Operating Systems Design Document (Home Work 3)

Name : Bharath Kumar Reddy Vangoor
SBU ID : 110168461
Date : 12/03
Group ID : HW3-BVANGOOR

Design :

Implemented a user program (submitjob), asynchronous system call (sys_submitjob) which will take arguments from the user and pass it on to the kernel system call "sys_submitjob" which executes the job asynchronously in background and notify the user when it is finished. System call is implemented as kernel module. Following is the flow of my user program to kernel and back to user level (in higher level):

1. User program (submitjob) is invoked with proper arguments, after verifying the arguments passed by the user are correct, the user program packs the arguments passed in to a "struct arguments" and type casts it to "void" and passed it to kernel system call.
2. But before that the system call module should be inserted to kernel modules, as soon as it is inserted a consumer thread is created but not started as running.
3. When the system call is invoked from user program, the arguments are copied into the kernel space and verified.
4. Depending upon the job type the kernel program first checks the import want arguments like "input file", "output file" whether are accessible and if the checks pass the job is added to the work queue and a success return value is passed to the user.
5. The work queue, jobs counter is protected by the MUTEX lock since there can be multiple users (producers) can invoke the call.
6. As soon a new job gets added to the work queue, the kernel process that adds job (producer) will check the counter (no. of jobs) value and it is 1, then wakes up

the consumer thread that was created during module “init” method.
7. And when the consumer wakes up it picks up the job that is at the head of the work queue and proceed to run the job. After the job is done it sends the return code back to the user process through netlink socket that was created in “init” module method using pid of the user that was sent as part of arguments.

Producer Logic :

1. Grab the MUTEX lock
2. Check if count of number of jobs is greater or equal to MAX_JOBS, if true then error with -EBUSY but unlock the MUTEX lock
3. If the number of jobs are less then add the job to work queue and increment the counter(no of jobs).
4. if count of jobs is equal to 1 (i.e first job) then wake up the consumer.
5. Release the MUTEX lock.

Consumer Logic :

1. First thing check for “kthread_should_stop” because during the “exit” module calls the consumer should get killed.
2. Grab the MUTEX lock. (if above check is not true)
3. Check whether the counter is equal to 0 (i.e No jobs) then set task as “TASK_INTERRUPTIBLE”, release the lock and yield to scheduler. But we should grab the lock as soon as we wake up that is after the “schedule()” call.
4. We again check for “kthread_should_stop” to check whether consumer is made run by exit method.
5. If not, then remove the first job from queue, decrease the counter.
6. Release the MUTEX lock.
7. Process the job.
8. Send the error/success return value to the user using net link socket.

Design Decisions :

1. Used MUTEX lock to protect the work queue, counter because the time taken to process the requests are more.
2. Used Net link sockets to communicate between User process and Kernel process because using Net links there is a option to send data between them but using signals we can send only one return value. And I used Non blocking Net link sockets through which user can do some other work and get notified when kernel sends some message to it.
3. Used First Come First Serve (FCFS) policy to serve the jobs that get added to the work queue.
4. Used double linked list for implementing the work queue as I felt reordering of the jobs will be easier.
5. Implemented “list”, “remove”, “reorder” tasks as synchronously where as tasks like “encryption”, “decryption” are sone asynchronously.
6. I’ve kept MAX_JOBS at a time as 10 but this can always be configured like can be increased, decreased.
7. Started Kernel consumer thread, netlink socket, linked list initialisation of head during the “init” Module method as that is the starting place of the sys call.
8. I assigned each job a unique ID so that it can be identified from the user program, this is done by initialising a global counter at “init” method and whenever a new job comes incrementing it and passing the same value to user when successful.
9. Passing a “structure list” that gets added to the arguments when user wants to list the current jobs in pending. This list will be filled by the kernel with current jobs in the work queue (of course by acquiring lock) and returned to user program upon which it prints to console.
10. Used encryption, decryption form HW1 submissions.

Features Supported :

The implementation supports following operations (tasks) :

1. **encryption** (job name : “encrypt”) : Expects input file, output file, algorithm name (but supporting only one cbc(aes), so giving other names have NULL effect) and password (greater than 6 characters). Which encrypts the content in input file and writes to the output file. ASYNCHRONOUS JOB.
2. **decryption** (job name : “decrypt”) : Expects input file, output file, algorithm name (but supporting only one cbc(aes), so giving other names have NULL effect) and password (greater than 6 characters). Which decrypts the content in input file and writes to the output file. ASYNCHRONOUS JOB.
3. **list** Jobs (job name : “list”) : Expects no arguments and returns the pending process that are waiting in work queue(if any). SYNCHRONOUS JOB.
4. **remove** Job (job name : “remove”) : Expects unique id of the job submitted to remove from the work queue. SYNCHRONOUS JOB.
5. **reorder** Job (job name : “reorder”) : Expects unique if od the job, position to which the job has to be ordered in the work queue. SYNCHRONOUS JOB.

