Shared memory based Inter-VM(application communication)

Abhishek Bagade 163059005
Vertika Srivastava 163050007

21st November 2016

Contents

Chapter 1

Problem Context

Virtual Machines (VM) needs to share a varying amount of data depending upon the nature of the workload, standalone servers require almost no data sharing but in case of transactional interaction between VMs the data to be shared increases manifold. For example consider the case of simulation server, we can spawn a VM for every simulation with different parameters supplied as input. Now these VMs run copies of the same codebase which may be in order of 100s of TeraBytes. Normally sharing data between VMs includes passing data via the network stack, this involve a lot of data copy at each network layer and processing delays.To avoid these overheads and reduce redundancy we use shared memory as a data sharing tool,this also provides zero-copy and low latency data sharing mechanism. Using shared memory requires many intricate details such as mapping the shared memory, avoiding overwrites etc., handling such details in every application involves unnecessary duplication of efforts and wastage of resources.Considering above issues, Use of shared memory communication between applications necessitates an easy to use Application programming interface (API).

Benefits of API:

1. High-level abstraction for Application developers.
2. Reduced overall application development time.
3. Easy to integrate.

Chapter 2

Problem Description/Goals

Creating an Application programming interface(API) would provide a convenient abstraction to application developer for sharing data between different virtual machines. API provides many advantages such as ease of use, hiding implementation details and easy deployment. Creation of API would efficiently accomplish our goal of sharing data between Applications in collocated VMs. The above end goal was broken up into following short-term goals:

1. Literature survey.
2. System setup and configuration.
3. Establish data sharing between host-to-guest and guest-to-guest using IVSHMEM mechanism.
4. Identify all the security and concurrency issues.
5. Develop an Application programming interface(API).
6. Basic testing.

Chapter 3

Approach

3.1 Literature Survey

Study and understand the working of IVSHMEM mechanism from various resources including forums,mailing lists,thesis and sample code. Filtering resources according to their applicability to our problem proved to be a challenging task as the documentation for IVSHMEM is scarce. The code samples encountered were rendered obsolete due to changes in linux kernel or were poorly maintained, which forced us to formulate a solution from scratch instead of using preexisting libraries such as DPDK(Data plane development kit).

1. **Phd thesis** This thesis is considered as the basis for the IVSHMEM linux kernel patch. It helped us gain insight on how the IVSHMEM mechanism works internally.
2. **DPDK IVSHMEM library** Understanding how DPDK implements IVSHMEM shared memory device. Code samples helped to understand coding techniques.
3. **Synerg homepage of Vaibhao Tatte** While mostly obsolete,helped us by directing to correct resources.
4. **Nairobi-embedded.org how to design a PCI driver** A good resource which helped us understand how the PCI-driver is implemented.

3.2 Basic setup

Study of qemu/kvm web resources to understand how to start a VM with shared memory and IVSHMEM device enabled. Running multiple VMs on a single system degraded the performance to unacceptable levels, finding a work-around included enabling the enable-kvm flag.

3.3 DPDK IVSHMEM library

DPDK was initially the library of choice for IVSHMEM implementation but running the code samples revealed that IVSHMEM standard is rendered obsolete and does not work in latest linux kernels, further correspondence with the developer community revealed that IVSHMEM is not included in the latest version of DPDK.

3.4 Implementation Of code directly from thesis

The code samples provided were easy to understand but were poorly maintained filing bugs in the github repository evoked a response from the author that the library is not maintained and the code samples were obsolete due to linux kernel changes.

3.5 Implementation from scratch using PCI device driver

Implementation from scratch proved to be hard initially but once the driver module was ready the actual code proved to be relatively straightforward.

3.6 Making an API for applications

Creating API involved learning how to create static library in C for our API. The functionality initially implemented in c file was ported to a static library.

Chapter 4

Design Details

4.1 System Architecture and Design

The system architecture had the following major considerations:

1. **Shared memory partitioning:** Dynamic partitioning was initially though of but the time spent on it was considerable, after deciding that the time required would be considerably higher we settled on static partitioning the memory into 2 equal parts. Partitioning ensures that no VM overwrites some other VMs data, at the same time a VM can read from any other VMs partition.

2. **How to identify which VM is running(VM_id):** We thought of initially getting the VM_id by running a server on a separate domain and the server separately issuing ID to every VM. The task though seemingly simple involved a lot of nuances which were hard to resolve. We eventually settled on Identifying the VM by its hostname which could be easily setup. The hostname would give us Vm_id by simple text processing.

