ECE 5984 Virtualization Technologies

Project 2:

Inter-VM communication channels

Team1
A K M Fazla Mehrab
Mincheol Sung

Introduction

Two virtual machines (VM), running on top of any hypervisor, may need to communicate with each other for several reasons. This communication can be done in several ways and there are few implementations also available. For example, the Inter-VM shared memory device (ivshmem) is designed to share a memory region between multiple QEMU processes running different guests and the host. kvmtool also have similar implementation which is not functional yet.

In this project we are assigned to implement three different Inter-VM communication mechanisms between two or mode VMs. First one is to communication between VMs using IN/OUT port instructions. Second one is communication using shared memory and the final one is using interrupts between VMs along with shared memory.

For the first one, in kvmtool, we use register and reserve I/O ports and copy data to local port where other kvmtool is notified by interrupt. For the second one, the shared memory one, we create, register and plug the shared memory to guest in kvmtool. For the final one, in addition to the steps for second one, we allocate and register the irq line in kvmtool.

On the other hand, at the guest side, a kernel module is written which gets the physical address of the shared memory and the irq line. As userspace application cannot use the physical memory a re-mapping is done which is accessed by the userspace application by mapping the memory. The kernel module registers an interrupt handler so that it can respond to interrupt in the irq line by send another interrupt to userspace application. Userspace application register a interrupt handler to respond to interrupt from kvmtool.

An interprocess communication (IPC) is developed so that each kvmtool can send a signal two each other when a data is available in the shared memory from the upper layer, i.e. VM. This IPC is done by sending signal between kvmtool instances where required PID of each of the instances are exchanged via shared memory.

We evaluate our implementation by sending various size of data between virtual machines. The results demonstrate our design is relatively fast and scalable.

Design

Our shared memory based inter virtual machine communication require modification on kvmtool, guest kernel and implementation of a user application. We need to modify kvmtool for allocating the shared memory. The kvmtool has a host virtual address of the shared memory and it should be converted to guest physical address which is still virtual address of the host. For accessing the guest physical address, the guest kernel should be modified and this can be achieved by a kernel module. The kernel module is charged to map the physical address of the shared memory to user accessible virtual address space.

In addition to shared memory, we also need a interrupt system to notify other kvmtool that data is ready in the shared memory. We could utilize port io, kvm interrupt system, and software signaling. Port io is a simple way to interrupt from the guest to kvmtool because in/out functions cause switch of non-root to root mode. CPU is in the root mode and executes io_in/out operation codes which sending signal to other kvmtool. The receiving kvmtool's signal handler then executes kvm__irq_trigger() to trigger irq to the guest. This triggers the receiving guest to handles the irq and switch of root to not-root mode of the CPU. Now, the receiving guest's kernel handles the up-coming irq and sends a signal to target guest user application to be ready to use data in the shared memory. This sequence is described in Figure 1.

As we implement transparently memory sharing, there is no data copy at all. It means that the sender write data into the shared memory, the data is transparently seen by the receiver. This zero copy can achieve significant performance benefit compare to any other inter process communication (IPC) using copying. Bottleneck in our design could be lying on the interruption and evaluation is required for a break down of the overhead of each software layer.

There are a lot of alternatives. For example, each virtual machine can communicate with messages through TCP/IP socket. Other can be that guest and kvmtool transfer data with port io only. Then sender kvmtool writes data on a file and receiver kvmtool reads data from the file. All of this alternatives are slower than our design due to many data copies. Compared to them, our design has zerocopy in data transfer. It is generally known fast.

