International Conference on Control, Automation and Systems 2007
Oct. 17-20, 2007 in COEX, Seoul, Korea

Sharing Data Between Processes Running on Different Domains in Para-Virtualized Xen

Yun Chan Cho[1] and Jae Wook Jeon[2]

[1] Department of Mobile Communications, SungKyunKwan University, Suwon, Korea
(Tel : +82-31-290-7937; E-mail: yccho@ece.skku.ac.kr)
[2] School of Information and Communication Engineering, SungKyunKwan University, Suwon, Korea
(Tel : +82-31-290-7129; E-mail: jwjeon@yurim.skku.ac.kr)

Abstract: Recent embedded systems such as mobile phones provide various multimedia applications and execute them concurrently. Embedded systems are being developed to provide new unique solutions and services. This forces manufacturers of embedded systems to select a more productive and sophisticated operating system such as Linux. However, there are several limitations in using a general purpose operating system (GPOS) because of software instability and workload of transporting legacy real-time applications into the system. A good solution is to use a virtualization technology such as Xen, which allows many guest operating systems to be executed simultaneously and in a stable manner on one physical machine. If Xen is applied to embedded systems such as mobile phones, the Xen can execute the legacy real-time operating system for critical tasks and also GPOS executing user-friendly task. In this case, the Xen should provide the IPC mechanism between processes on each operating system on Xen. However, Xen doesn't provide a simple way for sharing data between processes running on different guest operating systems. In this paper, we propose a simple method for sharing data between the processes in different guest operating systems by using a mechanism provided by Xen.

Keywords: Xen, para virtualization, IPC, shared memory, mobile phone

1. INTRODUCTION

Embedded systems are being used in more application domains than standard personal computers. End-users are enjoying services such as voice call service, streaming audio/video service and image photography using mobile phones, PDAs and so on. These embedded systems continue to be requested to provide advanced applications to end-users. To meet these end-users demands, embedded systems need to use a powerful General Purpose Operating System (GPOS), which already has a number of developed applications and supports an open development environment. Typical GPOSs include Linux, Windows OS, Mac OS X, and Unix. But these GPOSs have several challenges such as real time response, security and stability of system. So it is difficult to use a GPOS alone in some embedded system due to these challenges.

A new approach for solving these challenges is virtualization technology. Virtualization is a technique for hiding the physical characteristics of hardware resources from other guest operating system running itself [1]. So the Xen, which is one of virtualization technology, can execute the legacy real-time operating system for critical tasks and also GPOS executing user-friendly task. The approach solves the challenges of GPOSs above.

Xen is a para-virtualization and virtual machine monitor (VMM) for x86. Xen supports execution of several guest operating systems called as domain, with unprecedented levels of performance and resource isolation [1]. Despite of strict isolation between domains, Xen provides a way to communicate between domains. It is a split device driver mechanism. This mechanism is used in Block I/O or network I/O [2]. Because Xen doesn't permit unprivileged domains to access devices such as hard disk or network card directly, unprivileged domains should request the privileged domain to access these physical devices through a split device driver. It is very difficult for domains on Xen to share some data between processes on different domains without using specific devices, because of the high complexity of the split device driver mechanism. In this paper, we propose a new mechanism for sharing data between the processes in different domains. The following paragraph outlines the organization of this paper.

In the chapter 2, we present the basic mechanisms related with sharing data on Xen. This helps to understand the proposed mechanism for sharing data in this paper. We provide the design and implementation of the proposed mechanism in chapter 3. And we conclude this paper in chapter 4.

2. BASIS FOR SHARING DATA ON XEN

Now Xen should use the split driver mechanism to permit that domains share the data generated in block I/O or network I/O. The split driver mechanism means that when a domain needs to do I/O operations in a block or network device, the domain does I/O operations indirectly through the privileged domain. Sharing the data in the split driver mechanism is implemented as reading or writing the data on pages that are shared between the privileged domain and unprivileged domain. The split driver mechanism consists of several other mechanisms such as Xenbus and Xenstore. The new mechanism which is introduced later in this paper uses the split driver, first will present the mechanism to implement data sharing between domains.