3. **Handling multiple intra-VM writes** To ensure that no Application over writes data written previously. An offset is maintained to track the last written byte.

4.2 API design

The API design is as shown in figure.

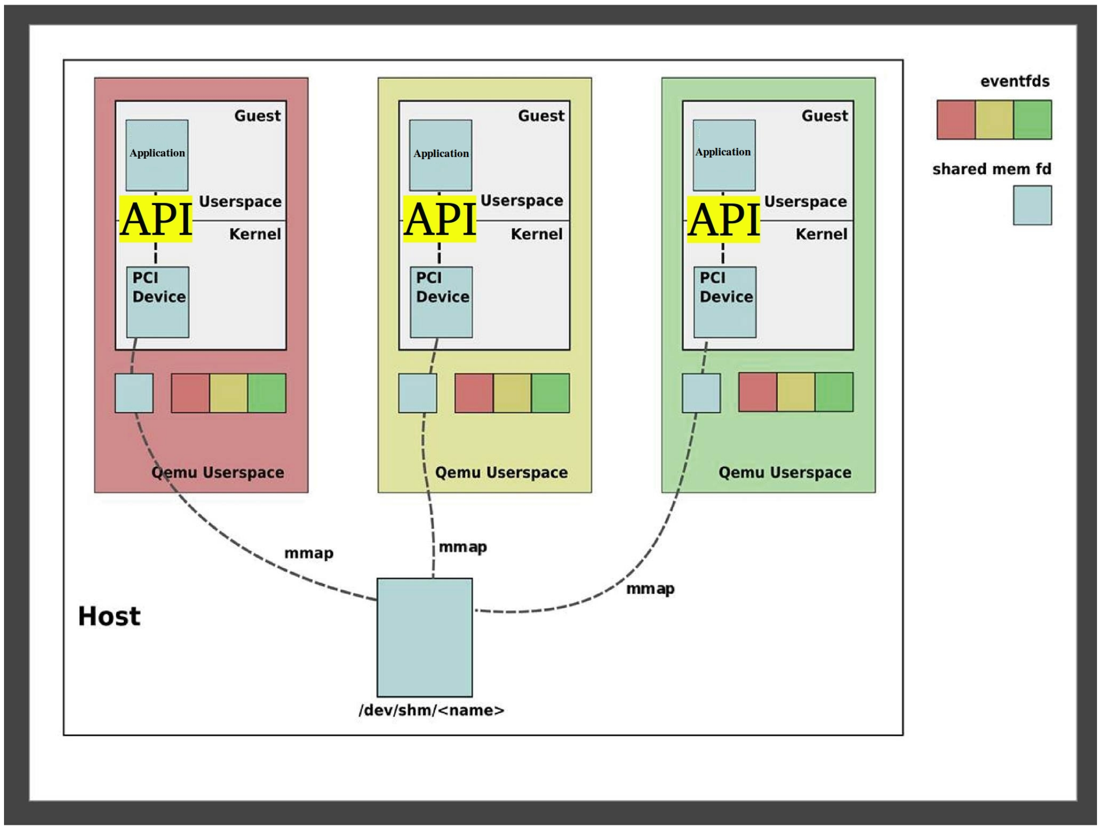

Figure 4.1: API Design

The API exposes following 4 functions to Application programmer:

1. **int init(struct shared_mem *data)** This function sets up the shared memory region. The data points to a structure containing the shared memory file name and its size. This function returns and integer greater than 0 for success and ¡ 0 for failure
2. **int readFromMap(char* recvBuff,int readVmId);** This function call reads the data stored in shared memory region pointed by the readvmid into the buffer provided by recvbuff.
3. **int writeFromMap(char *writeBuff);** This function call writes the data in writeBuff to corresponding VMs shared memory region.
4. **int shutdown()** This function unmaps all the data in shared memory region.

Chapter 5

Implementation

5.1 Virtual PCI Device Driver

Virtual PCI-driver involves setting various parameters, writing the driver from scratch proved to be very difficult especially understanding all the parameters and hex values. So we decided to try out a code from an online resource which itself was a well commentated version of the standard kvm-IVSHMEM.c standard device driver. The driver works perfectly in our test setup. Basically the PCI driver for IVSHMEM included setting the following values for the PCI parameters.

