unlink and unlocks it after the unlink operation. We also made sure to filp_close the file before using the safe_unlink method.

Rename:

Similarly for the rename operations, we made sure to use lock_rename & unlock_rename methods before & after using vfs_rename respectively. For the rename, we made sure to capture and release the reference count using dput & dget.

6. Flow diagram of the sys call:

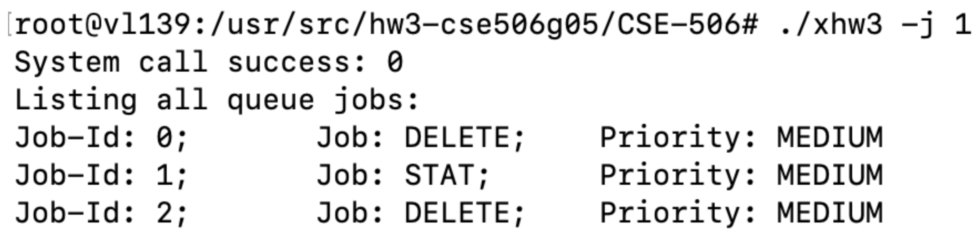
The complete flow of the sys call is as shown


Syscall with args from userland
Argument Validation and determine the job_type
Job Type
Asynchronous Jobs Any other supported jobs (delete/rename/encrypt etc.)
Job Specific Argument Validation
Is Queue Full
Yes
Is wait_queue full
Yes
No space Return queue full error to user
No
Place in wait_queue
No
Place the job in the map and the enqueue the job based on the priority
Placed in the queue once space is available
Return Job Id back to user if successful enqueue
Picked by queue after initial delay
Job picked by one of the threads after some time
Check if the module has been removed (for graceful exit)
Yes
Do Nothing terminate thread
No
Check if the user has access to files depending upon the job
No
Write Access error to output file terminate thread
Yes
Perform the operation depending on job args and write the output to preset file in preset dir
Job success/failure terminate thread
Poll Job status (Either from thread or a separate get status syscall)
Is Job complete
No
check if the user has access to the job
Return Job is waiting status to user
Return No access to user
Yes
check if the user has access to the job
Return the complete status along with the output filename
Synchronous Jobs Queue Jobs (change priority/delete job)
Job Type
Delete a queued job
Check in the map if the job is still waiting
Present in map
No
Do Nothing Job Running or completed
Return failed status to user
Yes
Check the jobs uid against current uid and delete if user has access
Return Success/Failure status to user
Change Priority of queued job
Check the job status from the map and check the current priority
old_priority = new priority
Yes
Do Nothing Return success to user
No
Check user access and Delete the job from the queue and insert in the queue with new priority
Return Success/Failure status to user

Async Queue syscall Flow

7. Examples:

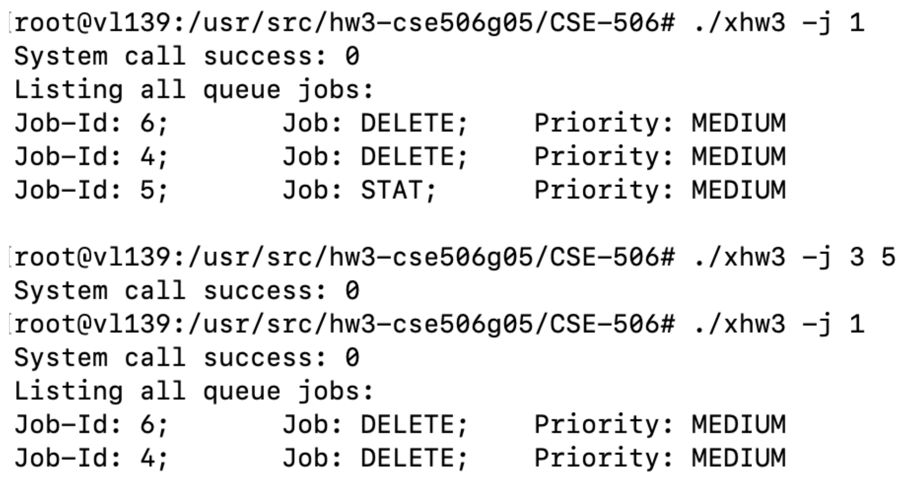
Listing Queued jobs:



```
[root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 1
System call success: 0
Listing all queue jobs:
Job-Id: 0;       Job: DELETE;    Priority: MEDIUM
Job-Id: 1;       Job: STAT;      Priority: MEDIUM
Job-Id: 2;       Job: DELETE;    Priority: MEDIUM
```

List queued jobs

Delete Job from Queue:

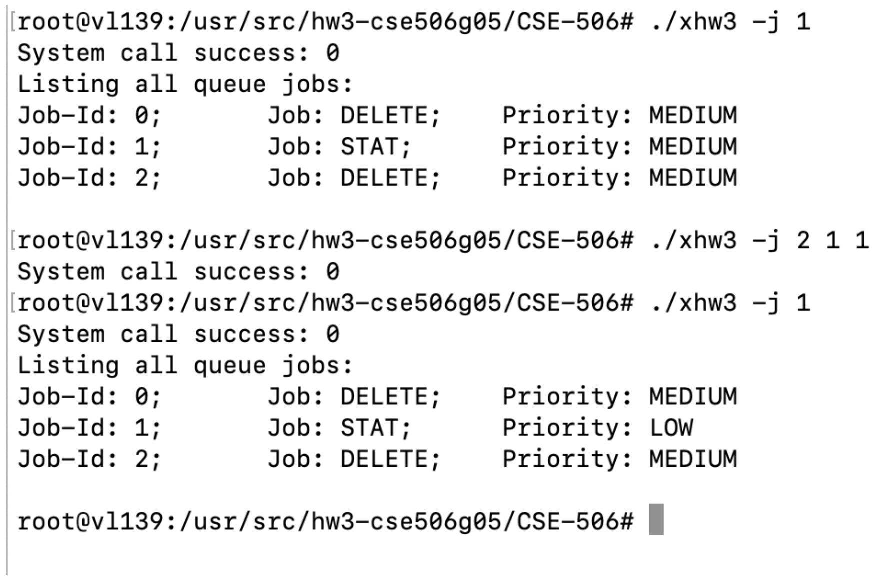


```
[root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 1
System call success: 0
Listing all queue jobs:
Job-Id: 6;       Job: DELETE;    Priority: MEDIUM
Job-Id: 4;       Job: DELETE;    Priority: MEDIUM
Job-Id: 5;       Job: STAT;      Priority: MEDIUM

[root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 3 5
System call success: 0
[root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 1
System call success: 0
Listing all queue jobs:
Job-Id: 6;       Job: DELETE;    Priority: MEDIUM
Job-Id: 4;       Job: DELETE;    Priority: MEDIUM
```

Delete a queued job

Change priority of a Job:



```
root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 1
System call success: 0
Listing all queue jobs:
Job-Id: 0;       Job: DELETE;    Priority: MEDIUM
Job-Id: 1;       Job: STAT;      Priority: MEDIUM
Job-Id: 2;       Job: DELETE;    Priority: MEDIUM

root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 2 1 1
System call success: 0
root@vl139:/usr/src/hw3-cse506g05/CSE-506# ./xhw3 -j 1
System call success: 0
Listing all queue jobs:
Job-Id: 0;       Job: DELETE;    Priority: MEDIUM
Job-Id: 1;       Job: STAT;      Priority: LOW
Job-Id: 2;       Job: DELETE;    Priority: MEDIUM

root@vl139:/usr/src/hw3-cse506g05/CSE-506#
```

Change priority of a job