978-89-950038-6-2-98560/07/$15

2.1 Grant table

Xen uses a grant table to share the pages between guest OSes internally. All domains have a grant table, which is referenced when Xen shares pages between domains. The Fig. 1 shows the data structure of the grant table in the Linux kernel which is used as a domain in this paper.

```
struct grant_entry{
    uint16_t flags;
    domid_t domid;
    uint32_t frame;
}
```

Fig. 1 Data structure of grant table.

domid as a member variable of grant_entry is a domain identifier that indicates the domain permitted to access the page matched by a frame as a member variable of grant_entry. The frame member is the physical address of the page shared with other domain, which identifier is domid member of the grant_entry. The flag as a member in grant_entry is a variable that includes the value of the various options of shared page such as read or write permissions. This grant table structure is used to setup shared pages between domains when domains use the spilt device driver for block or network I/O.

2.2 Xenstore and Xenbus

Xenstore is information storage space that is shared with all domains on Xen [3]. Xenstore maintains information such as configuration or state, not scalable data. All domains have a unique path in Xenstore. They store information made up of a pair consisting of key and value. Xenstore is accessed by functions defined in xs.h [3]. We read or write keys and values in Xenstore using management tools such as xm.

A feature of Xenstore is that the watch [3] is set on a key. This means that whenever a value of the key that has a watch is changed, the already registered callback function is executed. Xenstore is used in the split device driver to provide information between domains.

Xenbus offers bus abstraction for virtual driver such as split device driver. Usually Xenbus plays the role of interface in reading or writing information such as configuration or state in Xenstore [3].

2.3 Split Drivers

In this section we discuss the split device driver, which is an important component to implement the mechanism related with sharing data on Xen.

The split device driver mechanism consists of two parts. The first part is the backend driver that accesses directly the physical devices such as hard disk and network card [4]. And the second part is the frontend driver which requests access to the backend driver for the physical devices. It functions similar to a proxy. From now on we call the privileged domain dom0. And we call the unprivileged domain domU [9].

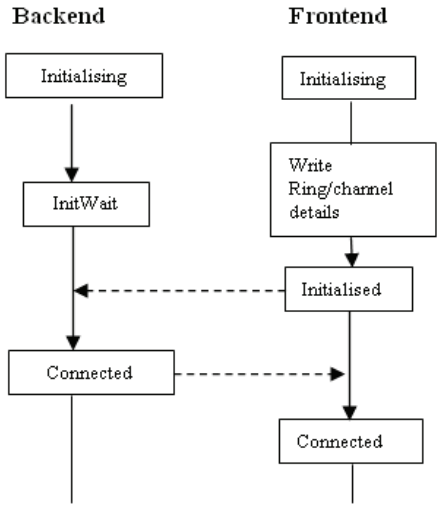

Fig. 2 Connection process of Backend and Frontend

Drivers related to block or network I/O are implemented on Xen using a split device driver mechanism. The Fig. 2 presents the process where a frontend driver for block I/O on domU connects a backend driver for block I/O on dom0 after they are created.

After the backend driver is created initially in dom0, the backend driver should be registered in Xenbus. And the backend driver exists in the “Initializing” state until the value of frontend key and frontend-id key are written in the path of dom0 on Xenstore. After the frontend driver is created intially in domU, the frontend driver should also be registered in Xenbus. And similar to the backend driver, the frontend driver exists in the “Initializing” state until the value of the backend key and the fronted-id key are written in the path of domU on Xenstore [3].