Major Data Structures :

1. Arguments structure :

```
struct arguments {
        int unique_code; /*Some thing like process id for each job*/

        int operation_type; /*encryption, decryption,list jobs, remove
job, reorder job*/

        char *input_file_name; /*can be more than one*/

        char *output_file_name; /*output file*/

        char *algorithm_name;   /*cipher name, compression name,
checksum name for encryption, decryption*/

        char *pass_phrase;  /*pass key used in ecryption/decryption*/

        int delete_orig_file;  /*flag to delete input file*/

        int pid; /*Process ID for kernel to know which one called*/

        int position; /*for passing position to reorder the job*/

        void *list; /*extensibility for list operations used in list
jobs*/
    };
```

2. For listing the jobs form kernel, passing the following Data structure into which the kernel copies :

```
struct jobs_list {
      int pid[MAX_JOBS];
      int uid[MAX_JOBS];
      int Jid[MAX_JOBS];
      int count;
 };
```

The above structure is added to the “void *list” in arguments structure when calling list jobs.

3. Kernel Work queue :

```
struct node {
      struct arguments *job;   /*pointer to the job*/
      struct node *next;
      struct node *prev;
 };
```

Major Functions :

User level functions :

1. allocate_arguments(int, int, int, int);
Which take sizes for input file, output file , algorithm name, pass_phrase and returns a allocated structure object.

2. free_arguments(struct arguments *);
free the arguments object one by one.

3. check_operation() which will check whether the task passed by user is correct or not.

4. socket(PF_NETLINK, SOCK_RAW, NETLINK_USER); create a netlink socket
5. bind(sock_fd, (struct sockaddr *)&src_addr, sizeof(src_addr));
Binding to the socket.
6. recvmsg(sock_fd, &msg, 0); For receiving message from nettling from kernel.

Kernel level functions :

sys_submitjob.c
1. Consumer Function : consumer_fn() : The method that acts as consumer.
2. Producer code. Not written as a function but included in main system call.

3. kthread_create(consumer_fn, NULL, our_thread); For creating consumer thread.
4. kthread_stop(thread1); For stopping the consumer thread in exit module method.
5. netlink_kernel_create(&init_net, NETLINK_USER, &cfg); Creating net link socket.
6. mutex_lock(&list_lock); MUTEX lock
7. mutex_unlock(&list_lock); MUTEX unlock

lists.h
1. allocate_node(void) Allocating node object
2. delete_node(struct node *old) Freeing the node object
3. delete_job(struct node **head, int job_id) For removing the job from work queue by searching job_id.
4. insert_node(struct node **head, struct node *new) For adding new job to the list.
5. get_head_of_list(struct node **head) For removing job at the head of the list
6. reorder_list(struct node **head, int pos, int val) For recording the job with unique id "val" to position within the list.
7. delete_entire_list(struct node **head) To delete entire list used in "exit" module method.

encryption_decryption.h
1.encrypt_decrypt(struct arguments *job) For encrypting, decrypting the job.
2. file_rename(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry)
For renaming the file from old to new
3. file_unlink(struct inode *dir, struct dentry *dentry) For unlinking the file
4. generate_md5(char *src, char *dest, int len) generating MD5 hash.
5. encrypt_Cipher(char *key, char *src, char *dest, unsigned int len, int *written) For encrypting
6. decrypt_Cipher(char *key, char *src, char *dest, unsigned int len) For decrypting
7. buildPadding(char *array, int val), reconstructPadding(char *array) For taking care of padding.

file_operations.h

1. check_input_output_file(struct arguments *arguments) To verify input, output files before adding to work queue.

2. wrapfs_read_file(struct file *filp, void *buf, int len) Read from file
3. wrapfs_write_file(struct file *filp, void *buf, int len) Write to file.

Locking Semantics :

Used MUTEX lock to protect list, counter. When ever producer adds new job to list he grabs MUTEX lock and releases it, similarly consumer grabs MUTEX lock and removes job from queue and releases it. Similar actions are done by jobs like “list”, “remove”, “reorder”.

Queue Behaviour :

Queue is First come First Serve (FCFS) so the jobs that are added first are executed first and vice versa.

References :

1. Net link sockets : http://stackoverflow.com/questions/3299386/how-to-use-netlink-socket-to-communicate-with-a-kernel-module
2. Free electrons to browse the code and check the implementations.(http://lxr.free-electrons.com/source/)
3. fs/open.c, fs/read_write.c for reading and writing.
4. Documentation/crypto/api-intro.txt for crypto api implemntations
5. Wrapfs implementations for vfs_unlink and vfs_rename (http://lxr.fsl.cs.sunysb.edu/linux/source/fs/wrapfs/inode.c#L231)
6. fs/ecryptfs/crypto.c for encrypting, decrypting and also MD5 hash generation in kernel.
7. Professor's hw1.txt file containg the code for reading and writing.
8. Patch provided by the Professor for initial start of syscall implementation.