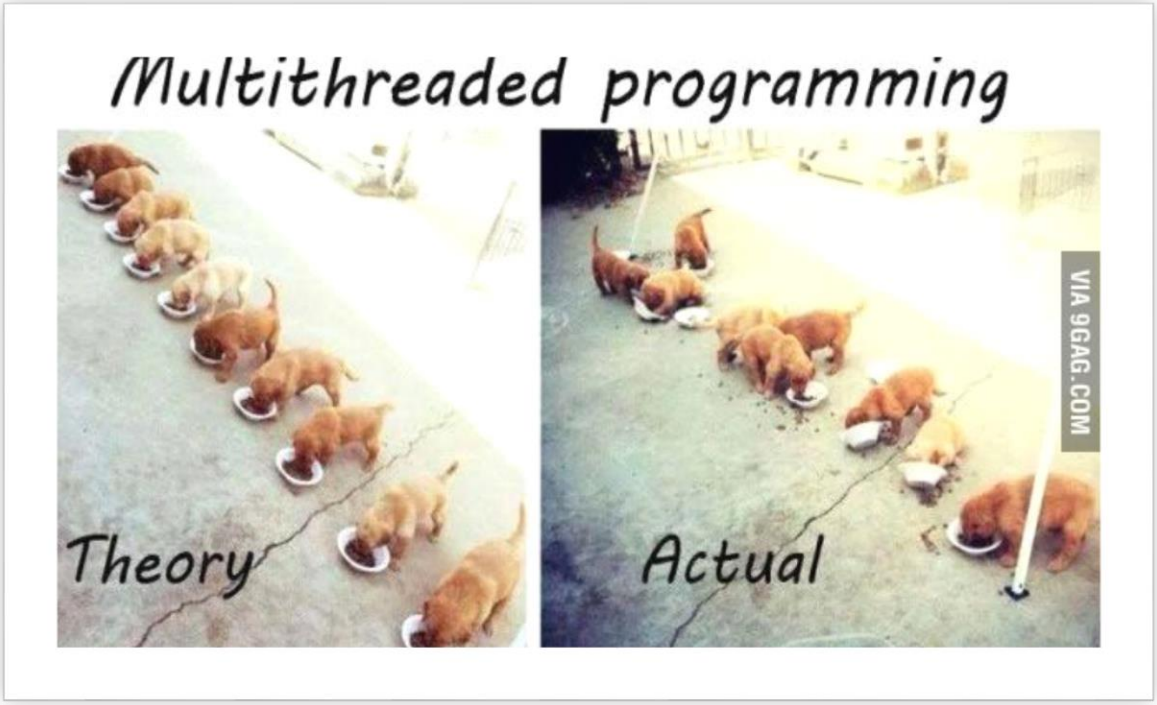


KWQM

IN KERNEL WORK QUEUE MANAGER

Aadarsh Jajodia–110347086

Shahzeb Nihalahmed Patel 110369918

Image courtesy: www.9gag.com

Introduction

KWQM, as we call it, is an in kernel implementation of asynchronous work queue manager implementation. User can call the submitjob system call to submit a job to kernel where we have worker threads spawned that work on the job in background and not blocking the user program or thread.

Approach

The approach involves maintaining a work item(job) queue in the kernel. This is a priority queue that maintains priorities at three levels i.e. high, medium and low. It then spawns a predefined number of worker threads. The user program submits the job to the kernel through the submitjob system call which gets enqueued in the work item queue based on the priority of the job. The user process then returns. The worker threads then remove the job based on first come first serve + priority of the job. It then performs the job that is mentioned. Based on the outcome of the job performed, it writes the result on the netlink callback socket that servers as a callback. The callback is received on the user side to receive this callback.

How did we do?