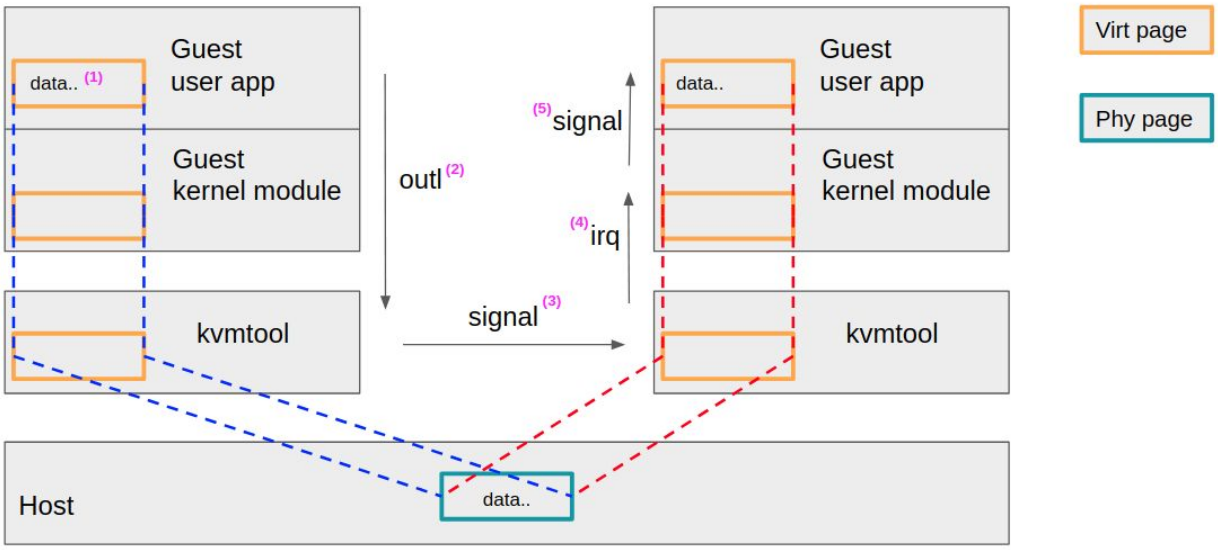

Figure 1. Shared memory and interruption

Implementation details

Host side (kvmtool)

Step 1 (IO port implementation)

1. we register io port and reserve port 0x700 ~ 0xb00 (range of 0x400) with **ioport__register()**.
    - Port 0x700 is used for sending a guest physical address of the shared memory.
    - Port 0x800 is used for sending an irq line.
2. we wrap struct ioport_operaions to our io_in/io_out implementation.
    - io_in does simply copy local data (guest physical address of the shared mem, irq line) to port's data.
    - io_out act as interrupt in our implementation. So io_out sends signal to other kvmtool when it is called.

Step 2 (Shared Memory)

1. Shared memory region is created by **setup_shmem_region()** in kvmtool. It fills the shmem region with a physical address, size, and handle. The handle is very import because shared memory can be referred by the handle. So each kvmtool should use same handle for memory sharing. In our implementation, we use default handle named "**/kvm_shmem**". After init the shmem region, open it and plug it into the guest by **setup_shmem()**.
2. Shared memory registration is done by **kvm__register_mem()**. It registers and allocates the shared memory with the guest physical address, size, and host virtual address of the shared memory.

Step 3 (Interrupt)

1. Irq line allocation: we can get unique irq line by calling **irq__alloc_line()**.
2. Irq registration: the irq line is registered with **kvm__irq_line()**.

GUEST side

Kernel module

1. Char device: we create a char device (**/dev/project2**) so that user application can communicate with it.
2. Port io: when the module loaded, it gets the physical address of the shared memory, and the irq line from port io. We use **inl()** for getting 32bit data.
3. Memory remap: The physical address of the shared memory is not allowed to be assessed by the user application. So we remap the physical memory into user vma (virtual memory area) that user can access. This remapping is done when the user application mmap(). We implement project2_mmap() for that.
4. Interrupt handling: we register an interrupt handler of the irq line we got through port io. We can use kernel function, **request_irq()**, to register **project2_interrupt_handler()**. **project2_interrupt_handler()** is simply sending signal (SIGUSR1) to the user

application with its PID and the PID is obtained when the user application opens the kernel module character device (/dev/project2). You can check **proejct2_open()**.

User application

1. Memory map is needed for user application to access the shared memory. We can simply map memory with **mmap()**.
2. Signal handler registration: When the kernel module gets interrupt from kvmtool, it sends a signal to the user application. So the user application needs to register signal handler. We use UNIX signal handler, **signal()**.

