UP23 Lab05

Date: 2023-04-24

Does Sharing Memories Make Us Feel Closer?

Shared memory is the most efficient way to exchange data between processes. In this lab, we aim to implement a simplified share memory mechanism as a kernel module that offers persistent share memories in the kernel for data exchange between user-space processes.

Preparation

You may not have experience in implementing a kernel module. Before you start your implementation, you may read some relevant kernel documents and tutorials.

- Please check the `file+stdio` course slide and read the `hellomod` example to see how a simple kernel module is implemented.
- The Linux kernel documentation (https://www.kernel.org/doc/html/latest/), including
  - ioctl based interface (https://www.kernel.org/doc/html/latest/driver-api/ioctl.html)
  - Memory allocation guide (https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html)
  - Memory management APIs (https://www.kernel.org/doc/html/latest/core-api/mm-api.html)
- Linux kernel memory mapping (https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html) labs materials from the OS course offered by the University Politehnica of Bucharest. ***Note:*** this is for kernel 5.10.14 and the introduced functions might be different from our lab experiment runtime.

Our development package (including runtime and development files) can be found here (dist.tbz) (https://up23.zoolab.org/up23/lab05/dist.tbz). You may download and play with it before our lab officially starts.

You can develop the module on Apple chip macs but note that all the files must be cross-compiled to x86_64 architecture. Please read the rest of the lab instructions carefully before you implement this lab.

> Our course video `file+stdio` has introduced how `ioctl` works with a kernel module. This lab extends it by implementing more features in the kernel module.

The Challenge Server

The challenge server can be accessed using the `nc` command:

```
nc up23.zoolab.org 10548
```

Upon connecting to the challenge server, you must first solve the Proof-of-Work challenge (ref: pow-solver (https://md.zoolab.org/s/EHSmQ0szV)). Then you can follow the instructions to upload your kernel module for evaluation. For how the hints to implement the module, please refer to the specification and hints for more details.

To simplify the uploading process, you can use our provided `pwntools` python script to solve the pow and upload your kernel module file. The upload script is available here (view (https://up23.zoolab.org/code.html?file=up23/lab05/submit_e803f71b7df255232bc91b3bb1f94412.py) | download (https://up23.zoolab.org/up23/lab05/submit_e803f71b7df255232bc91b3bb1f94412.py)). You have to place the `pow.py` file in the same directory and invoke the script by passing the path of your solver as the first parameter to the submission script.

> Please note that the submission script of this lab is ***different*** from other labs because the QEMU emulator outputs `\r\n` as its newline character instead of the typical `\n` on UNIX machines. To ensure that `pwntools` can correctly interpret lines returned from the server, we have an additional line `r.newline = b'\r\n'` in the script to interpret outputs properly.

Specification

The specification of the kernel module is summarized as follows.

1. The module has to ***automatically*** create 8 devices in `/dev` filesystem (from `kshram0` to `kshram7`). Each device corresponds to a kernel memory space of 4KB (default) allocated in the kernel using the kzalloc (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/slab.h#L713) function. You may use an array of customized data structures to store your required information.

   > Note that `kzalloc` has several limits as mentioned here. For example, the maximal size of a chunk that can be allocated with `kzalloc` is limited. However, for simplicity, our test cases do not allocate more than the limitations.

2. Please remember to release allocated memories using the kfree (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/slab.h#L211) function when you unload the module.

3. The size of each shared memory file can be listed from `/proc/kshram` . A sample output is shown below.

   ```
   00: 4096
   01: 4096
   02: 4096
   03: 4096
   04: 4096
   05: 4096
   06: 4096
   07: 4096
   ```

4. The module has to support the following `ioctl` commands, defined in `kshram.h` .

   ```
   #ifndef __KSHRAM_H__
   #define __KSHRAM_H__

   #include <asm/ioctl.h>

   #define KSHRAM_GETSLOTS _IO('K', 0)
   #define KSHRAM_GETSIZE  _IO('K', 1)
   #define KSHRAM_SETSIZE  _IO('K', 2)

   #endif /* __KSHRAM_H__ */
   ```

   The purpose of each `ioctl` command is explained below

   - **`KSHRAM_GETSLOTS`** returns the number of slots available in the module. It should be 8. This command does not have an additional argument.

