CS 238: Advanced Operating System - Mid Term Report
Camera Virtualization

Team members:

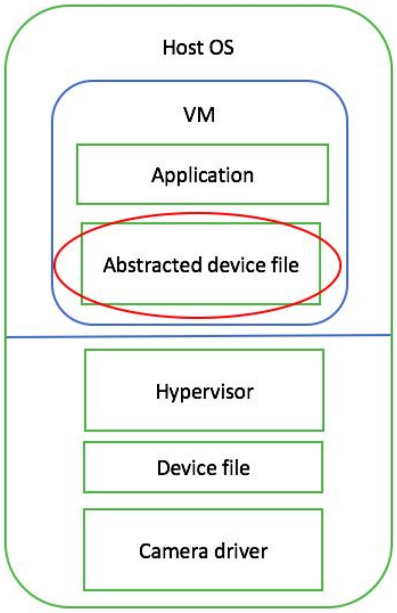
1. Arpita Venkatesh Huddar - ahuddar - 87009145
2. Avinash Nath Aita - aaita - 88289772

Introduction:
Virtualization has gained its importance for containerizing and providing isolation between multiple VMs that can be run on host hypervisor/OS. One of the three major components, I/O virtualization is useful to share the I/O device (such as GPU, Ethernet, Camera) across guest VMs. I/O virtualization is much harder compared to CPU and memory virtualization due to the diversity in their implementations. Many approaches have been proposed so far. For UNIX systems in specific, I/O devices are abstracted with Device files (found on /dev). Our project is an attempt to use the device file associated to the camera on the host OS (Unix based) for I/O virtualization (for Type 2 Hypervisor - QEMU / KVM).

Overview:
Camera Virtualization - Access the camera on the host OS (device driver in host OS) by a guest VM by creating an abstract interface/device file on the guest VM that interacts with the device file on the host. The interaction between the guest VM and the host OS / hypervisor is achieved through hypercalls. A hypercall causes a transition from a guest OS to the hypervisor, similar to a system call that causes a transition from the user space to the kernel. The guest uses hypercalls to request privileged services (here, accessing device file) from the hypervisor.

Approach:
Create an abstract device file that replicates the original using hypercalls for communication.

- This approach is inspired from Paradice: A Paravirtualization approach for I/O virtualization. We aim to completely eliminate Common Virtual Driver (CVD) since the problem at hand is to solve Camera virtualization only.



Character Device file:
The abstract device file created is an actual character device file where the file operations are implemented / overwritten.

- Requires registering the device

(The interception of system calls on this file can be done using system call table and the callee) - We will explore if this device file created can actually be used to invoke the hypercalls required and completely eliminate the VM kernel module if unnecessary.

Scope:

Scope of the project is to demonstrate camera virtualization using device file virtualization. We plan to create a dummy driver in host OS and aim is to interact with the driver from the VM. This can be done by forwarding the file operations from guest VM to host OS. Plan is to implement the following file operations: open, read, write, ioctl, mmap and release.

Execution Steps:

1. Create an abstract device file (for camera) in guest OS.
2. For successful abstraction - Create a kernel module with the following functionalities:
    a. Intercept system calls on the device file created in Step 1.
        i. Make an alternate system call table that “points” to the alternative implementation. (Alternatively, Kprobe infrastructure / SystemTap provides similar functionality)
        ii. Verify the system call is on the created device file in Step 1 to choose alternate system call table.
        iii. Make a hypercall to QEMU/KVM hypervisor with the arguments specifying. (This might require a common page mapping such that guest VM and hypervisor can share the data): System call (intended) and corresponding arguments
3. For system call forwarding to the host OS - Create a kernel module with the following functionalities:
    a. The hypercalls intercepted by the hypervisor invokes this module functions.
    b. The functions in these modules map and invoke actual system calls intended by the vm.
        i. This requires mapping the common page, reading data from it and re-creating the arguments.
        ii. Corresponding system calls are invoked based on the data read with its arguments and the return values are handled (passing it back with the hypercalls / the mapped common page).

Timeline:

Week 7:

- Installed CentOS VM on Fedora Host OS.
- Setup Networking.
- Working on creating a kernel module (on VM) to create an alternate system call table.

Week 8:

- Create abstract device file
- Intercept system calls on device file and invoke hypercall
- Map common page to store system call and its arguments

Week 9:

- Create host side module to forward system calls
- Modify both kernel modules to accommodate for all system calls in the scope

Week 10:

- Verify return path and any other modifications.