**A Report on**

**Operating Systems HW-3**

Sai Madhuri Bhavirisetty(109273785)

Siva Sankara Reddy Bommireddy(109293017)

Rini Rachel Bollarapu(109396402)

**Source Files :**

Files related to work queue and job execution are written on our own. Part of the code in other files which were written to get the job done (compression-decompression, encryption-decryption, checksum) is taken from Crypto API. The files written and submitted along with this design report are portrayed below:

* commons.h
* use\_syscall.c
* queue.c
* xjob.h
* sys\_xjob.c
* file\_opshw3.c

Given tasks and our accomplishments for this homework are listed below:

**Task 1: Create a Linux kernel module**

We have created a Linux kernel loadable module named sys\_xjob which when loaded into Linux, will support a new multi-mode system call of the form

int sys\_xjob(void \*args, int argslen)

The job structure named “my\_job” was written in commons.h file which wraps up the ‘job particular’ information like job unique ID, filename, input filename, output file name etc. In addition to this information, there is also a union which has in turn structures for arguments of job types. All the above mentioned structures were written in commons.h file. Commons.h file contains all other macros required by our program.

**Task 2: Instantiation of work queue and kernel thread**

While initiating the sys\_xjob module using init\_sys\_xjob function in sys\_xjob.c file, we are creating a work queue, consumer thread (kthread), wait queues (for both producer and consumer) and a netlink socket. We have used one kernel thread which is consumer thread. In the same file, the function xjob fetches the arguments from our my\_job structure (given by user program) and identifies the job using the job\_type argument and send it to the work queue. The required structure for work queue and the functions that are implemented as a part of work queue implementation are initialized in xjob.h and defined in queue.c .

**Task 3: Waking up of Queue to get the job done**

While initializing the module, as we explained before, we are initializing a consumer thread, so that it would continue running in the background, i.e; it resides in the consumer wait queue until a job appears in the work queue. When a job appears in the work queue, consumer thread wakes up , pops the job from the queue and submits it to processing. When the queue length is less than the maximum queue length, producer will be in producer wait queue. So, when there is no jobs in the work queue, consumer kthread will not run . All the functionality explained above was written in function called consumer\_init in sys\_xjob.c file.

**Task 4: Achieving Asynchronous Processing (Call Back Mechanism)**

We used both netlink socket and signals to achieve asynchrony in job processing. This is because using either of them individually has some drawbacks. If we use only signals, after the user process issues the job, it is processed and then while notifying the user, one has to identify which job is done, for this everytime the job is compared with the all the structures of work queue which is a overhead. If we use netlink sockets, we have to keep continuously polling or selecting which is also a overhead. To overcome these problems, we used them exploiting the properties from both of them. This mechanism was implemented in sys\_xjob.c file (netlink socket initialization) and use\_syscall.c (signal handling and messaging mechanism).

**Task 5: Removing and Listing all queued jobs**

These two tasks are successfully implemented. The code for these tasks can be found in queue.c file. As asked they were implemented using our sys\_xjob() system call itself.

**Task 6: Handling of Locks and Concurrency Issues**

Mutex lock is used for all the work queue mechanism and file\_lock lock is used for locking of files. TO the best of our knowledge, all issues of efficiency, correctness, concurrency, races, deadlocks are handled without any flaw.

**Task 7 : Features our system call supports**

All these features are implemented in file\_opshw3.c file.

* Encryption(or Decryption) : When a file ‘F’ is given along with a key ‘K’ by user process, our code can successfully encrypt and decrypt the file and returns the new file ‘F1’. Our function\_\_\_\_\_ returns 0 on success, otherwise an appropriate error number. Algorithms supported by our function are AES. Part of the code is taken from Crypto API. Related code is present in file\_opshw3.cMethod name : encrypt\_file(): Takes the input file name and the output file name as the input. Reads the input file with block size 4096 bytes and performs encryption using the methods crypto\_alloc\_blkcipher to allocate buffer, crypto\_blkcipher\_setkey to set the key, crypto\_blkcipher\_encrypt of CryptoAPI. decrypt\_file(): Takes the input file name and the output file name as the input. Reads the input file with block size 4096 bytes and performs encryption using the methods crypto\_alloc\_blkcipher to allocate buffer, crypto\_blkcipher\_setkey to set the key, crypto\_blkcipher\_decrypt of CryptoAPI.
* Compression(or Decompression) : We also implemented a function that compresses a given file ‘F’ when a compression algorithm is specified by user program. After the successful compression and decompression, a new file ‘F1’ is created with required output. Our function compress\_file() returns #bytes after compressing, it returns the appropriate error code if anything goes wrong. Compression algorithms supported by our function are Deflate, lzo.
* Methods: compress\_file() and decompress\_file(): Compression is done at the rate of 512bytes blocks. crypto\_alloc\_comp() to load the module related to the algorithm, crypto\_comp\_compress to perform compression and crypto\_comp\_decompress to perform decompression.
* Checksum Calculation:
* We have written code to calculate the check sum for an input file ‘F’. If the algorithm is specified by the user process, after successful completion of the checksum calculation, if the calculation is successful, it will return the check sum value for that file, otherwise it returns the appropriate error code. MD5 checksum algorithm is supported by our code. Part of the code is taken from Crypto API.
* Methods : checksum() method is called which takes hashing algo as the input and computes the checksum value. crypto\_ahash\_init, crypto\_ahash\_update, crypto\_ahash\_final are the methods used to perform this operation.

**Special Features we have created :**

**Notification Flag :**

While the user specifies the job structure, we gave the user a choice of specifying whether he wants to be notified about the completion and results of the job or not. If the output\_mode variable in my\_job structure is set to ‘NOTIFY’, then the user wants to be notified about the job and if the variable is set to ‘SILENT’ . then the user does not want to be notified. The usage of this particular flag is that, only when the flag is set to ‘NOTIFY’, we will send a signal (message) back to the user notifying the outcome of his/her job. Otherwise, we will not initiate the signal mechanism at all.

**One kthread per one core:**

We have created one kernel thread per one core. It leads to increase in performance (in terms of throughput) by maximizing the concurrency when a multi core processor is used.

**Prioritized Work Queue :**

We have designed a prioritized work queue in which the popping out mechanism works on the priority of the job, not FCFS. Our priority mechanism makes use of two level priorities on the basis of ‘NOTIFY’ and ‘SILENT’ flags set by the user process while setting up the job. We made an assertion that “as user sets the ‘NOTIFY’ flag, he/she really care about the outcome of the job and so that job needs special attention to decrease the waiting times of the user”. So, now if the work queue has more than one jobs and one of them has ‘NOTIFY’ flag set, then that job is pre-empted and popped out by the consumer kthread prior to other jobs in the work queue.

**Instructions to run our system call:**

We have submitted Makefile along with all the required submissions which completes our Homework-3. Makefile can be found in hw-3 subdirectory.

* As Makefile is available, using ‘make’ command we can compile the kernel.
* Using sh install-module.sh command, we can install our new module.
* Now, as we are all set we can compile our system call and look for the required output.