ECE 5984 Virtualization Technologies

Project 2: Inter-VM communication channels

Group Members: Anway Mukherjee, Divya Ramanujachari

1. Introduction

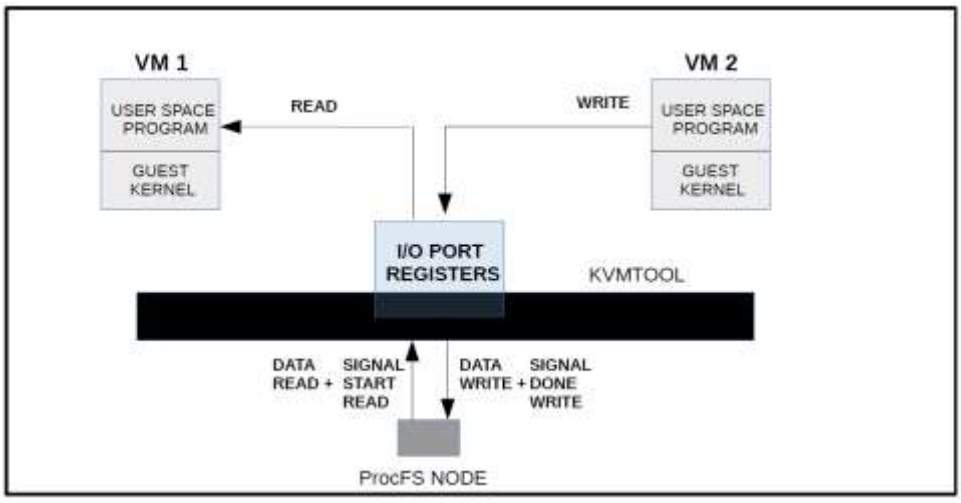
In this project, three different inter-VM communication mechanisms are implemented and their performances are compared. Inter-VM communication specifically indicates a scenario in which multiple virtual machines are co-located on a single host. In this case, traditional TCP/IP communication using sockets would involve significant overhead that could be avoided by taking advantage of the fact that the VMs are hosted on a single machine. The communication mechanisms implemented in this project are based on (i) use of IN/OUT port instructions, (ii) shared memory without interruptions, and (iii) shared memory with interruptions. The mechanisms were evaluated by comparing the communication latency for different size of inputs and by varying the packet sizes. The results showed that port communication latency increased with increase in packet size while the shared memory mechanisms performed best in the case of large packet size.

2. Using IO Ports for Inter-VM Communication

Design

I/O ports refer to specific addresses on the x86's IO bus that provide communication with devices (https://wiki.osdev.org/I/O_Ports). For inter-VM communication purposes, each VM is treated as a device connected to the host. To set up communication, the kvmtool is modified to catch IN/OUT instructions on some specific ports. These ports are used by userspace programs running in the guest VM to establish a communication channel. A procfs node in the host OS is used as the communication link between host kernel and user space (http://tjworld.net/books/ldd3/#UsingTheProcFilesystem). Furthermore, use of this setup enables implementation of synchronization between the read and write operations by use of a mutex lock.

The sequence of events that occur when VM1 wishes to communicate with VM2 are detailed below.



- A program is executed in the user space of VM2 that writes to a virtual I/O port.
- The VMM intercepts the communication request and forwards it to the kvmtool as it cannot handle I/O natively.
- The kvmtool writes the received data to a procfs node created in the host OS driver.
- Simultaneously, a program running in the user space in VM1 attempts to read from a virtual I/O port. The VMM intercepts this request and forwards it the kvmtool, which reads and forwards the data from the procfs node created by the same host OS driver.

Implementation

In this section, the exact additions/modifications in the programs are described.

Userspace program:

port_forwarding.c

This program takes in user input that specifies if the operation to be performed involves receiving or sending information. The port number to be used for communication is hardcoded to be 0x0092, a PS2 legacy I/O Port specific to x86. The ioperm function is executed to allow the program to access ports. In case the user desires to perform read, the value received in port 0x0092 is printed. Otherwise, the information to be written is output using the function outb, which takes in the value and port number as parameters and outputs a byte of data.

