VDUSE - "vDPA Device in Userspace"

vDPA (virtio data path acceleration) device is a device that uses a datapath which complies with the virtio specifications with vendor specific control path. vDPA devices can be both physically located on the hardware or emulated by software. VDUSE is a framework that makes it possible to implement software-emulated vDPA devices in userspace. And to make the device emulation more secure, the emulated vDPA device's control path is handled in the kernel and only the data path is implemented in the userspace.

Note that only virtio block device is supported by VDUSE framework now, which can reduce security risks when the userspace process that implements the data path is run by an unprivileged user. The support for other device types can be added after the security issue of corresponding device driver is clarified or fixed in the future.

Create/Destroy VDUSE devices

VDUSE devices are created as follows:

1. Create a new VDUSE instance with ioctl(VDUSE_CREATE_DEV) on /dev/vduse/control.
2. Setup each virtqueue with ioctl(VDUSE_VQ_SETUP) on /dev/vduse/$NAME.
3. Begin processing VDUSE messages from /dev/vduse/$NAME. The first messages will arrive while attaching the VDUSE instance to vDPA bus.
4. Send the VDPA_CMD_DEV_NEW netlink message to attach the VDUSE instance to vDPA bus.

VDUSE devices are destroyed as follows:

1. Send the VDPA_CMD_DEV_DEL netlink message to detach the VDUSE instance from vDPA bus.
2. Close the file descriptor referring to /dev/vduse/$NAME.
3. Destroy the VDUSE instance with ioctl(VDUSE_DESTROY_DEV) on /dev/vduse/control.

The netlink messages can be sent via vdpa tool in iproute2 or use the below sample codes:

```
static int netlink_add_vduse(const char *name, enum vdpa_command cmd)
{
        struct nl_sock *nlsock;
        struct nl_msg *msg;
        int famid;

        nlsock = nl_socket_alloc();
        if (!nlsock)
                return -ENOMEM;

        if (genl_connect(nlsock))
                goto free_sock;

        famid = genl_ctrl_resolve(nlsock, VDPA_GENL_NAME);
        if (famid < 0)
                goto close_sock;

        msg = nlmsg_alloc();
        if (!msg)
                goto close_sock;

        if (!genlmsg_put(msg, NL_AUTO_PORT, NL_AUTO_SEQ, famid, 0, 0, cmd, 0))
                goto nla_put_failure;

        NLA_PUT_STRING(msg, VDPA_ATTR_DEV_NAME, name);
        if (cmd == VDPA_CMD_DEV_NEW)
                NLA_PUT_STRING(msg, VDPA_ATTR_MGMTDEV_DEV_NAME, "vduse");

        if (nl_send_sync(nlsock, msg))
                goto close_sock;

        nl_close(nlsock);
        nl_socket_free(nlsock);

        return 0;
nla_put_failure:
        nlmsg_free(msg);
close_sock:
        nl_close(nlsock);
free_sock:
        nl_socket_free(nlsock);
        return -1;
}
```

How VDUSE works

As mentioned above, a VDUSE device is created by ioctl(VDUSE_CREATE_DEV) on /dev/vduse/control. With this ioctl, userspace can specify some basic configuration such as device name (uniquely identify a VDUSE device), virtio features, virtio configuration space, the number of virtqueues and so on for this emulated device. Then a char device interface (/dev/vduse/$NAME) is exported to userspace for device emulation. Userspace can use the VDUSE_VQ_SETUP ioctl on /dev/vduse/$NAME to add per-virtqueue configuration such as the max size of virtqueue to the device.

After the initialization, the VDUSE device can be attached to vDPA bus via the VDPA_CMD_DEV_NEW netlink message. Userspace needs to read()/write() on /dev/vduse/$NAME to receive/reply some control messages from/to VDUSE kernel module as follows:

```
static int vduse_message_handler(int dev_fd)
{
        int len;
        struct vduse_dev_request req;
        struct vduse_dev_response resp;

        len = read(dev_fd, &req, sizeof(req));
        if (len != sizeof(req))
                return -1;

        resp.request_id = req.request_id;

        switch (req.type) {

        /* handle different types of messages */

        }

        len = write(dev_fd, &resp, sizeof(resp));
        if (len != sizeof(resp))
                return -1;

        return 0;
}
```

There are now three types of messages introduced by VDUSE framework:

- VDUSE_GET_VQ_STATE: Get the state for virtqueue, userspace should return avail index for split virtqueue or the device/driver ring wrap counters and the avail and used index for packed virtqueue.
- VDUSE_SET_STATUS: Set the device status, userspace should follow the virtio spec: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html to process this message. For example, fail to set the FEATURES_OK device status bit if the device can not accept the negotiated virtio features get from the VDUSE_DEV_GET_FEATURES ioctl.
- VDUSE_UPDATE_IOTLB: Notify userspace to update the memory mapping for specified IOVA range, userspace should firstly remove the old mapping, then setup the new mapping via the VDUSE_IOTLB_GET_FD ioctl.

After DRIVER_OK status bit is set via the VDUSE_SET_STATUS message, userspace is able to start the dataplane processing as follows:

1. Get the specified virtqueue's information with the VDUSE_VQ_GET_INFO ioctl, including the size, the IOVAs of descriptor table, available ring and used ring, the state and the ready status.
2. Pass the above IOVAs to the VDUSE_IOTLB_GET_FD ioctl so that those IOVA regions can be mapped into userspace. Some sample codes is shown below:

```
static int perm_to_prot(uint8_t perm)
{
        int prot = 0;

        switch (perm) {
        case VDUSE_ACCESS_WO:
                prot |= PROT_WRITE;
                break;
        case VDUSE_ACCESS_RO:
                prot |= PROT_READ;
                break;
        case VDUSE_ACCESS_RW:
                prot |= PROT_READ | PROT_WRITE;
                break;
        }

        return prot;
}

static void *iova_to_va(int dev_fd, uint64_t iova, uint64_t *len)
{
        int fd;
        void *addr;
        size_t size;
```

```
        struct vduse_iotlb_entry entry;

        entry.start = iova;
        entry.last = iova;

        /*
         * Find the first IOVA region that overlaps with the specified
         * range [start, last] and return the corresponding file descriptor.
         */
        fd = ioctl(dev_fd, VDUSE_IOTLB_GET_FD, &entry);
        if (fd < 0)
                return NULL;

        size = entry.last - entry.start + 1;
        *len = entry.last - iova + 1;
        addr = mmap(0, size, perm_to_prot(entry.perm), MAP_SHARED,
                    fd, entry.offset);
        close(fd);
        if (addr == MAP_FAILED)
                return NULL;

        /*
         * Using some data structures such as linked list to store
         * the iotlb mapping. The munmap(2) should be called for the
         * cached mapping when the corresponding VDUSE_UPDATE_IOTLB
         * message is received or the device is reset.
         */

        return addr + iova - entry.start;
}
```

3. Setup the kick eventfd for the specified virtqueues with the VDUSE_VQ_SETUP_KICKFD ioctl. The kick eventfd is used by VDUSE kernel module to notify userspace to consume the available ring. This is optional since userspace can choose to poll the available ring instead.
4. Listen to the kick eventfd (optional) and consume the available ring. The buffer described by the descriptors in the descriptor table should be also mapped into userspace via the VDUSE_IOTLB_GET_FD ioctl before accessing.
5. Inject an interrupt for specific virtqueue with the VDUSE_INJECT_VQ_IRQ ioctl after the used ring is filled.

For more details on the uAPI, please see include/uapi/linux/vduse.h.