When information of Fig. 3 is written in Xenstore, the backend driver and the frontend driver can proceed with the remaining step of the connection process [3].

```
/local/domain/0/backend/vbd/U/<deviceID>/...
frontend /local/domain/U/device/vbd/<deviceID>
frontend-id U state XenbusStateInitialising ...

/local/domain/U/device/vbd/<deviceID>/...
backend /local/domain/0/backend/vbd/U/<deviceID>
backend-id 0 state XenbusStateInitialising ...
```

Fig. 3 Information written on Xenstore for connection

After the backend driver on dom0 reads the value of the frontend driver and that of frontend-id in the path of dom0 on Xenstore, it exists in the “InitWait” state. The next backend driver registers the watch on the state key of the frontend driver in Xenstore as requesting the operation to Xen. And the backend driver observes

whether the frontend driver state in Xenstore is changed. The frontend driver on domU also reads the value of the backend driver and one of backend-id in the path of dom0 on Xenstore. And the frontend driver allocates a free page for the Ring I/O and updates the grant table with the domid value of dom0 and the physical address of the shared page. And then the frontend driver allocates a new event channel , which is used for event communication with the backend driver. Lastly, the frontend driver writes two pieces of information in Xenstore. First piece of information is the reference index in the grant table of the frontend driver. It is also called ring-ref and indicates a page allocated for Ring-I/O. The second piece of information is the value of the event channel allocated for event communication with the backend driver on dom0. After writing these two pieces information in Xenstore, the state of the frontend driver in domU switches to the "Initialized" state. Lastly, the frontend driver observes whether the backend driver state in Xenstore is changed.

When the state of the frontend driver in Xenstore is changed to the "Initialized" state, the backend driver in dom0 reads the ring-ref and the value of the event channel in Xenstore and saves them in local variables. Backend driver in dom0 requests Xen that backend can share a page indicated by ring-ref with domU. If Xen permits the backend driver to share a page with the frontend driver, the backend driver requests Xen for the backend driver in dom0 to connect the event channel with the frontend driver in domU, by passing the value of the event channel of the frontend driver. And the backend driver switches to the "Connected" state. Because the state of backend driver in Xenstore changes, the frontend driver in domU wakes up to call the backend_driver_changed function. After processing several steps, the frontend driver switches the state of domU to the "Connected" state.

From now on both the backend driver and the frontend driver share data through a predefined algorithm, which uses Ring I/O. This split device driver makes other shared pages, using the method that Xen uses when making shared pages for Ring I/O.

3. IMPLEMENTATION FOR SHARING DATA BETWEEN USER PROCESSES ON DEFFERENT DOMAINS

Now Xen permits domains to share data through a split device driver mechanism that is used in block and network I/O. This mechanism is a unique interface for sharing data between domains. However when domains would like to share the data not related with block I/O or network I/O, it is difficult for a domain to easily share these data with other domains using this interface. So we implement a simple interface that can support data sharing between user processes on different domains as using basically split driver mechanism.

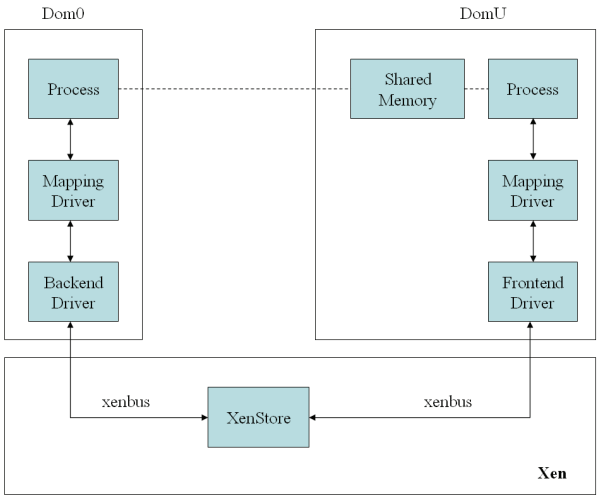

Fig. 3 Design for sharing data in process level

The Fig. 3 shows a block diagram of this paper's mechanism that implements the processes in different domains on Xen to share data easily. Fisrt Dom0 registers the backend driver using an interface provided by Xen for sharing domU's memory. And dom0 create a dummy device file and defines a read/write operation with the address of shared memory, then registers a character device driver to the dom0 kernel (we will call this driver the mmaping driver) so that the process of dom0 can open the created device file and access the shared memory through read/write requests to the mmaping driver.

domU also registers the frontend driver using an interface provided by Xen, for sharing the memory of dom0. And as dom0 did, domU creates a dummy device file and defines read/write operations with the address of shared memory, then registers a character device driver to the domU kernel as the mapping driver. Using the mapping driver, the domU process can open the dummy device file and access the shared memory through the mapping driver. The backend driver of dom0 and the frontend driver of domU basically use the mechanism of the split device driver.