- **`KSHRAM_GETSIZE`** returns the size of the shared memory corresponding to the opened device. You should manage the size of each allocated memory internally by yourself. This command does not have an additional argument.
- **`KSHRAM_SETSIZE`** resizes the size of the shared memory file based on the third parameter passed to the `ioctl` (https://man7.org/linux/man-pages/man2/ioctl.2.html) function. Given an allocated memory pointer, you may resize it by using the krealloc (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/slab.h#L210) function in the kernel. This command uses an additional argument to pass the size to be set.

  > Note that our test cases always resize a shared memory only when the shared memory is not in-use.

5. Your module must support `mmap` file operation. In the `mmap` file operation, you have to map to memory allocated in the kernel to user-space addresses so that the user-space program can access it directly.
6. We use three simple programs to evaluate your implementation. You can read these codes to see how the user space program interacts with your module.
   - `check_msg` (view (https://up23.zoolab.org/code.html?file=up23/lab05/kshram/check_msg.c), download (https://up23.zoolab.org/up23/lab05/kshram/check_msg.c)): The program opens `/dev/kshram0`, calls `mmap` to map the memory, prints out the message in the shared memory, and updates the shared memory with a randomly generated message.
   - `check_resize` (view (https://up23.zoolab.org/code.html?file=up23/lab05/kshram/check_resize.c), download (https://up23.zoolab.org/up23/lab05/kshram/check_resize.c)): The program iteratively opens each device, resizes the memory, validates the size of the resized memory, and clears the content of the memory.
   - `check_fork` (view (https://up23.zoolab.org/code.html?file=up23/lab05/kshram/check_fork.c), download (https://up23.zoolab.org/up23/lab05/kshram/check_fork.c)): The program creates two child processes and asks them to map the shared memory of `/dev/kshram0` independently. Once done, one process starts writing a random message into the mapped memory, and the other verifies the message's correctness in the shared memory.

Lab Hints

Here are some hints for you. You can implement your kernel module locally first and then verify your solution on the challenge server.

1. The required development files can be downloaded from here (dist.tbz) (https://up23.zoolab.org/up23/lab05/dist.tbz). The archive contains a Linux kernel, an `initrd` filesystem, test executables, a script to run the emulator, and files required to build a kernel module.

   > The kernel version we used in the development file archive is 6.2.6 (https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.2.6.tar.xz).

2. Please install the qemu system emulator in your development platform. For Ubuntu-based dockers, you can install it using the command `apt install qemu-system-x86`. It would work on both Intel and Apple chips. You can even install the native one on Mac by using `brew install qemu`.

3. Once you have the qemu system emulator, you can simply type `sh ./qemu.sh` to boot the Linux kernel in a virtual machine. The current design uses the archive `rootfs.cpio.bz2` as the `initramfs` image. You can add more files in the filesystem by extracting files from the archive, adding files your want, and re-packing the archive.

4. If you plan to have your files in the `initramfs` image, you can extract the files using bzip2(1) (https://linux.die.net/man/1/bzip2) and cpio(1) (https://linux.die.net/man/1/cpio) utilities, and re-pack the image using the same tools.

   > You may need to set the cpio format to `newc` format. Also please ensure that you pack all the required files in the image.

5. A sample `hello, world!` module is available here (hellomod.tbz) (https://up23.zoolab.org/up23/lab05/hellomod.tbz). You may implement your module based on the `hello, world!` example. It has sample file operations and `/proc` file system implementations.

   > In the `qemu` virtual machine runtime, you can use the commands `insmod` and `rmmod` to install and remove modules, respectively.

6. You can use a single device class to manage multiple devices. Based on the `hello, world` module, once you have created the class, you can then use the `device_create` function to create multiple devices. Note that the `device_create` accepts a format string for the device name. You may leverate it for easier device creation.

