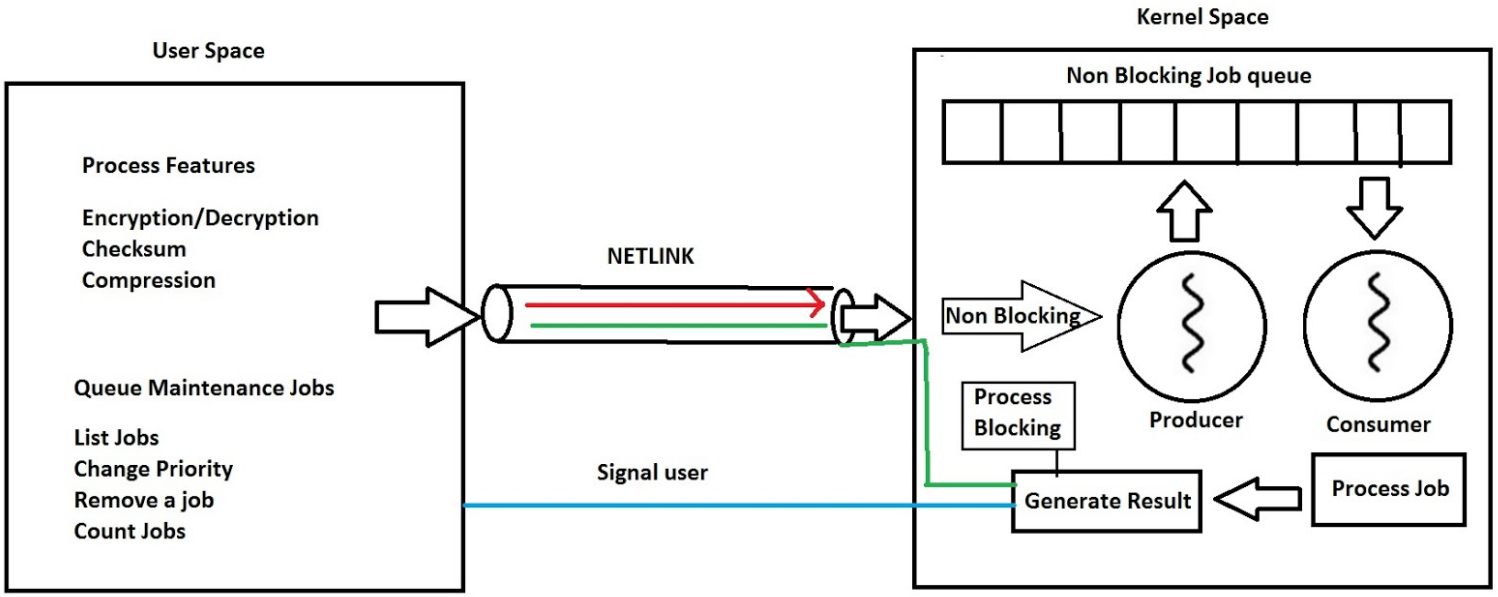
Asynchronous and Concurrent Processing

By: cse506g21

Design overview:



User program takes requests from the user and submits them to the kernel process via netlink socket. It then proceeds to do useful work(in this case accept more requests from user).
Kernel creates a producer and a consumer thread on init. All process features are jobs queued by the producer and are consumed and processed by the consumer. After processing a job the consumer will send its result back to the user along with the its job id(user provided).
All queue maintenance jobs(list, remove, change priority, count of jobs) are processed by the kernel process itself and are not handled by the producer/consumer threads.

Input Data structure:

In "sys_submitjob.h"

```
struct job
{
        int job_code;
        char *user_job_id;
        int priority;
        int is_atomic;
```

```
        int pid;
        void *operation;
};
```

Extensibility of input data structure:
All the common attributes of a job are added in the structure. We added a void operation that can encode any kind of operation we want to submit to the kernel. All the blocking and nonblocking operations that we have implemented are encapsulated in respective structures and passed to this void pointer.

user_job_id is provided by the user and is unique for each job.