We used one page for processes on different domains to share data. The page is the Ring I/O page, which is used by the split device driver. A process of dom0 can open the dummy device file with which dom0 registered as a mapping driver, reads some data from the dummy device file or writes some data in the dummy device file. This step is the same as that of domU. If a write operation of a process on dom0 is finished, a process of domU can read data that dom0 already wrote to a shared page. The offset that the two process use in the shared page is shared through Xenstore.

3.1 Creation of backend driver for sharing data

To share pages with other domains in Xen, the first dom0 registers itself as the backend driver in Xenbus. The Fig. 4 is the general way that Xen registers the backend driver.

```
Static struct xenbus_driver shmback = {
    .name    = "shm"
    .owner   = THIS_MODULE,
    .ids     = shmback_ids,
    .probe   = shmback_probe,
    .remove  = shmback_remove,
    .otherend_changed = frontend_changed
};

void shmif_xenbus_init(void)
{
    Xenbus_register_backend(&shmback);
}
```

Fig. 4 Registration of backend driver

Xen can register the backend driver by calling the xenbus_register_backend function. This function registers the linux driver, which uses the bus, by calling the driver_register function in the linux kernel. If the backend driver is successfully registered in Xen, the xenbus_dev_probe function that the backend driver indicated as the probe function is called. And the backend driver on dom0 waits until Xen writes the values of frontend and frontend-id in Xenstore. These values are generally written in Xenstore through the xm. As soon as these two values are written in Xenstore, the backend driver on dom0 calls the watch_otherhand function. This allows Xen to monitor the value of the frontend driver state in Xenstore. If the frontend driver on domU is registered in Xenbus, the frontend driver state is changed to "initialized". Because the frontend driver state is changed, Xen recognizes this and calls the otherend_changed function which the backend driver defined in initialization. The frontend_changed function reads values of ring-ref and event-channel of the frontend driver from Xenstore. The value of ring-ref is used as the identifier of the pages when two domains share pages using the split device driver mechanism. We didn't use value of ring-ref because we use a page allocated for Ring I/O descriptors as a page that two domains share for data transmission.

The value of the event-channel is used to generate events for other domains. A domain informs other domains to finish an operation such as read or write through this event.

Next the frontend_changed function allocates a free page in dom0's kernel. After that, it calls the hypervisor [3] function in Fig. 5 with the value of ring-ref so that the backend driver on dom0 can share a page with domU.

```
gnttab_map_grant op = {
    .host_addr = shmif -> blk_ring_area->addr,
    .flag = GNTMAP_host_map,
    .ref = shared_page,    /* value of ring-ref */
    .dom = shmif->domid,
}

HYPERVISOR_grant_table_op(GNTTABOP_
map_grant_ref, &op, 1);
```

Fig. 5 Hypervisor function for sharing a page

Lastly the frontend_changed function on the backend driver calls another hypervisor function of Fig. 6 to connect the event channel with the frontend driver on domU. And the backend driver registers the handler executed by calling the bind_evtchn_to_irqhandler function when events that have a value of evtchn are generated.

```
struct evtchn_bind_interdomain bind_interdomain;

Bind_interdomain.remote_dom = shmif->domid;
Bind_interdomain.remote_port = evtchn;

HYPERVISOR_event_channel_op(EVTCHNOP_
bind_interdomain, &bind_interdomain);
```

Fig. 6 Hypervisor function for connecting the event

When event is occurred, this event handler makes that values of shared page' offset related with read/write on Xenstore can be changed properly. Later we will talk about this offset in detail. Lastly, the frontend_changed function sets the state of the backend driver to "Connected" in Xenstore. The backend driver waits until an event occurs in the connected event channel.

3.2 Creation of frontend driver for sharing data

To share some pages with other domains in Xen, the first domU registers itself as frontend driver in Xenbus. The method that registers the frontend driver in Xenbus is similar to that of the backend driver.

