CSE-506 Homework Assignment #3

December 3, 2015

GROUP MEMBERS

1. Vivek Tiwari - vitiwari@cs.stonybrook.edu (SBU ID: 110310824)
2. Karan Pugla - kpugla@cs.stonybrook.edu (SBU ID: 110452661)
3. Anish Ahmed - aniahmed@cs.stonybrook.edu (SBU ID: 110560809)

FILES INCLUDED

```
1. common.h
2. xwh3_helper.c
3. xhw3.c
4. generate.sh
5. src/
   |-- 1.common_kernel.h
   |-- 2. common_kernel.c
   |-- 3. sys_submitjob.c
   |-- 4. checksum.c
   |-- 5. encrypt_decrypt.c
   |-- 6. compress_decompress.c
6. tmp/
   |-- 1. ifile*
   |-- 2. ofile*
7. kernel.config
8. Makefile
9. install_module.sh
10.   ../kernel/workqueue.c
11.   ../include/linux/workqueue.h
```

DESIGN DECISION

Work Queue:

The linux workqueue API is used for asynchronous job handling. Workqueue API provides support for adding jobs and job handlers which are executed asynchronously on different threads. It perform locking on resources and puts threads to sleep whenever required, this suits our need(as mentioned in the assignment). Workqueue also takes decides the number threads to use depending on the number of cores of the CPU.

One drawback of workqueue is that it only supports 2 priorities for the jobs added. We have modified the code in the API to support multiple priorities.
We have used Linux Concurrency Managed Workqueue (cmwq), to manage asynchronous processing of jobs. A work item is a simple struct that holds a pointer to the function that is to be executed asynchronously. Whenever subsystem wants a function to be executed asynchronously it has to set up a work item pointing to that function and queue that work item on a workqueue (wq).
Detailed description of workqueue implementation is provided in system documentation.

Signal:
We use Signals to notify user of job completion/failure. The reason we used signals is because it keeps the asynchronous paradigm intact. Unlike other options, like using netlinks, the user program doesn't have to wait on a blocking system call to receive the completion/failure notification of the jobs.

SYSTEM DOCUMENTATION

System Call:

```
long submitjob(void *arg, int argslen)
```

The system call accepts a void *arg which should be of type struct jobRequest_t.

```
struct jobRequest_t{
	int action;		//Job Action. LIST/ADD/REMOVE/CHANGEPRIORITY
	int jobId;		//Job Id (Assigned by system call)
	int type;		//Job Type : COMPRESS/DECOMPRESS/ENCRYPT/DECRYPT/CHECKSUM
	char inputf[1024];	//Input File Name
	char outputf[1024];	//Output File Name
	int delInputf;		//Delete input file after job done? (0 or 1)
	char algo[50];		//Algorithm name
	char passphrase[300]; //Passphrase used for encryption/decryption
	int priority;		//Priority of job. An int in the range (1,100)
	int status;		//Job Status (Assigned by system call). PENDING/COMPLETE/ERROR
	char *buf;		//Useful for system call to pass back list of jobs
	char checksumResult[33];	//Result of checksum calculation
	int result;		//Result of encrypt/decrypt/compress/decompress. File size, 
Error.
};
```

We have used the system call (submitjob) not only for adding a job, but also to remove a job, list all jobs, change priority of a job. The intended action is specified by the field ‘action’.
The system call performs error checking on *arg and gets all jobRequest parameters out of it.

WorkQueue Design:
We have used Linux Concurrency Managed Workqueue (cmwq), to manage asynchronous processing of jobs. A work item is a simple struct that holds a pointer to the function that is to be executed asynchronously. Whenever subsystem wants a function to be executed asynchronously it has to set up a work item pointing to that function and queue that work item on a workqueue (wq). The workqueue is a global variable:

```
struct workqueue_struct *common_queue
```

Special purpose threads, called worker threads, execute the functions off of the queue, one after the other. If no work is queued, the worker threads become idle.