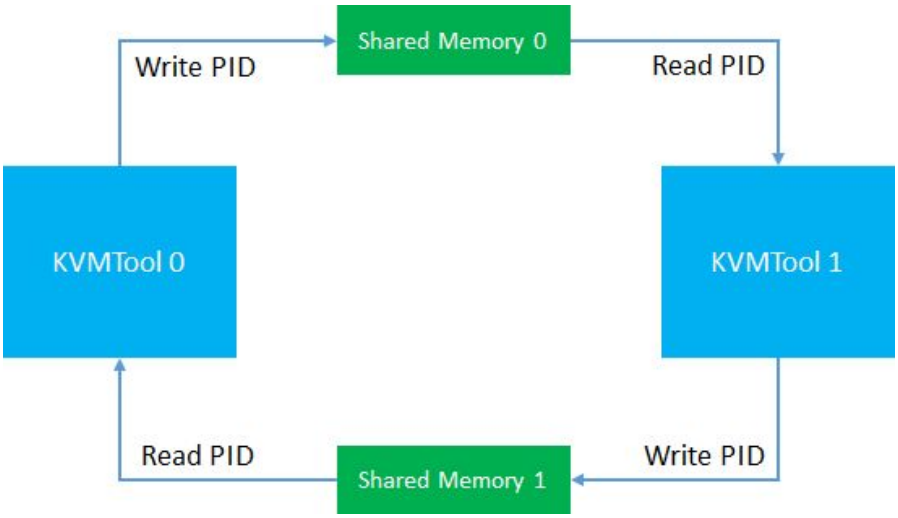

Figure 2. Inter-process communication between two kvmtool instances

IPC (kvmtool to kvmtool)

1. When we want two VMs to communicate with each other, we assign IDs for each of the corresponding kvmtools so that they can know who is the counterpart to communicate with. For example, as shown in the above figure, two kvmtool instances are assigned IDs 0 and 1. In order to facilitate the ID assignment, we added an extra argument (first argument), which takes an integer as the ID, in the existing kvmtool main function. So when we launch a kvmtool instance we provide with ID 0 or 1 as first argument.
2. According to our design, when a data is written in the shared memory, kvmtool is notified via interrupt so that it send a signal to the associated kvmtool of the destination VM. We send signal using a c library function called **kill()** which requires PID of the destination process. So we create two small shared memory, one for each kvmtool instance: say shared memory 0 and shared memory 1, for PID exchange between kvmtools.
3. We create to keys, one for each shared memory, using **ftok()** function. Then we create shared memory IDs from the corresponding keys using **shmget()** function. Finally with these generated IDs we allocate shared memories using **shmat()** function. As the the

pathname used to create the keys are same for kvmtool 0 and 1, they both both know which shared memory to uses for PID exchange.

4. When kvmtool 0 is launched, it writes its own PID in shared memory 0 and waits for kvmtool 1 to write it's PID on shared memory 1 and vice versa. Once PID are exchanged between kvmtool 0 and 1, they now know whom to send signal when an interrupt is triggered from guest VM.

Performance analysis

Experimental setup

Host machine

CPU: Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
Mem: 95549MB
OS: debian 9 (linux 4.9.0)

Guest

VCPU: 8
Mem: 3072MB
OS: ubuntu 16.04 (linux 4.9.0)

We launch two guests for sender and receiver each. We could allocate 128MB of shared memory. Actually we could allocate memory upto 256MB but the user program failed to mmap. So we limit the shared memory size to 128MB. The sender sends 4KB, 2MB, and 1GB of data to the receiver. For 4KB and 2MB, the sender just copy data to the shared memory. However, for 1GB the sender copy 128MB of chunk of data to the shared memory in an iteration and sends interrupt to the receiver. The receiver gets interrupt and store the transferred data from the shared memory to its local buffer and send ack to the sender. When the sender receive the ack, it can transfer another 128MB of data by copying data to the shared memory. After the receiver gets data, it writes the transferred data to a file on the guest's file system. The output files' name are output_4k, output_2m, and output_1g on a same directory with the receiver code. We can verify the data transferred correctly with these output files. The files contain 4KB, 2MB or 1GB of a character such as 'A' defaultly. You can choose the character to send by passing a parameter to the sender code.
The sender timestamps right before copy data from the local buffer to the shared memory and send interrupt. The receiver handles the interrupt and sends an ack. The sender receives the ack and timestamps. We can get latency by subtracting the first timestamp from the second timestamp.