Inter-VM communication enabler:

bridge_comm.c

This program creates a procfs node in the host OS that is used for information transfer between the host kernel and user space. The **proc_create** function is used to create a proc entry “bridge_comm_VM” under /proc. Its access mode is set to 0777, and **proc_fops** is passed as the structure in which the read and write

file operations for the proc entry are mapped. Specifically, this structure defines the mapping “read: read_proc_r” and “write: write_proc_w.” A mutex lock is used to implement synchronization, i.e., ensure that no conflict arises from the port being read from and written to at the same time. The following resource was used as reference for the setup and functioning of proc entries for read/write operations: http://tuxthink.blogspot.in/2013/10/creating-read-write-proc-entry-in.html.

kvmtool change:

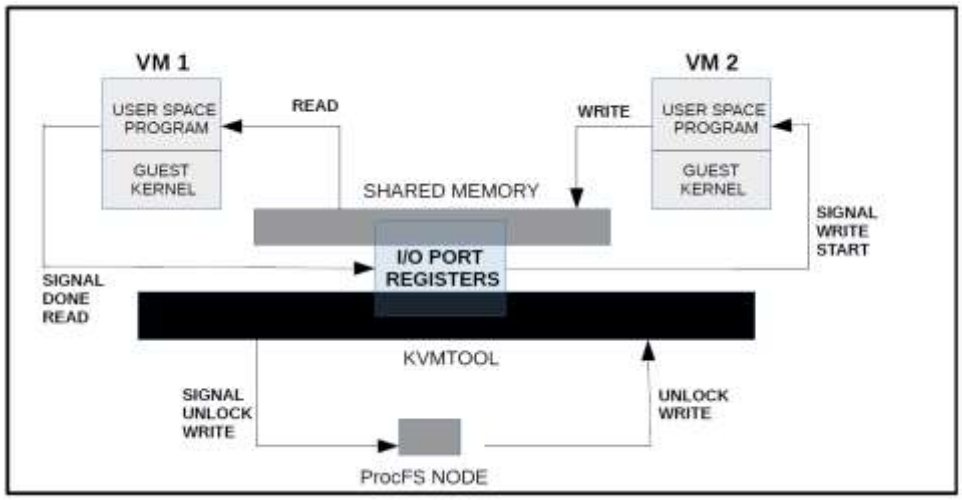
ioport.c

The following functions were added/modified in the ioport program present in kvmtool. In the **ps2_control_a_io_in** function, the bridge_comm_VM proc is triggered in the read mode. The obtained information is written to the port using ioport__write8. Similarly, in **ps2_control_a_io_out**, information is read from the IO port using ioport__read8. The appropriate function signatures are mapped for input and output operations in the structure **ioport_operations ps2_control_a_ops**.

3. Shared Memory without Interrupt

Design

In this approach, a memory region is designated in the host for enabling inter-VM communication. In terms of implementation, the shared memory is a file whose node address and properties are known to kvmtool; this file is accessed when read/write operations are triggered. The shared memory is exposed to the guest kernel as a PCI Base Address Register (BAR). mmap is used in the userspace application to create a mapping in the virtual address space of the calling process, thereby allowing the programs to simply read/write to a virtual memory area. This design is inspired by the approach used in ivshmem in qemu which is shown below. Note that this approach does not require a device driver. For notification and synchronization of the read/write access to the shared memory, the port implementation from the previous part is reused.



The sequence of events that occur when VM1 wishes to communicate with VM2 are detailed below.

- A program is executed in the user space of VM2 to write to the shared memory. This involves searching for available PCI devices, retrieving their information, loading the device region, and performing the write operation.
- The kvmtool is responsible for setting up the shared memory region and exposing the underlying physical file as a PCI device. It accesses the procfs on the host OS to synchronize read/write operations. On successful write, it notifies the host OS driver by writing to a designated port.
- The host uses a procfs to maintain a mutex lock to enforce synchronization.
- Simultaneously, a program running in the user space in VM1 attempts to execute the read operation. The kvmtool uses the physical file details to obtain the data and mmap it to the virtual memory from which data can be accessed directly. Its operation is identical to that of the program in VM2 except that read is performed instead of write.

Implementation

In this section, the exact additions/modifications in the programs are described. The resource http://nairobi-embedded.org/mmap_n_devmem.html was used as a reference.

