Operating Systems

Homework Assignment #6

Due 16.6.2016, 23:50

Part 1

In this part of the assignment you will write an HTTP server.

The HTTP server listens for connections, with workers handling clients in the background. Whenever a client connects, its socket is handled by one of the workers.

The program will receive two arguments: number of workers and port number. Port number is an **optional** argument, use default HTTP port if not provided. Max requests (parameter of *listen*) can be set to # of workers.

Choose whether to implement workers as threads or processes. Both options are ok, and are up to you, however the implementation differs quite a bit. You should come up with a solution that divides work properly between the workers. You may spawn workers in advance, or upon connection of a client. Your choices should match the option you chose, and should be explained (and argued) in your answers, as detailed below.

On any error with a client, the server should return an HTTP response of *Internal Server Error* (HTTP 500) and close the connection. It is ok to do that in the main thread/process if that is where the error originated.

In each worker, when handling a client socket:

- Read the client's HTTP request and parse it.
  - Only support GET/POST operations – return *Not Implemented* (HTTP 501) for any others.
  - Do not parse **any** info except the request line. Ignore request headers, cookies, etc.
- Return a minimal HTTP response to the client according to the request
  - Find the requested file on your machine (treat it as a path).
  - If it is not found, return *Not Found* (HTTP 404) response (including a simple body).
  - If it is a directory, return an HTTP response with an *HTML* body listing the directory's contents.
  - If it is a file, return an HTTP response with the file's contents as the body.
- Close the socket.

Pressing CTRL+C (SIGINT) should exit the server **properly**! Terminate all workers, close sockets, etc. You can & should wait for workers to finish handling any current clients before terminating (i.e., empty the queue).

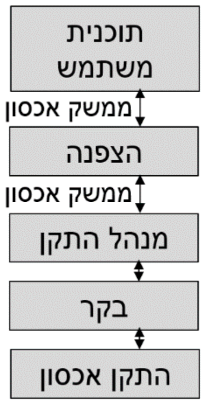
You can test your server using your browser – which should work with your server.
Navigating to URL $http://localhost:<port>/<file>$ should display the contents of <file> (which can also be a path, such as "`~/Downloads/note.txt`".

Guidelines

- Clearly document your code. Explain your solution in a main comment, argue your choices and correctness. Explain the benefits of your chosen solution compared to other possibilities.
- Use **strtol** (man 3 **strtol**) to convert a string to integer. You may assume input is valid and in correct range.
- Do not slave over parsing strings, assume all data is in valid format (address, HTTP request, etc.)
- **Submit:** `http_server.c`.

Part 2

In this part of the assignment you will implement an “encrypted storage” kernel module, for storing data in encrypted form.



When accessing the device, from the user’s perspective it behaves like normal persistent storage of data, which can be read from and written to as any file. However, the device will use VFS operations to store the data in **encrypted** form in the filesystem.

A device of this module is a *plaintext* device, which is associated with a *ciphertext* file in the filesystem (a standard file, which itself could possibly be a device) and a *key* file. Data written to the plaintext device will be encrypted using the key file, and the ciphertext (the encrypted data) stored in the ciphertext file. Reading from the plaintext device will invoke reads from the ciphertext device; the ciphertext data will be decrypted with the key file and returned to the user. Thus, from the user perspective, reads and writes are consistent and produce the expected results, while in the filesystem the data is stored in encrypted form.

When the user first opens a device file of your module, it is not associated with any ciphertext or key files. The user sets or changes these using *ioctl* commands, as specified below. However, until the user sets both files, the device is unusable and all relevant commands should fail with *–EINVAL*.

Your device should support the following:

- Initialization & Cleanup – should be as minimal as possible. If basing on existing code, remove unnecessary variables, initializations, allocations and deallocations, etc.
- Open & Release – multiple devices may be opened in parallel, and each opened device file can be configured with its own ciphertext and key files (which may also be changed, see below). However, when first opened both should be marked as NULL (i.e., not initialized by the user). *Be careful with memory allocation!*
- Read – to read data, read ciphertext from the ciphertext file, decrypt it with the key (see below), and return the decrypted data to the user.
- Write – to write data, encrypt the user buffer using the key (see below), and write the resulting ciphertext to the ciphertext file.
- Seek – perform a seek operation on the ciphertext file. Note that you should seek the key file to a matching position as well, to encrypt/decrypt correctly.
- Set ciphertext file – this will be done using *ioctl*, where the *cmd* is **1**, and the *arg* is a filename string of the new ciphertext. Make sure you reset/update the position of the key, and treat user flags correctly; some require special attention!
- Set key file – also via *ioctl*, where the *cmd* is **3**, and the *arg* is a filename string. You should determine the size of the key file and store it, for use by the *read*/*write* and *seek* operations.