Xen can register the frontend driver by calling the xenbus_register_frontend function in Fig. 7. As it did in initialization of the backend driver, this function registers the linux driver, which uses the bus, through calling the driver_register function in the linux kernel. If the frontend driver is registered in Xen successfully, the xenbus_dev_probe function that the frontend driver indicates as a probe function is called. And the frontend driver on domU waits until Xen writes the values of the backend driver and backend-id in Xenstore.

```
static struct xenbus_driver shmfront = {
    .name    = "shm"
    .owner   = THIS_MODULE,
    .ids     = shmfront_ids,
    .probe   = shmback_probe,
    .remove  = shmback_remove,
    .otherend_changed = backend_changed
};

void shmif_xenbus_init(void)
{
    Xenbus_register_backend(&shmback);
}
```

Fig. 7 Registration of frontend driver

As soon as these two values are written in Xenstore, the frontend driver on domU calls the its probe function which is member of shmfront struct. This function allocates a page for saving Ring I/O descriptors. It calls the setup_blkring function and the setup_blkring function internally calls the gnttab_grant_foreign_access function. The gnttab_grant_foreign_access function set s values in grant table, which is named as grant_entry, with domid and frame. This table is referenced when Xen verifies that the selected page can be shared between two domains. If the gnttab_garnt_foreign_access function is performed successfully, it returns the value of ring-ref. And the setup_blkring function continues to allocate an event channel for connection with the backend driver on dom0, using the xenbus_alloc_evtchn function.

Because the backend driver on dom0 connects itself in this event channel, hypervisor function of Fig. 8 in the frontend driver should be executed in advance.

```
Struct evtchn_alloc_unbound alloc_unbound;

Alloc_unbound.dom = DOMID_SELF;
Alloc_unbound.remote_dom = dev->otherend_id;

HYPERVISOR_event_channel_op(EVTCHNOP_
alloc_unbound, &alloc_unbound);
```

Fig. 8 Hypervisor function for allocating an event

The frontend driver registers the handler executed when events that have a value of evtchn are generated, by defining the bind_evtchn_to_irqhandler function. And the frontend driver writes values of ring-ref and evtchn in Xenstore through the xenbus_printf function because these values are used in the backend driver for shared memory and event operations. And then the frontend driver changes the state of itself to "Initialized" in Xenstore. The frontend driver on domU calls the watch_otherhand function. This makes Xen monitor the value of the backend driver state in Xenstore. After the backend driver on dom0 reads the values of ring-ref and evtchn which the frontend driver wrote in Xenstore, the frontend driver on domU changes the state of itself to "Connected" in Xenstore. And the frontend driver waits until the backend driver state in Xenstore is changed to "Connected" state. If the backend drivers state in Xenstore is changed to "Connected" state, the frontend driver on domU calls the backend_changed function. This function sets the frontend driver state to be "Connected" in Xenstore and makes the frontend driver wait until events occur in the connected event channel.

3.3 Mapping driver as intercessor between a split driver and a process

Up to now, we have set the shared page between domains at the split driver level. From now on, we introduce how the user process on the domain accesses the shared page in kernel on each domain.

Linux permits user processes to access the data in kernel mode through linux device drivers. We enable a user process to access shared page, which is shared in two domains and is located in kernel on each domain. First we created a dummy device which is used in the registration of the device driver. We defined that the read and write functions of this driver access a page shared between two domains. And we made the linux kernel in dom0 and domU register a character device driver with the dummy device. After that, user process can open this device file and read data from the device file, write data to the device file.