Configuration Space	Register Byte	Index (size) Value
PCI_VENDOR_ID	0x00 (16-bits)	0x1af4
PCI_DEVICE_ID	0x02 (16-bits)	0x1110
PCI_CLASS_DEVICE	0x0a (16-bits)	0x0500
PCI_BASE_ADDRESS_0	0x10 (32-bits)	0xfeb22000
PCI_BASE_ADDRESS_2	0x18 (32-bits)	0xfea00000
PCI_SUBSYSTEM_VENDOR_ID	0x2c (16-bits)	0x1af4
PCI_SUBSYSTEM_ID	0x2e (16-bits)	0x1100
PCI_INTERRPT_LINE	0x3c (8-bits)	0x0b
PCI_INTERRUPT_PIN	0x3d (8-bits)	0x01

Table 5.1: PCI device IVSHMEM-specific flags

5.2 Files

1. **api.h:** It is the header file and contains function declarations. It also includes the structure that will be used by applications to pass configuration values to API.This file has to be included by the developer in his

application to access the API functions.

2. **api.c:** It contains the definitions of the functions provided in the API and other global variables.
3. **api.a:** This is the static library build up to provide the API.Developer has to include this will compiling his c application.

5.3 Functions

1. **int init(struct shared_mem *data)**

   Return Value :

   return > 0 : on success

   return < 0 : on error

   Parameters:

   struct shared_mem data {

   char *filename;

   int filesize;

   }

   This structure will be use to pass configuration parameters like filename of the shared memory region and filesize for the size of shared memory region.

   It is the initialization function that needs to be called first to initialize configuration parameters for the API.This function sets up the shared memory region and maps in user space. It takes vm id from the hostname of the vm.This hostname is got from gethostname().Vm hostname is set for the vm in the format vm¡id¿.This id part is extracted and used throughout as vm id.

2. **int readFromMap(char* recvBuff,int readVmId);**

   Return Value :

   return > 0 : on success

   return < 0 : on error

   Parameters:

   char* recvBuff : Application will pass pointer to a buffer.Data read from shared memory region will be placed in this buffer.

   int readVmId : It specifies the id of the VM from whose partition data is to be read.

   It reads the data from the partition of VM specified in the parameters.The shared memory region has already been mapped in user space.

3. **int writeFromMap(char *writeBuff);**

   Return Value :

   return > 0 : on success

   return < 0 : on error

   Parameters:

   char* writeBuff : Application will pass pointer to a buffer.Data from this buffer will be written in the shared memory region of the VM.

   This function writes the data provided in writeBuff in the corresponding VMs shared memory region.Application cannot write in the partition of some other VM.It maintains an offset to keep track of last byte written.A new write called will write from offset in the partition.Every write call with offset as 0 resets the memory of its partition.

4. **int shutdown()**

   Return Value :

   return > 0 : on success

   return < 0 : on error

   This function unmaps all the data in shared memory and closes file descriptor that was used to point to this shared memory region.

Chapter 6

Experimentation

To test the API we created a test application. The application takes input from the user and performs appropriate operations. There are 3 operations in the menu

1. **Read:**
   The user is prompted to input the id of VM to be read from the program returns the string present in the supplied VMs shared memory section.

2. **Write:**
   User is prompted to enter a string, this string is then written to memory section of the VM running the application.

3. **Exit:**
   Exits program.

Chapter 7

Inferences

1. Shared memory can be effectively utilized and implemented using IVSHMEM, but using shared memory involves handling various challenges such as race condition,concurrency control etc. these problems have to be handled by the API.

2. The solution provided does mitigate some of the problems,but introducing Dynamism in the solution would enhance the scalability of the API.

3. The API designed manages to hide implementation specific details effectively, providing a high-level abstraction with well defined functions.

4. Though IVSHMEM effectively solved the data sharing problem its use is discouraged on system development forums.This suggests that there are issues in IVSHMEM which restricts its appeal as a robust solution.

Chapter 8

References

1. **Shared-Memory Optimizations for Virtual Machines**
   A. Cameron Macdonell
   https://era.library.ualberta.ca/files/h702q7744/thesis.pdf

2. **DPDK: IVSHMEM library**
   http://dpdk.org/doc/guides-16.04/prog$_{g}uide/IVSHMEM_{l}ib.html$

3. **Writing a PCI device driver for QEMU/KVM**
   http://nairobi-embedded.org/linux$_{p}ci_{d}evice_{d}river.html$

4. **IVSHMEM info. by SYnerg,IITB**
   http://www.cse.iitb.ac.in/synerg/doku.php?id=public:students:vaibhao:IVSHMEM

5. **Code samples for IVSHMEM server**
   https://github.com/henning-schild/IVSHMEM-guest-code/tree/master/IVSHMEM-server