Encryption:

The encryption and decryption process is identical. When the input data is plaintext (data that is not encrypted), it produces ciphertext (encrypted data). When the input data is ciphertext, it produces the original plaintext. This process is as follows.

The encryption is performed by bitwise XOR-in the contents of the input and the key file. The position of the data should match the position of the key, such that the 1st byte of the ciphertext file was encrypted with the 1st byte of the key file, etc. Note that the key file can be of any size. In case it exceeds the size of the ciphertext file, only part of the key file will be used. If the ciphertext file size exceeds the size of the key file, you should reuse bits of the key file by restarting from the start.

For example, if the user read bytes 1000-2000 from the device, these correspond to the same positions in the ciphertext file. If the key file size is 500 bytes, it will be read twice, such that byte 1700 of the ciphertext file will be decrypted with byte 200 of the key file. *Be careful with memory handling!*

Guidelines:

- You may place a size limit on filename strings and return an error if commands exceed it.
- Your module will implement a *character device*
- To implement ***ioctl*** in your module, use the ***unlocked_ioctl*** function pointer in *struct **file_operations***. Note that it differs a bit from what we saw, its declaration is:
  `long (*unlocked_ioctl) (struct file *, unsigned int cmd, unsigned long arg);`
- The ciphertext and key files are *specific to each opened device file!* If the user opens two device files associated with your module, changes to the choice of files in one should not affect the other. Even if it is the same device file! (i.e., with the same major/minor combination).
- Note that **no locks** are needed for the implementation.
- Make sure to follow all of the rules we learned for kernel programming. Be careful with user space addresses, and remember that calls may fail – and that is OK as long as errors are properly propagated back to the user with reasonable error values. Read/Write operations may read/write (from the ciphertext file) less than the user requested, and it's also ok! In such a case, just use whatever data was read/written in a single call – don't repeatedly read/write to reach the length requested by the user. This is not true for the key file, which *should* be repeatedly read from to match the length read/written.
- Do not assume any read/write command returns the number of bytes requested, nor the seek command seeking to the position requested.
- **Submit**: **`cipherdev.c`**, your modified **`Makefile`**.

VFS Operations

As we've learned, user-space operations are not allowed within kernel modules, since they are executed in kernel mode.

The existing system calls to access files (open/close, read/write, etc.) are in user-space, and thus we cannot use them. Instead, we perform calls to the VFS directly – to functions of the kernel itself. These calls will replace the file I/O system calls we've learned.

First, before **each** VFS call, we must update our address space (from kernel space to user space), and **return it once finished**.

Thus, use the following code right **before each** VFS call:
mm_segment_t oldfs = get_fs();
set_fs(get_ds());

Right **after each** VFS call, even before checking errors, call the following to restore the address space:
set_fs(oldfs);

To open or close a file, use the calls *filp_open, filp_close*. These calls act the same as *open/close* system calls, except using the kernel's *struct file** instead of a file descriptor. The *filp_close* function receives a second argument, which should be *NULL* (the first is *struct file**).
To check for errors, use the *IS_ERR* macro on the *struct file** result of *filp_open* (e.g., *if (IS_ERR(myfile))*), and use the *PTR_ERR* macro to get the error code. Since *strerror* is also a user-space function, printing the error code (along with a descriptive message) is enough – no error string is needed.

To read, write, or seek, use the following: *vfs_read, vfs_write, vfs_llseek*

These functions are, again, the same as the equivalent system calls, except receiving the kernel's *struct file** instead of a file descriptor. The return value of all of these functions is *int*, just as the equivalent system calls.

Use the following header files (at least):

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <linux/string.h>
#include <linux/buffer_head.h>
```