We use the ‘struct work_params’ to store a work/job

```
struct work_params {
	struct jobRequest_t *jobRequest;	//Associated job
	struct task_struct *task;		//Task struct of user process
	struct pid *pid;			//Pid of user process
	struct work_struct work;		//Work struct added to wq
};
```

After processing of a job is done, CMWQ removes it from the queue and free’s the associated work_struct instance of the job.

In order to track jobs that have been processed, we also maintain our own linked list. The linked list element is of type ‘struct workStatusQueue_t’.

```
spinlock_t workStatusQueueLock; //Lock acquired before editing the queue.
struct workStatusQueue_t {
	struct list_head list;			//Linked list head element
	struct jobRequest_t *jobRequest;	//Associated job
	struct work_struct *work;		//Work struct added to wq
};
```

This list is used to track status/result of all jobs (pending/errored/completed). The spinlock workStatusQueueLock is acquired anytime a addition/deletion of list element is performed.

Priority of a job:
Linux cmwq supports two priorities of a job queue, high priority and normal priority (Using flag WQ_HIGHPRI). All jobs added to a workqueue will be given the same priority that the whole queue has. It is not possible to schedule jobs differently, based on its priority within a queue. The jobs are queued to tail of a linked list, and are taken out from the front for processing (FIFO).

In order to support an integer priority, and processing jobs based on that, we modified the code of cmwq (workqueue.h and workqueue.c) and added the following methods.

```
static inline bool queue_work_priority(struct workqueue_struct *wq, struct work_struct *work)
bool queue_work_on_priority(int cpu, struct workqueue_struct *wq, struct work_struct *work)
static void __queue_work_priority(int cpu, struct workqueue_struct *wq,struct work_struct
*work)
static void insert_work_priority(struct pool_workqueue *pwq, struct work_struct *work, struct
list_head *head, unsigned int extra_flags)
```

The main change was done in insert_work_priority, where instead of adding a new work directly to the tail, we added it just after a work that has a higher priority than the new work to be inserted. This resulted in the cmwq work linked list to be always sorted in decreasing priority, and it always picked a job (at the head), that had the highest priority. Excerpt from the function:

```
params = container_of(work, struct work_params, work);
list_for_each_safe(pos, q, head) {
       temp = list_entry(pos, struct work_struct, entry);
       temp_params = container_of(temp, struct work_params, work);
       if(temp_params->jobRequest->priority > params->jobRequest->priority){
             list_add(&work->entry, prev);
             added = 1;
             break;
       }
       prev = &(temp->entry)
}
if(added == 0){
       list_add_tail(&work->entry, head);
}
```

Adding a job:
When a new job is queued by calling submitjob with jobRequest.action = ADD_JOB, a unique job_id is assigned to a jobRequest, and the status is marked as PENDING. A work_struct is made out of the job and added to the CMWQ workqueue common_queue. The function queue_worker is specified as the worker function. The job is added in decreasing order of priority.
Additionally the job is also added to the workStatusQueue.

<u>ENCRYPTION/DECRYPTION:</u>
Takes absolute path of file and passphrase using which file has to be encrypted/decrypted. An additional flag for deleting input file after job is done is also included.
MD5 of the passphrase and original file size is stored in the header of the encrypted file. Padding is added to file to extend the size of the file to the nearest multiple of 16. The file is encrypted using symmetric cbc(aes) algorithm. Encryption is performed in chunks of PAGE_SIZE.
While decryption the MD5 of the provided passphrase is matched with MD5 of the original passphrase from the header. If they don't match error is returned. If they match, file is read in chunks of PAGE_SIZE and decrypted using the passphrase. The size of the original file is extracted from the header and used to detect the mark till which the encrypted file is to be read.

<u>COMPRESSION/DECOMPRESSION:</u>
The following compression/decompression algorithms are implemented-

1. lzo
2. lz4
3. deflate

The system call is passed the absolute path of the file to be compressed/decompressed and the job output file name. It is also passed the algorithm using which compression/decompression is to be performed.
While compression, the file is read in chunks of PAGE_SIZE and compressed. Each compressed chunk is written to file with a header attached before it. The header has the following structure-

```
struct compressChunkHdr_t{
        char algo[10];
        loff_t length;
        loff_t origLength;
        loff_t nxtChunkOffset;
```

```
};
```

This has the algorithm name using which compression was performed, the length of the compressed chunk, the offset at which next chunk(header inclusive) is written to file.