Output Data structure:

Blocking calls:
For blocking calls like listing of jobs, remove, counting of jobs and changing of priority we are copying the result to the output buffer pointed to by the respective job structure

Non Blocking Calls:
Once done with the operation, the success or failure message is encoded in a char buffer and sent to the user via netlink. Once the result is written to the socket , we send a signal to the user to check his netlink socket to receive the new messages.

Structure of the user output buffer

return value (4 bytes)	user job ID len (4 bytes)	user job ID	return buffer length (4 bytes)	return buffer

User in his signal handling function will recieve message from the socket and will retrieve the result corresponding to the user job ID.

Asynchrony by use of Netlinks and Signal Handling:

In “user_submitjob.c”

We have used Netlinks to achieve asynchronous submission of jobs by the user. The user submits the jobs to the kernel process and then is free to do whatever it wants.
Whenever a job has been processed by the kernel, it puts the result of the job on the netlink socket and interrupts the user via a signal. The interrupted user checks the netlink socket, receives the message that the kernel has sent and continues doing “useful things”.
If the user has no more “useful things” and the kernel still hasn’t finished processing all the jobs sent by the user, it waits. User waits until it receives results for all the jobs that it has sent before exiting.

Producer Consumer Implementation:

In “sys_submitjob.c”

A producer and a consumer kthread are created and put to sleep on init of the kernel module.
The producer is woken up whenever module receives a job to perform a feature.

The producer adds the job to a list according to its priority. Jobs closer to list head are of higher priority. Jobs of same priority are placed consecutively in the list. The most recent job is inserted at the tail of all other jobs with the same priority. The queue is limited by max capacity.
If the queue reaches max capacity then producer will return, “Resource temporarily unavailable” to users trying to submit new jobs. It wakes up the consumer when the count of jobs in job queue is 1. The producer goes back to sleep after adding job to queue (and waking up consumer in case of first job) until it is woken up again by the kernel process on receipt of a new job.

The consumer consumes jobs from the queue starting at the head and processes them. After processing the job it interrupts the user and sends the result of the job back to the user via netlink.
The consumer goes back to sleep when the job queue is empty, until it is woken up again by the consumer on arrival of a new job.

Locking Semantics:

A mutex lock is taken to protect job queue and job count. We use this lock in producer, consumer as well as the four queue maintenance jobs.

User operations:

In “perform_job.h” and “process_job.h”

Following operations are provided to the user.

- **Encrypt/Decrypt :** CTR AES encryption and decryption of files with 128-bit key.
- **Compression/Decompression :** LZO compression and decompression of files.
- **Checksum :** MD5 checksum for files(32-bit)**.**
- **Remove Job :** Remove a pending job with some “user_job_id”.
- **Change Priority of Job :** Change the priority of a job with some “user_job_id”
- **List jobs :** list all pending job “user_job_id”s**.**
- **Count of Jobs :** count of pending jobs

All filenames should be absolute paths.

References:

- **Linux Crypto API :** http://lxr.fsl.cs.sunysb.edu/linux/source/include/linux/crypto.h
- **MD5 Checksum :** http://lxr.fsl.cs.sunysb.edu/linux/source/fs/ecryptfs/crypto.c
- **Netlink socket :**
  http://lxr.fsl.cs.sunysb.edu/linux/source/security/selinux/netlink.c
  http://stackoverflow.com/questions/3299386/how-to-use-netlink-socket-to-communicate-with-a-kernel-module
- **signal Handling:** http://people.ee.ethz.ch/~arkeller/linux/multi/kernel_user_space_howto-6.html
- **Kernel Threads:**
  http://www.crashcourse.ca/wiki/index.php/Kernel_threads
  http://www.programering.com/a/MDN4ljMwATk.html
  https://sysplay.in/blog/tag/kernel-threads/
- **Encryption/Decryption:**
  http://stackoverflow.com/questions/17283121/correct-usage-of-crypto-api-in-linux
  http://lxr.free-electrons.com/source/drivers/target/iscsi/iscsi_target_auth.c?v=3.15
  http://lxr.free-electrons.com/source/net/ceph/crypto.c