Userspace guest programs:

shmem_wo_intr.c

In this program, the memory-mapped I/O (MMIO) region of a PCI BAR is accessed via "/dev/mem," which memory maps a region of physical memory to the calling process' address space. The various functions that support this operation are defined in **shmem_wo_intr_wrapper.c**, namely, upci_scan_bus, upci_find_device, and upci_open_region. Details about these functions are described next. Once these are executed, a new virtual memory area is allocated within the virtual address space of the program instance that is associated with the MMIO region of the BAR. The read (**do_pci_read**) and write (**do_pci_write**) operations take as input the data region (BAR descriptor) and an offset (number of bytes) into the mapping. Essentially, the shared memory is abstracted as the virtual memory.

shmem_wo_intr_wrapper.h, shmem_wo_intr_wrapper.c

In the **upci_scan_bus** function, the file "/proc/bus/pci/devices" is read, and for each device in the file, the device's standard PCI configuration space information is added to an internal table. This includes information on the address and size of regions associated with available BAR for each device. Then, in **upci_find_device**, parameters such as DeviceID and VendorID are used to retrieve the device number. Using this device number and input BAR index, the BAR information is retrieved from the table in the **upci_open_region** function. The details obtained include the base address and size. Along with a file descriptor obtained using open on /dev/mem, the base address and size values are used as the offset and length parameters of the mmap interface, respectively.

Inter-VM communication enabler:

bridge_comm.c

The purpose of this program has already been described earlier. To support inter-VM communication using shared memory without interrupt, the program is extended by adding the functions write_proc_w_shmem and read_proc_r_shmem along with mapping structure file_operations proc_fops_shmem. In this instance, the proc created is called “shmem_comm_VM.” A mutex lock is used as before to provide synchronization.

kvmtool change:

ioport.c

Two new functions are added to this program. In the function **dummy_ioport_ops_io_in**, the shared memory is accessed in the read mode. fgetc is used to read the shared resource and once this is completed successfully, this status information is written to ioport__write8 to indicate that shared resource is now accessible for read. In **dummy_ioport_ops_io_out**, ioport__read8 is used to check if the shared resource is accessible, then this is written to the shared memory via /proc/shmem_comm_VM. The structure **dummy_ioport_ops** provides the appropriate mapping for ioport_operations. Further, the operation handler 0092 - PS/2 system control port A is modified to handle the port-mapped I/O implementation.

pci-shmem.h and pci-shmem.c

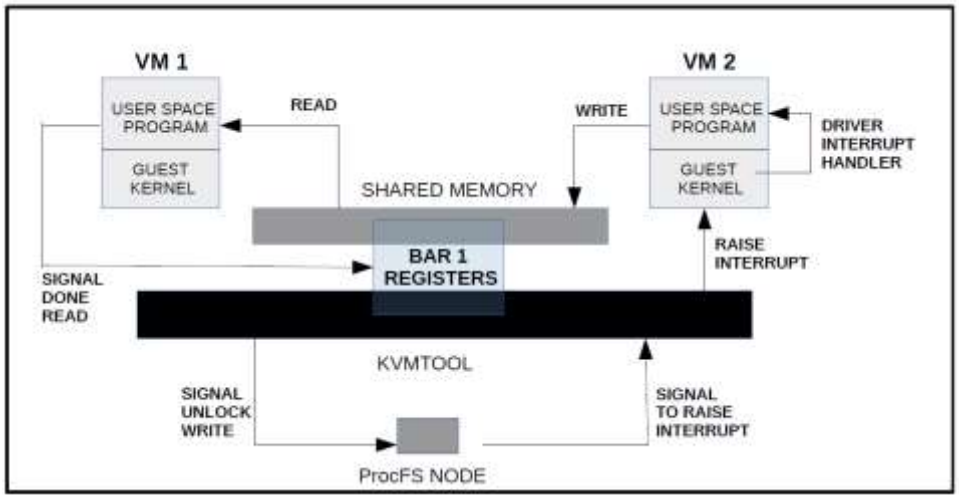
pci-shmem is an existing program in kvmtool that helps set up shared memory resources and expose it to the guest. In this instance, it is used to initialize shared space by taking as input parameters such as the base address, size, and mode. In the header file, the SHMEM_DEFAULT_HANDLE is set as “kvm_shmem.” In the c file, the various supporting functions are defined. The **setup_shmem** function opens the shared memory file and performs mmap. **shmem_parser** extracts the shared memory physical address, size, handle, and passes on the ownership. The **pci_shmem__init** function is modified to register the shared memory for inter-VM communication as a BAR. This calls the setup function to initialize the shared memory and plug it into the guest. Memcpy is implemented in the function **callback_mmio_msix** to read or write to the memory as appropriate.