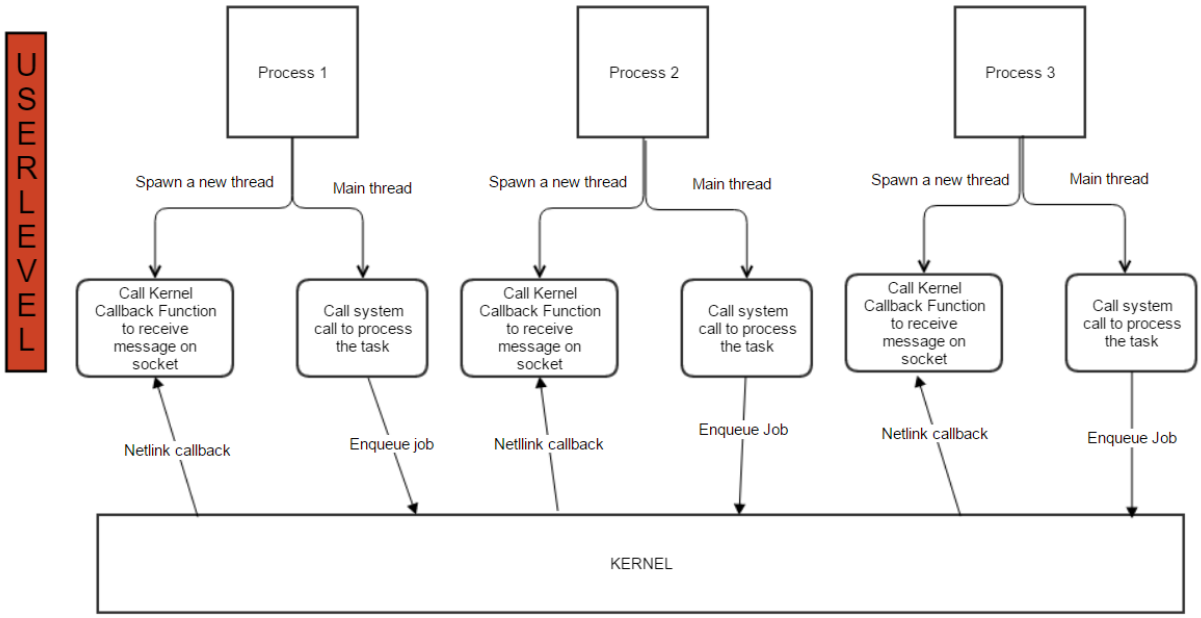
User Program

The user program fills a work item with all the job details. These are the arguments that are required to do the job by kernel worker thread. As of now, we have implemented just one feature, that is encryption and decryption of a file.

1. Id
2. Operation
3. Priority
4. Input file name
5. Output file name
6. Flags (operation specific flag)
7. Keylength
8. Key
9. Args (custom args)

The user program fills these arguments based on command line arguments and submits the job to the user. The job id is same as the process id of the user program. This is done to ensure the uniqueness of the job id. But user code can be modified to change this assignment. It then spawns a separate thread that functions as a callback thread bind with a function that listens on a netlink for callback data. The important thing to notice here is that the main kernel thread submits the job

and does something else (in our case, prints a message that it is waiting). The callback thread prints the data when it receives it and exits.



Other queue management requests handled by KWQM

1. Remove an enqueued job:

   This allows user to remove an enqueued job. User needs to provide the job id of the enqueued job

2. Change priority of an enqueued job:

   This allows user to change the priority of an enqueued job. The user needs to provide the job id of the enqueued job and the new priority to be assigned to the job.