While decompression, the header is read and the length of the compressed segment is extracted. Decompression is applied on chunk of this size. Then file is read from the nxtChunkOffset value in the header. The header is extracted and decompression is performed the same way repeatedly till end of file.

CHECKSUM:

The result provided by this job is similar to the result of system call ‘md5sum’. The absolute path of the file to be encrypted is provided.

The file is read in chunks of 16 bytes. The value of the hash is updated with every 16 bytes chunk read. Finally the checksum is copied back to checksumResult member of jobRequest_t object.

Job Processing:

CMWQ invokes the worker function: queue_worker. The function fetches all job parameters (jobRequest) by getting the container of work_struct.

Depending on the jobRequest->type, we know what exactly the job is, eg. Encryption on file infile, using algorithm X etc.

The job is processed and finally we store the job result in the appropriate place (result/checksumResult) in jobRequest. If an error is encountered, then it is stored in result.

Notifying on job completion:

After a job is processed, the queue_worker sends a signal to the user task struct stored (We store pid in work_params->pid). A check is performed to see if the user program is alive or terminated, and only if it still exists, we send out the signal (SIGIO)

```
task = get_pid_task(params->pid, PIDTYPE_PID);
if (task != NULL) {
	send_sig(SIGIO, task, 0);
	put_task_struct(task);
}
```

The user process has a signal handler, which on being triggered fetches the list of all jobs (Using syscall, explained below) that are in completed/error state and prints it out to the users. Therefore, if the user process is alive it will get the results of asynchronous jobs as and when they get processed.

Listing all jobs:

All jobs added to queue can be listed by call submitjob with jobRequest. It is essential to set the parameters below:

```
jobRequest->action = LIST
jobRequest->type = 0 or 1 (To specify if we want all jobs, or just the completed/errored ones)
jobRequest->buf      A user malloced buf where the system call will copy the results
```

The system call invokes function copy_work_status that loops through all jobs (workStatusQueue_t *workStatusQueue) and adds status of each job into the user buf. Additionally, each job that is reported to user is removed from the list. (As we are now done after listing its results to user).

Removing a job:

A job can be removed from the queued jobs by calling submitjob with the following parameters:

```
jobRequest->action = DELETE
jobRequest->jobId = Job Id of job that needs to be removed
```

We first cancel the work queued on CMWQ by calling the API provided:

```
err = cancel_work_sync(work);
```

If err is equal to 1, the job was succesfully cancelled by CMWQ. Otherwise if the job didn’t exist in the queue or is being currently processed, than err return is 0. On success, we also remove the job from our list workStatusQueue.
Syscall returns approproate success/error code.

Changing priority of job:

A job’s priority can be changed by calling submitjob with the parameters:

```
jobRequest->action = PRIORITYCHANGE
jobRequest->jobId = Job Id of job that needs to be removed
jobRequest->priority = New priority of job
```

We first find the job in our queue and verify that the new priority is not the same as the one set before. If not, then we cancel the job (remove job), and then add it again to the CMWQ with the new priority.

Test cases

The syscall implementation has been rigorously tested for multiple test cases. The test cases submitted can be run as follows -

1. generate.sh -

    sh generate.sh <number of input files>

    The command generates ifile* files which can be used by the user program.

2. xhw3.c

    ./xhw3 add_many <number of jobs> <stage> <wait flag>

    where stage is 1 when we encrypt, compress and calculate checksum for input files and 2 when we want to decrypt and decompress files.
    If wait flag is set to 1 then user program is never terminated and can perform useful work without waiting for reply from system call. If flag is set to 0 then user program terminates as soon as all jobs are submitted.

    ./xhw3 list_jobs

    command is used to list all the jobs which were in submitted. This will give the status for all jobs.