Result

Figure 3 and table 1 are results of latency of our shared memory based communication channel. Thanks to 128MB of shared memory, transferring 4KB and 2MB of data is fast. Latency of 2MB of data is only double of 4KB even though 2MB is 512 times larger than 4KB. Both 4KB and 2MB requires

1. Data copy on the sender side ((1) in the Figure 1)
2. Interrupt from the sender to receiver ((2),(3),(4),(5) in the Figure 1)
3. Data copy on the receiver side (from the shared memory to the local buffer)

The difference between 4KB and 2MB is data size of copying (memcpy 4KB or 2MB)

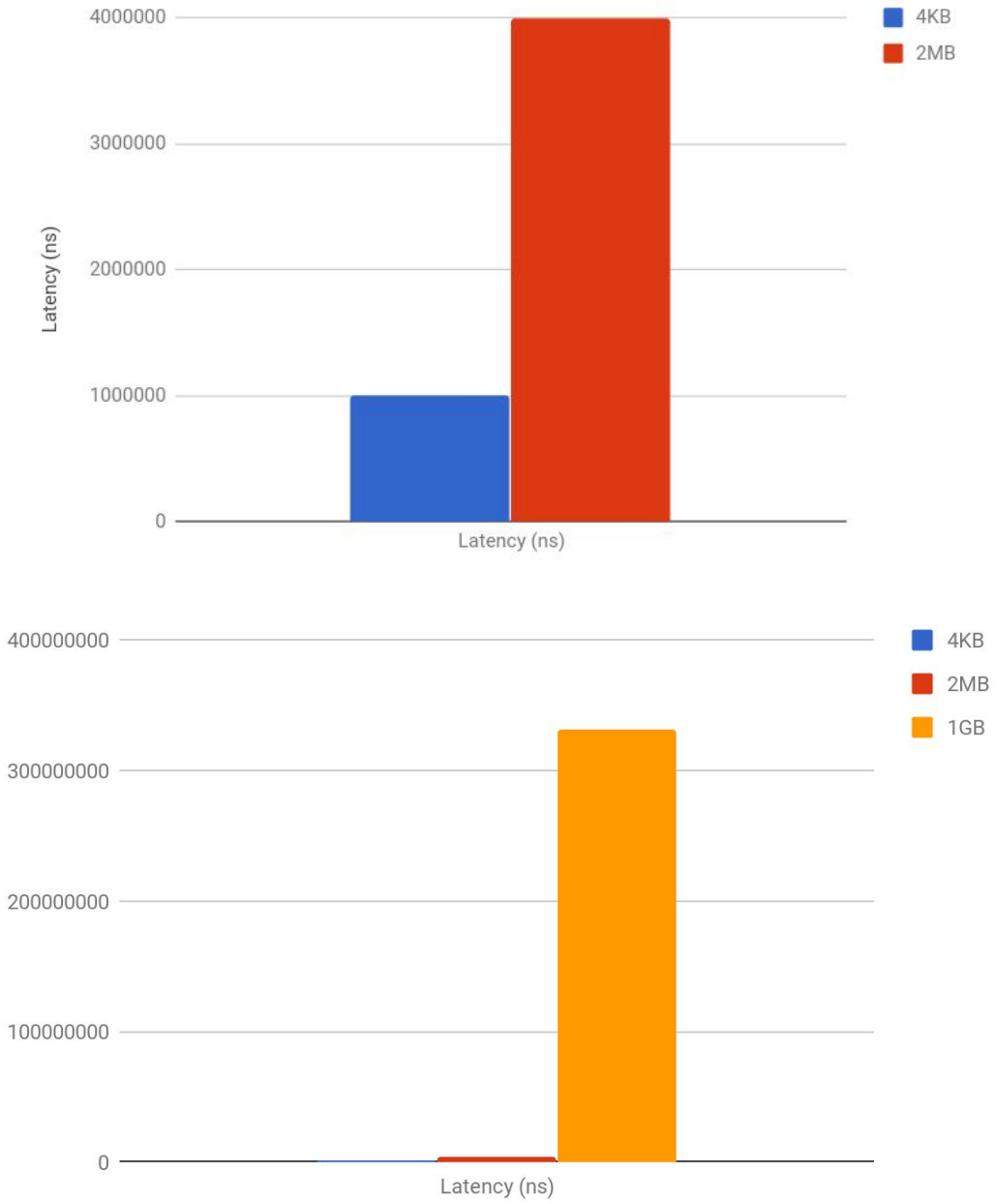

Figure 3. Latency of data transfer in ns

	4KB	2MB	1GB
Latency (ns)	999848	3999392	330949688

Table 1. Latency of data transfer in ns

Transfer 1GB of data is a different story. Since the shared memory is 128MB, we can't just do the same sequence as the case of 4KB and 2MB. Sending 1GB requires