4. Shared Memory with Interrupt

Design

As before, our design is inspired by the approach followed in ivshmem implemented in qemu in which a memory region is designated in the host for enabling inter-VM communication. The framework uses the eventfd mechanism, an event wait/notify mechanism that notifies userspace application events to the kernel and vice-versa, to send interrupt requests to the VM. While this design is an extension of the previous approach in that a shared memory region is used for inter-VM communication, in this case, ports are no longer used to implement synchronization. Instead, shared PCI registers are used to signal the completion of successful read/write operation. Further, a device driver is implemented for the ivshmem PCI device.

The sequence of events that occur when VM1 wishes to communicate with VM2 are detailed below.



- A program is executed in the user space of VM2 with the objective of writing to a shared memory region. This involves searching for available PCI devices, retrieving their information, loading the device region, and performing the write operation. The PCI register u8val is used to signal successful completion of an operation.
- The kvmtool is responsible for setting up the shared memory region and exposing the underlying physical file as a PCI device. It accesses the procfs on the host OS to synchronize read/write operations. On successful write, it notifies the host OS driver by raising an interrupt using the eventfd mechanism.
- The device driver which exposes the shared file as a sysfs node to the user is waiting for an interrupt. Once this is obtained, it unlocks the shared resource for reading by the other VM.
- Simultaneously, a program running in the user space in VM1 attempts to read from the virtual memory region. Its operation is identical to that of the program in VM2 except that read is performed instead of write.

Implementation

In this section, the exact additions/modifications in the programs are described. The resource http://nairobi-embedded.org/linux_pci_device_driver.html was used a reference.

Userspace guest programs:

uspace_guest.c

The userspace program includes function to read from/write to the MMIO region associated with the shared memory BAR of the ivshmem device. The PCI devices are found by calling upci_scan_bus, the required device details are retrieved using upci_find_device, upci_open_region is called to initialize data region for that device. File name, size, and mode of operation are specified. The value of dev.u8val is set and used for signaling. Once read operation is executed, it is set to a value of 0 which indicates “send back

signal notifying that read is complete, not unblock write." Before performing write, it is set to a value of 1 which indicates "send back signal indicating request to write."

shmem_wo_intr_wrapper.h, shmem_wo_intr_wrapper.c

These programs are retained from the previous implementation and have been described in detail earlier.

Guest Device Driver:

uio_ivshmem.c

The device driver searches for the PCI device, registers the device along with its functionality, and sets up the IRQ. The device is defined as a structure **ivshmem_dev** that has a pointer to register base address along with start and end values, a pointer to shared memory base address along with start and end values, and the interrupt value. In the function **ivshmem_probe**, the registers and shared memory associated with the device are mapped to the kernel space, and further processing is suspended till an interrupt occurs. The user space program reads on the sysfs node created by the driver. This acts as a semaphore that is only unlocked when the device issues an interrupt.

kvmtool change:

pci-shmem.c

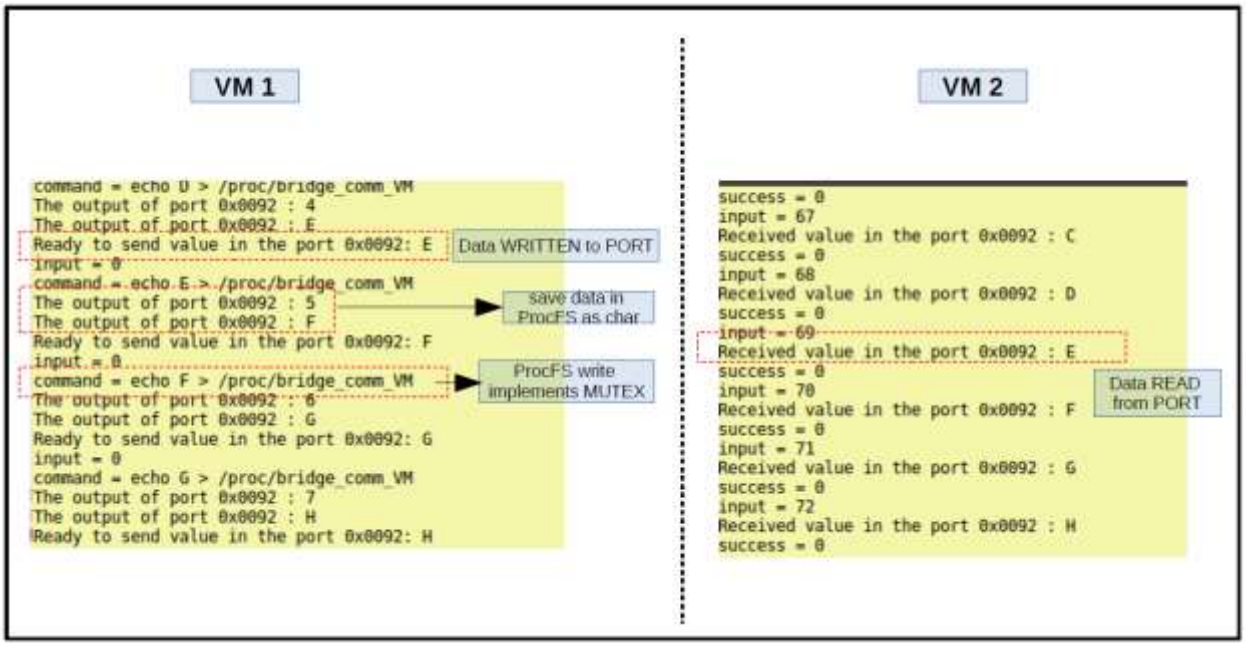
As described earlier, the pci-shmem program is provided in kvmtool to set up shared memory resources and plug into the guest. The following changes were made to this program in addition to the ones described in the previous section. The **pci_shmem__init** function is modified to set local_VM_fd = pci_shmem__get_local_irqfd(kvm). The **callback_mmio_msix** function originally performs the memcpy appropriately depending on whether the operation is a read or write. To support use of shared memory with interrupt as a VM communication channel, the interrupt value is taken into consideration.

The code for the implementation can be obtained from https://github.com/anwaymukherjee/inter_VM_communication.