7. You may have a look at the file_operations (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/fs.h#L2091) data structure for creating new file

operations such as `mmap` .

8. We mentioned that you can use `kzalloc` to allocate memory spaces. To prevent the memory from being swapped out, you may also check the implementation of the SetPageReserved (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/page-flags.h#L383) function. Also remember to clear the `reserve` flag by calling the ClearPageReserved (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/page-flags.h#L390) function.

9. You may want to have ***private data*** associated with an opened file. To do this, you will need to ...
    - Define a customized data structure for your private data.
    - When opening a file, allocating a space using kzalloc (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/slab.h#L713) for the customized data structure.
    - Assign the pointer of the allocated space to `file->private_data` .
    - Remember to release the spaces of the `file->private_data` using the kfree (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/slab.h#L211) function.

10. The function you can use to map kernel space memory to user-space processes is the remap_pfn_range (https://elixir.bootlin.com/linux/v6.2.6/source/include/linux/mm.h#L3025). You may also want to check how virt_to_page (https://elixir.bootlin.com/linux/v6.2.6/source/arch/x86/include/asm/page.h#L69) and page_to_pfn (https://elixir.bootlin.com/linux/v6.2.6/source/include/asm-generic/memory_model.h#L52) work.

11. A kernel module can only call functions exported from the kernel using `EXPORT_*` macros. To have more flexibilities for calling (un-exported) kernel functions, our kernel has exported the kallsyms_lookup_name (https://elixir.bootlin.com/linux/v6.2.6/source/kernel/kallsyms.c#L271) function. Similar to the dlsym(3) (https://man7.org/linux/man-pages/man3/dlsym.3.html) function, you can use it to look up and resolve function symbols available in the kernel.

    Note that you should not need the ***kallsyms_lookup_name*** function when implementing this module. We leave it here just in case you want to use it.

12. A successful running output from the challenge server should look like this example (https://up23.zoolab.org/code.html?file=up23/lab05/output.txt).

Additional Notes for Apple Chip Users

If you do not have a working x86_64 machine, you can still build modules on an Apple chip Mac locally. Here are some hints for solving this challenge on Apple chip Macs.

1. The kernel module must be developed in a Linux docker. But the QEMU emulator (`qemu-system-x86_64` invoked from `qemu.sh`) can be used in either a Linux docker or an Intel/Apple-chip Mac. For simplicity, we recommend you to do everything in a Linux docker.
2. You must install the compiler for x86_64 platform. The compiler and the emulator required for cross-compilation can be installed using the command:

   ```
   apt install gcc-multilib-x86-64-linux-gnu qemu-user-static
   ```

3. Once you have installed the required packages, you can then build a module inside the module source directory using the command:

   ```
   make ARCH=x86_64 CROSS_COMPILE=x86_64-linux-gnu-
   ```

   You can try it with the `hello, world!` module.

Grading

We have eight demo items here. To demonstrate yourimplementation, you must *always* perform from step 1 to step N (N <= 8). You cannot demo only a single item (except item #1) and get the points. You can demonstrate items #1 and #2 in your local machine. For items #3 to #8, you can simply upload your module to our service and our service will run the test cases automatically.

1. [10 pts] You can boot the Linux VM in a supported environment.
2. [10 pts] You can put the `hello, world!` module into the emulator and load it.
3. [10 pts] You can put your `kshram.ko` module into the emulator and automatically create 8 device files (from `kshram0` to `kshram7`) in `/dev` after loaded the module.
4. [5 pts] Unloading the kernel module does not crash the system (***Module unload test #1***).
5. [20 pts] Load your module again, and you can pass the test using the `check_msg` program.
6. [20 pts] You can pass the test using the `check_resize` program.
7. [20 pts] You can pass the test using the `check_fork` program.
8. [5 pts] Unloading the kernel module does not crash the system (***Module unload test #2***).

We have an execution time limit for your kernel module. You have to pass the tests within 120s (kernel boot time inclusive).