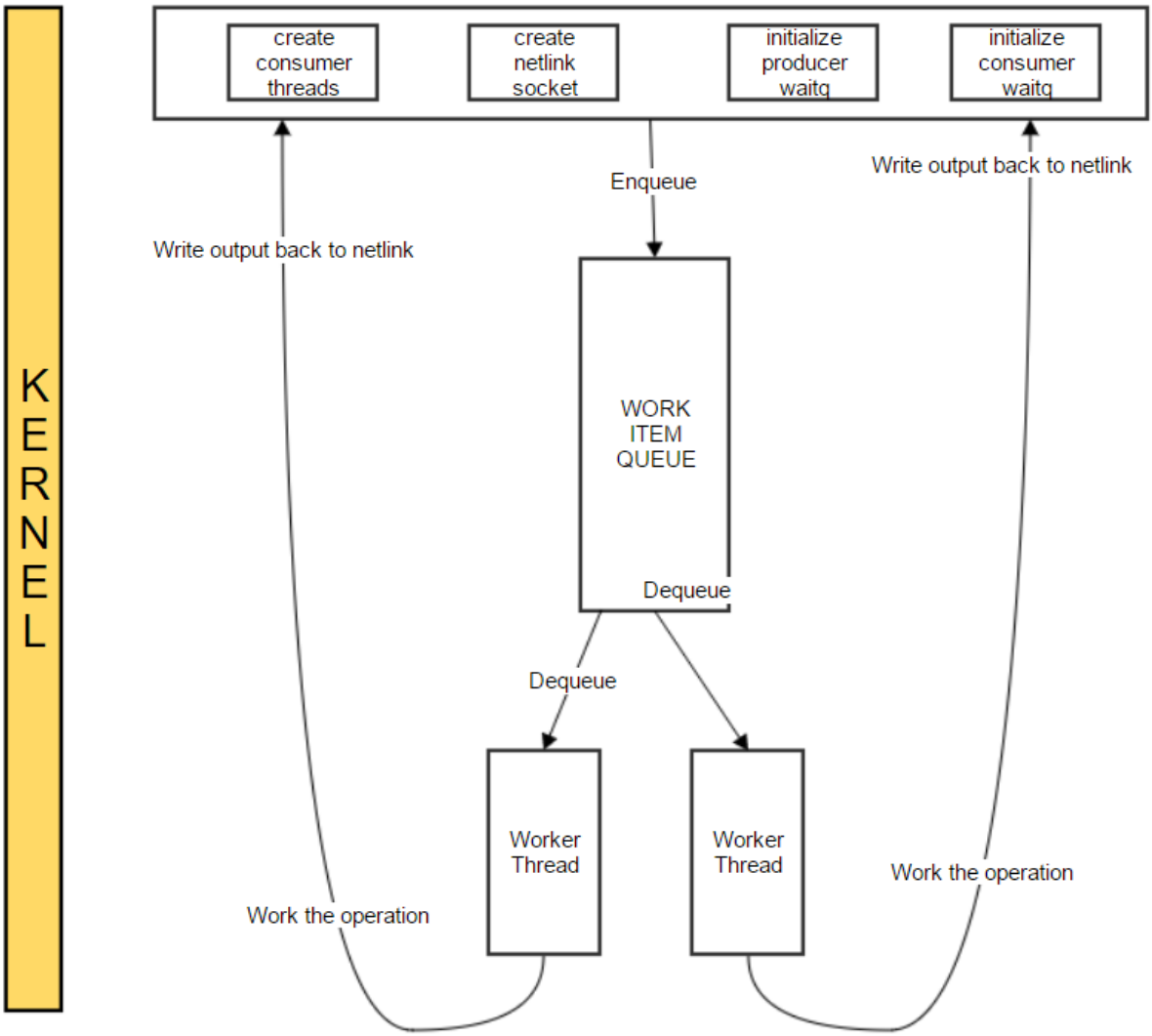
3. List all enqueued jobs.

   This allows the user to list all the enqueued jobs.

   The queue management requests are handled synchronously.

Kernel Section:

When the system call module is loaded, a queue with three heads is initiated. Each head represents a smaller queue of respective priority. The queue data structure is the union if this three queues, a mutex lock to be acquired while operating on queue and a counter to maintain the size of the queue. The queue size is limited. There are waitqueues for consumers and producers.



In case of queue management requests (remove, change priority, list), the lock is acquired on the queue and the respective job is performed. In case of listing, the job ids of the enqueued jobs are written to the user space address.

In case of normal job (encryption/decryption), the job is enqueued in the queue. If the queue is already full, the producer process is throttled. There is a limit to how much producers can be throttled. If the number of throttled producers reach this limit, more incoming producers are returned with an error that their job cannot be processed (ENOSPC).