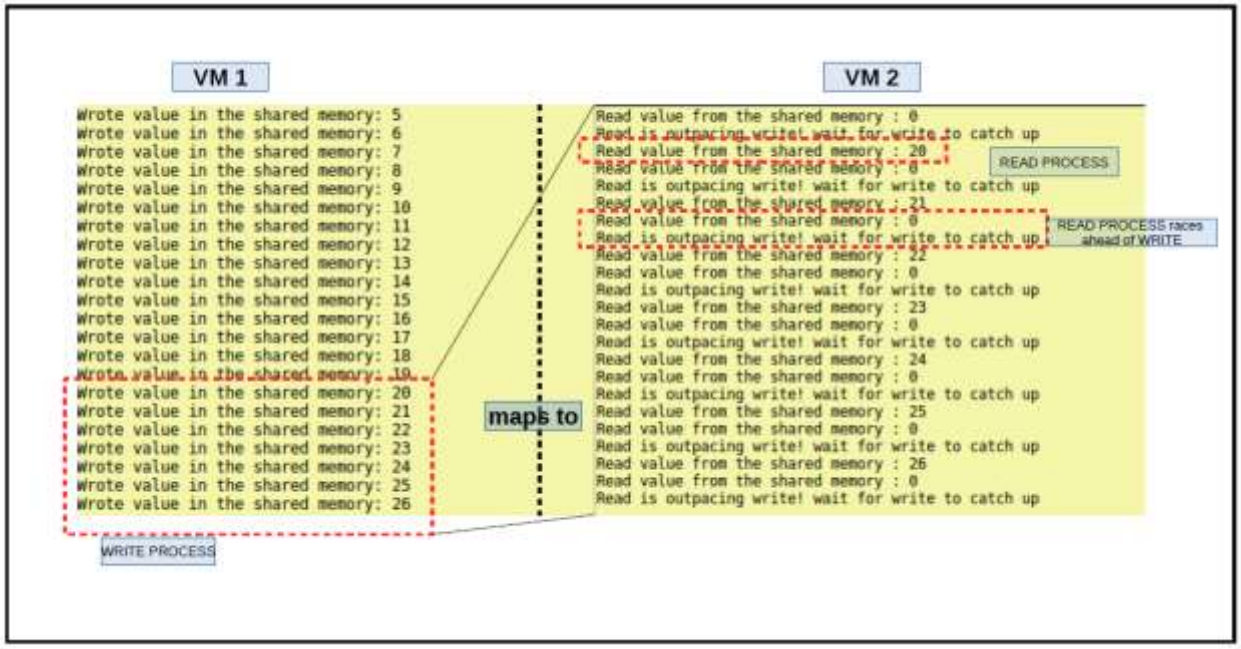
5. Performance Evaluation

The three different inter-VM communication modes were executed for different input and packet sizes. The below screenshots show the flow of execution for each of the mechanisms.

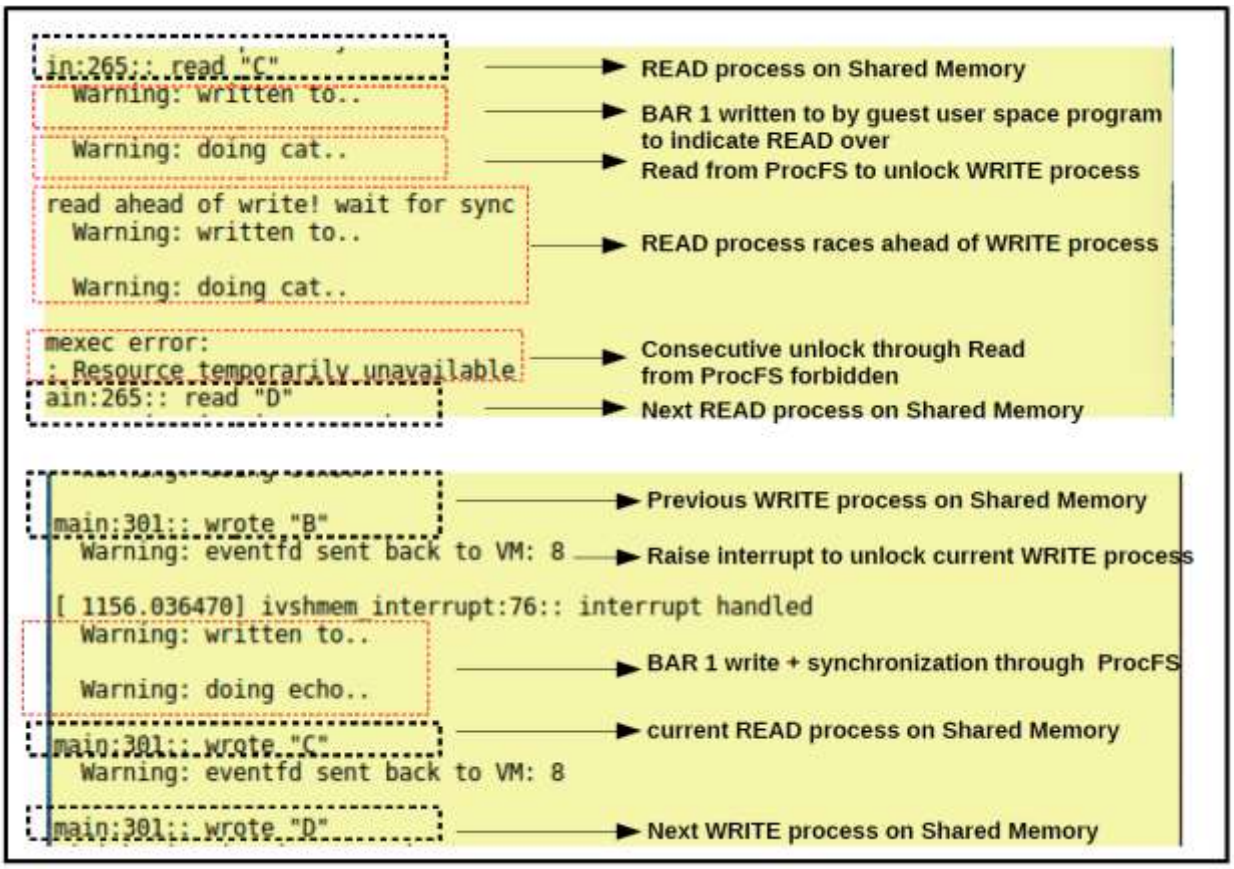
Port I/O



Shared memory without interrupt



Shared memory with interrupt



Layer specific overhead for each of the steps:

Read shared mem buffer : 2 ms

Write shared mem buffer : 3 ms

Interrupt handler : 0.01 ms

Synchronization

Procfs read : 5 ms

Procfs write : 5 ms

Sleep time :

read : 4 sec

write : 5 sec

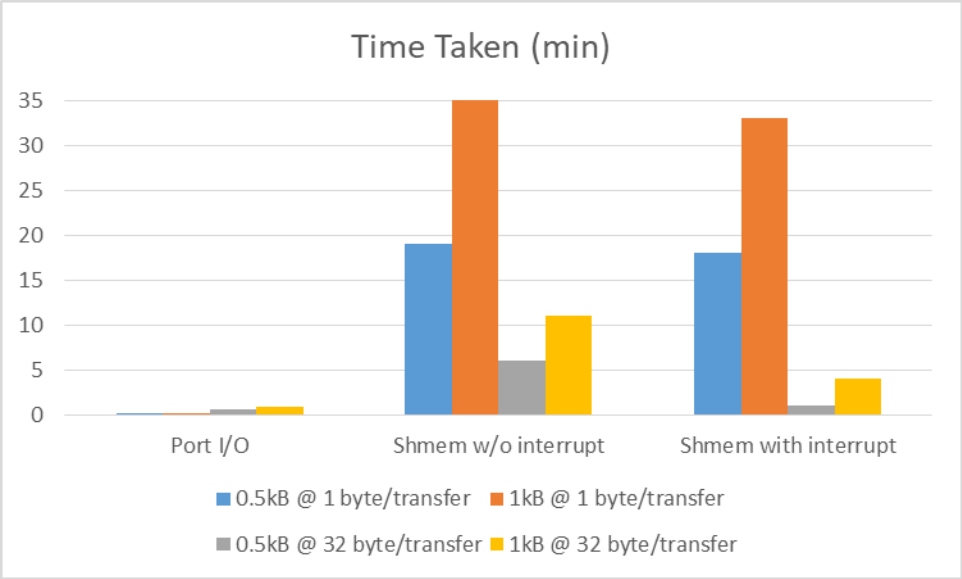
Reset shared mem buffer: 3 ms

From analyzing the above values, it is evident that the maximum overhead is introduced owing to the synchronization step.

The time taken for communication was recorded for different inputs as a performance metric for comparison. The table below shows the time taken in minutes for all three mechanisms for input sizes of 0.5 kB and 1 kB and packet rates of 1 byte/transfer and 32 bytes/transfer.

Mechanism	**Port Input/Output**		**Shared memory w/o interrupt**		**Shared memory with interrupt**	
Input Size (kB)	0.5	1	0.5	1	0.5	1
Packet Size: 1 byte/transfer						
Time Taken (min)	0.05	0.2	19	35	18	33
Packet Size: 32 byte/transfer						
Time Taken (min)	0.6	0.8	6	11	1	4

The following graph provides a comparative analysis of how the communication latency varies.



In the case of port I/O, the latency increases linearly both with input size and packet size. In the case of the shared memory mechanisms, the latency is high when the packet size is small. This can be attributed to the overhead imposed by synchronization and memory copy operations. With increase in packet size, the performance of shared memory improves as the packet size compensates for the overhead of data synchronization and memcopy. Within the large packet size, the latency increases linearly with size of input as expected. Further, increasing the input size to values such as 2 MB and 1 GB exponentially increases the experiment run time for very small data packets.

6. Conclusion

In this project, three different inter-VM communication mechanisms were implemented and their performances were compared in terms of the communication latency. The results showed that port input/output mechanism was most efficient for small packet sizes. Communication latency decreased with increase in packet size in the case of the shared memory mechanisms as the large size of each communication compensated for the overhead. Analysis of the overhead due to each step showed that synchronization was the most expensive.