1. Copy 128MB chunk of data to the shared memory
2. The sender sends interrupt to the receiver and waits for ack
3. The receiver gets the interrupt and copy the chunk of data to its local buffer
4. The receiver sends an ack to the sender
5. The sender receives the ack and copy the next 128MB chunk of data to the shared memory
6. Do this iterations until sending 1GB (1GB = 8 * 128MB, so 8 iteration)

So the latency of sending 1GB memory consists of 8 iterations:

1. Memcpy 128MB to the shared memory on the sender side
2. Send interrupt to the receiver
3. Memcpy 128MB to the local buffer on the receiver side
4. Send ack to the receiver

This takes ~0.33 seconds but still fast enough. The latency of 1GB is **only 82.75 times** larger than that of 2MB even though 1GB is **512 times** larger than 2MB. We conclude that this results look competitive.

Instructions for building and testing

Our deliverable consists of three parts:

1. Kvmtool (**kvmtool.patch**)
2. Kernel module on the guest (**modules/Makefile**, **modules/project2_modules.c**)
3. User application on the guest
    a. Sender side: **sender/Makefile**, **sender/project2_*.c**
    b. Receiver side: **receiver/Makefile**, **receiver/project2_*.c**
    c. Chat application: **sender/project2_chat.c**, **receiver/project2_chat.c**

KVMTOOL

Clone and checkout the kvmtool

```
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool.git
$ cd kvmtool
$ git checkout -b vt-class a508ea95f954d
```

Patch kvmtool

```
$ git apply kvmtool.patch
```

Build kvmtool

```
$ make
```

You can launch kvmtools with launch.sh and launch2.sh script on the kvmtool directory.
You should specify guest images and kernel images. In addition, note that you should set either 0 or 1 for sender and receiver guests. You can find difference between **launch.sh** and **launch2.sh** scripts.

KERNEL MODULE ON THE GUEST

Copy module codes to both sender and receiver guests.
To build the module

```
$ make
```

Then you can load the module

```
$ sudo insmod project2_module.ko
```

User application on the guest

Copy “sender” codes to the sender guest and copy “receiver” to the receiver guest.
To build sender or receive codes

```
$ make
```

You can find README to run applications and do the tests.

User application example (1. Chat application)

We include a simple chat application. You can run them by doing

```
$ sudo ./chat
```

on **both** sender and receiver guests.
Then it will prompt like below (blue box is a char client on a guest and redbox is an opposite client on other guest)

```
type “exit” for termination
>>
```

You can type message you want to send

```
type “exit” for termination
>>ping
>>
```

Then the other guest receives

```
type “exit” for termination
>>>>ping
>>
```

You can reply

```
type “exit” for termination
>>>>ping
>>Pong!!
>>
```

You can see the reply on the first guest

```
type “exit” for termination
>>ping
>>>>Pong!!
>>
```

To terminate the chat application, type “exit”

```
type “exit” for termination
>>ping
>>>>Pong!!
>>exit
$
```

User application example (2. Data sender and receiver)

To send 2MB data, note that you should run receiver first (purple box is a receiver, orange box is a sender)

```
$ sudo ./receive_2m
```

Then run sender

```
$ sudo ./send_2m
```

With a argument, you can send character you want

```
$ sudo ./send_2m C
```

Then, the receiver writes an output file

```
$ ls
output_2m              Makefile
project2_receiver_2m.c                  receive_2m
```

The “output_2m” is 2MB file with character “C”

```
$ du -sh output_2m
2.0M     output_2m

$ cat output_2m
CCCCCCCCCC … CCCCCCCCCCC
CCCCCCCCCC … CCCCCCCCCCC
CCCCCCCCCC … CCCCCCCCCCC
CCCCCCCCCC … CCCCCCCCCCC
CCCCCCCCCC … CCCCCCCCCCC
CCCCCCCCCC … CCCCCCCCCCC
```

Conclusion

We designed shared memory based inter virtual machine communication channel. By sharing memory between the virtual machines, we can implement fast and scalable communication channel. Our evaluation shows the latency of sending various size of data and demonstrates our design is relatively fast and scalable. This results could be done by minimizing data copy but still on the end sides (e.q., data copy from sender’s local buffer to the shared memory, and one from the shared memory to the receiver’s local buffer).