If the job gets enqueued successfully, the worker thread dequeues it based on its priority. Higher priority jobs are dequeued first. It then performs the job based on the type of operation specified (encryption/decryption on our case as of now). The result of this operation is performed is recorded and written on netlink socket. This has a map of process ids which in our case is same as the job id. The callback thread in user mode then reads this message (it reads for its own process id) and prints the result.

Throttling Consumer Worker threads: The consumer threads are throttled when there is no item in the work queue. As soon as an item is enqueued, the consumer threads are woken up in the pursuit of completing the job. All throttled consumers are woken up so as to complete more than one jobs that are enqueued will the consumer is woken up. This is a lazy approach we have applied to throttle the consumer thread. The worker threads that don't get any valid job are throttled again. This is to make job processing efficient.

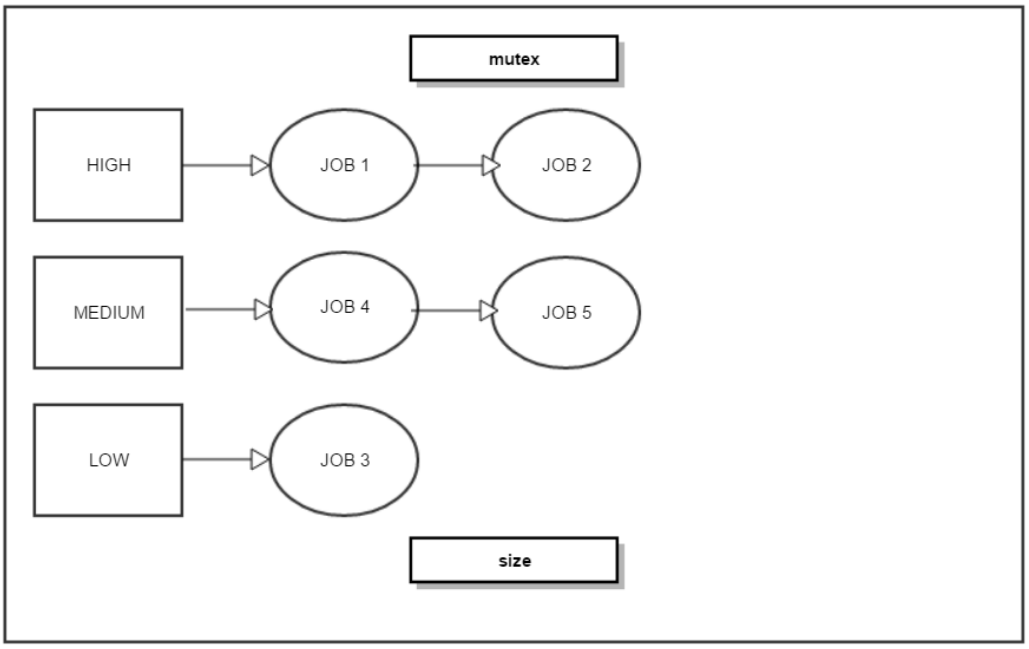
Throttling Producer threads: When the work item queue is full, the producer thread is throttled. However, we have limited the number of producer threads than can be throttled. If this limit is reached, more producers are come to enqueue a job are returned with an error of ENOSPC.

Functionality:

Our implementation as of now support the encryption and decryption of a file. Appropriate arguments are to be provided while running the user executable. The job is submitted to the user and user code returns, and the return value of the work done is received on netlink.

Queue details:

The work item queue is ab abstract data structure that actually contains a head pointer to tree lists, each representing a priority. Work item queue has a mutex lock which is supposed to be acquired before using the queue or performing any kind of operation on the queue. There is also a size variable to keep the track of the current size of the queue. It is modified under the mutex lock. The size of the queue is the cumulative size of list nodes



Enqueue: the work item (job) is enqueued in the appropriate list based on the type of priority of the work item. The job is enqueued at the tail

Dequeue: The work item is first removed from the high priority list, then the medium one and then the lower one. The work item is removed from the head.

Future work:

Our implementation now has a scope of doing only one type of job. i.e. encryption and decryption. As we made the main focus of the assignment to efficiently implement multiple producer consumer queue without any kernel panic, we could implement only one feature.