The data structure having several functions such as open, read, and write should be set before the linux in each domain registers character device driver with the dummy device. When the user process in user mode calls these functions, these functions access the shared page in kernel mode The two file offsets in read and write functions are important information. Whenever these offsets are changed, they are written in Xenstore. The following shows the point of time at which the register_chrdev function should be called in the backend driver.

```
static void frontend_changed(struct xenbus_device
*dev, enum xenbus_state frontend_state)
{
 …
 register_chrdev(SHM_DEV_MAJOR,
SHM_DEV_NAME, &shm_fops);
 …
}
```

Fig. 9 Registration of mapping driver in backend driver

The register_chrdev function, which registers a character device in the Linux kernel, is called by the shmback_probe function. The shmback_probe function in the backend driver is called when the frontend driver state in Xenstore becomes "Initialized". When the frontend driver finishes the basic step for sharing pages such as allocating a shared page and event channel, the frontend driver state of domU in Xenstore is changed to

"Initialized". After identifying the frontend driver state as "Initialized", the backend driver in dom0 calls the frontend_changed function. And the frontend_changed function calls the register_chrdev function. As the backend driver calls the register_chrdev function, and the character device driver is registered in the linux kernel. This paper names this driver to mapping driver. The user process can access the shared page via this mapping driver.

The role of the mapping driver in the frontend driver is similar to the one of the backend driver. But the point in time at which the register_chrdev function is called is different to the backend driver. When the values of the backend driver and backed-id are written in Xenstore, the shmfront_probe function is called and the next register_chrdev function of Fig. 10 is called in the shmfront_probe function.

```
static int shmback_probe(struct xenbus_device
*dev, const struct xenbus_device_id *id)
{
  …
  register_chrdev(SHM_DEV_MAJOR,
SHM_DEV_NAME, &shm_fops);
}
```

Fig. 10 Registration of mapping driver

Irrespective of dom0 or domU, the process in user mode can access the shared page between dom0 and domU, performing read/write operations on the registered dummy device.

When processes share some data in the linux kernel through the pipe or message queue, the linux kernel guarantees that they use the same read/write offset. we made a simple method to support this. This method is that both the backend driver and the frontend driver use the same offset of read/write operation as saving the offset in Xenstore. After the read or write operation is finished in a domain, the domain necessarily saves the read/write offset in Xenstore. And if a domain starts read or write operations in a shared page, the domain should read the offset from Xenstore preferentially and replace the local offset with the new offset of Xenstore. The method to use these mapping driver is same in dom0 and domU. But because this mechanism is very primitive way, some mechanisms should be added to be used in applications which use shared memory in two domains.

3.4 Access of sharing data in process level

The top level in which we are about to access the shared page between dom0 and domU ultimately is the process level in user mode. A process of each domain can access the shared page through the calling functions of the mapping driver that is already registered in the kernel. If a process calls the open function with the device file as a parameter, the corresponding open function of mapping driver is called to make the process ready to access the shared page. After that, if a process calls read or write functions, the corresponding read or write functions of the mapping driver are called and the shared page is accessed properly. Lastly, the mapping driver requests Xen to set the read/write offset of itself in Xenstore. If Xen processes the request of the mapping driver on the domain successfully, operation on the shared page is finished. The process on the other domain can also perform the same operations. With this, we can implement sharing data in top level.

4. CONCLUSION

Xen can load many other operation systems, called domains, and have guaranteed protection between domains. But legitimately sharing data across domains is very difficult. When block I/O or network I/O occur in Xen, a split device driver mechanism is used for sharing of I/O data. But this mechanism also is very complex to use in general data sharing not in block I/O and network I/O. This paper introduced a way to share data easily between user processes on two domains. First we simplify the complexity of the split driver mechanism through only using a page allocated for Ring-I/O descriptor. And we added the mapping driver as interface for permitting access of the user process on the shared page. The process in user mode reads data from the shared page or writes data to the shared page through calling the functions of the mapping driver. Finally we design and implement a simple mechanism of sharing data at the process level. The mechanism introduced in this paper can be a basis for high-performing data communication between processes on different virtualized domains.

REFERENCES

[1] Tim Abels, Puneet Dhawan, Balasubramanian Chandrasekaran, "An Overview of Xen Virtualization."

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, Andrew Warfield, "Xen and the Art of Virtualization", In Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003

[3] http://wiki.xensource.com/xenwiki ; Xen Wiki.

[4] Jiuxing Liu, Wei Huang, Bulent Abali, Dhabaleswar K.Panda, "High Performance VMM-Bypass I/O in Virtual Machines", 2006