**Community Enterprise**

**Operating System**

**BAYOT, JULY D.**

**ITP51**

**OPERATING SYSTEM**

**TABLE OF CONTENTS**

**Chapter 1**

Introduction 4

History 4

**Chapter 2**

Work with process 6

Basic CentOS Process and Job Management in CentOS 6

**Chapter 3**

Linux scheduling 11

**Chapter 4**

Memory Management 13

Prerequisites on the host 14

Restrictions for guests 14

Collaborative memory management ­15

Ballooning 15

Memory Tuning 16

**Chapter 5**

I/O Threads 17

Logical Volume Management 17

Path Redundancy / Data Integrity 17

**Chapter 6**

I/O Systems 20

**Chapter 7**

File Systems 22

File System Theory 22

Indexing Methods / Inodes 23

FAT 24

Workings of File Systems 24

Allocation Table 25

Separate file and system areas 25

Network File Systems 25

File systems for OSDevers 25

"Beginners" filesystems 26

**Chapter 1**

**Introduction and History**

COMMUNITY ENTERPRISE OPERATING SYSTEM

**Introduction**

Community Enterprise Operating system (CentOS) offers an open-source, enterprise-class free operating system that is practicably compatible with Red Hat Enterprise Linux (RHEL). “Gregory Kurtzer” is the founder of CentOS. CentOS developers use the RHEL source code to generate a product that is highly comparable to RHEL.

CentOS provides a development platform in one of the best and most powerful available distributions. It is a community-driven free software project built to provide a robust platform for open-source communities to grow. It is highly adaptable, as well as safe and strong. In addition, it features several corporate-level security updates that declare it an excellent choice for any use.

**History**

CentOS was released in May 2004 as an operating system that is completely free and based on the Linux kernel. CentOS is originated from RHEL. Its goal is to deliver an enterprise-class computing platform that is freely available and maintain Red Hat’s binary compatibility. CentOS was introduced as a CAOS build and was founded by “Gregory Kurtzer.”

After that, “David Parsley,” Tao Linux primary developer, announced in June 2006 that Tao Linux would be retired and its development would be absorbed into CentOS (Tao Linux is another RHEL clone). By using the yum update, Tao users were able to upgrade their current system version to CentOS. Unfortunately, “Lance Davi,” the founder of CentOS, was reported missing on the CentOS project website in July 2009. “Davis” had stopped contributing to the CentOS project, but he had kept his CentOS website.

The CentOS team allegedly contacted “Davis” in August 2009 and got the domains of centos.org and centos.info. CentOS became the most popular Linux distribution in July 2010 and overtook Debian’s popularity for web servers, accounting for over 30% of all Linux web servers. However, in January 2012, it was dropped to second place by Debian. Red Hat declares in January 2014 that the team will sponsor the CentOS project, assisting in the development of a platform that is compatible according to the requirements of the open-source developers that work around the operating system and emerging technologies.

CentOS trademarks have been shifted to Red Hat. RHEL standards and open-source team group, which work independently from the RHEL team, employs the CentOS lead developers. In the hosting market, CentOS is considered the most reliable distribution. CentOS is exceptionally compatible with most Linux software because of its binary compatibility with RHEL. CentOS is also the most suitable Linux distro for most hosting control panels.

**Chapter 2**

**Process Management**

**Work with Processes**

Quick Note − Process PID in Linux

In Linux every running process is given a PID or Process ID Number. This PID is how CentOS identifies a particular process. As we have discussed, systemd is the first process started and given a PID of 1 in CentOS.

Pgrep is used to get Linux PID for a given process name.

As seen, the pgrep command returns the current PID of systemd.

**Basic CentOS Process and Job Management in CentOS**

When working with processes in Linux it is important to know how basic foregrounding and backgrounding processes is performed at the command line.

* fg − Bringsthe process to the foreground
* bg − Movesthe process to the background
* jobs − List of the current processes attached to the shell.
* ctrl+z − Control + z key combination to sleep the current process.
* & − Startsthe process in the background

Let's start using the shell command sleep. *sleep* will simply do as it is named, sleep for a defined period of time − sleep.

If you are following along, you'll notice the foreground job is stuck in your shell. Now, let's put the process to sleep, then re-enable it in the background.

* Hit control+z
* Type: bg 1, sending the first job into the background and starting it.

**nohup**

When working from a shell or terminal, it is worth noting that by default all the processes and jobs attached to the shell will terminate when the shell is closed or the user logs out. When using nohup the process will continue to run if the user logs out or closes the shell to which the process is attached.

**ps Command**

The ps command is commonly used by administrators to investigate snapshots of a specific process. ps is commonly used with grep to filter out a specific process to analyze.

In the above command, we see all the processes using the python interpreter. Also included with the results were our grep command, looking for the string python.

Following are the most common command line switches used with ps.

|  |  |
| --- | --- |
| Switch | Action |
| a | Excludes constraints of only the reporting processes for the current user |
| x | Shows processes not attached to a tty or shell |
| w | Formats wide output display of the output |
| e | Shows environment after the command |
| -e | Selects all processes |
| -o | User-defined formatted output |
| -u | Shows all processes by a specific user |
| -C | Shows all processes by name or process id |
| --sort | Sorts the processes by definition |

**pstree Command**

pstree is similar to ps but is not often used. It displays the processes in a neater tree fashion.

**top Command**

top is one of the most often used commands when troubleshooting performance issues in Linux. It is useful for real-time stats and process monitoring in Linux. Following is the default output of top when brought up from the command line.

Common hot keys used while running top (hot keys are accessed by pressing the key as top is running in your shell).

|  |  |
| --- | --- |
| Command | Action |
| b | Enables / disables bold highlighting on top menu |
| z | Cycles the color scheme |
| l | Cycles the load average heading |
| m | Cycles the memory average heading |
| t | Task information heading |
| h | Help menu |
| Shift+F | Customizes sorting and display fields |

Following are the common command line switches for top.

|  |  |
| --- | --- |
| Command | Action |
| -o | Sorts by column (can prepend with - or + to sort ascending or descending) |
| -u | Shows only processes from a specified user |
| -d | Updates the delay time of top |
| -O | Returns a list of columns which top can apply sorting |

**kill Command**

The kill command is used to kill a process from the command shell via its PID. When killing a process, we need to specify a signal to send. The signal lets the kernel know how we want to end the process. The most commonly used signals are −

* SIGTERM is implied as the kernel lets a process know it should stop soon as it is safe to do so. SIGTERM gives the process an opportunity to exit gracefully and perform safe exit operations.
* SIGHUP most daemons will restart when sent SIGHUP. This is often used on the processes when changes have been made to a configuration file.
* SIGKILL since SIGTERM is the equivalent to asking a process to shut down. The kernel needs an option to end a process that will not comply with requests. When a process is hung, the SIGKILL option is used to shut the process down explicitly.

**pkill** will allow the administrator to send a*kill* signal by the process name.

**killall**will kill all the processes. Be careful using *killall*as root, as it will kill all the processes for all users.

**free Command**

free is a pretty simple command often used to quickly check the memory of a system. It displays the total amount of used physical and swap memory.

**nice Command**

nice will allow an administrator to set the scheduling priority of a process in terms of CPU usages. The niceness is basically how the kernel will schedule CPU time slices for a process or job. By default, it is assumed the process is given equal access to CPU resources.

**renice**

renice allows us to change the current priority of a process that is already running.

**Chapter 3**

**CPU Scheduling**

**Linux scheduling**

Based on the hardware layout of the physical cores, the Linux® scheduler maintains hierarchically ordered scheduling domains.

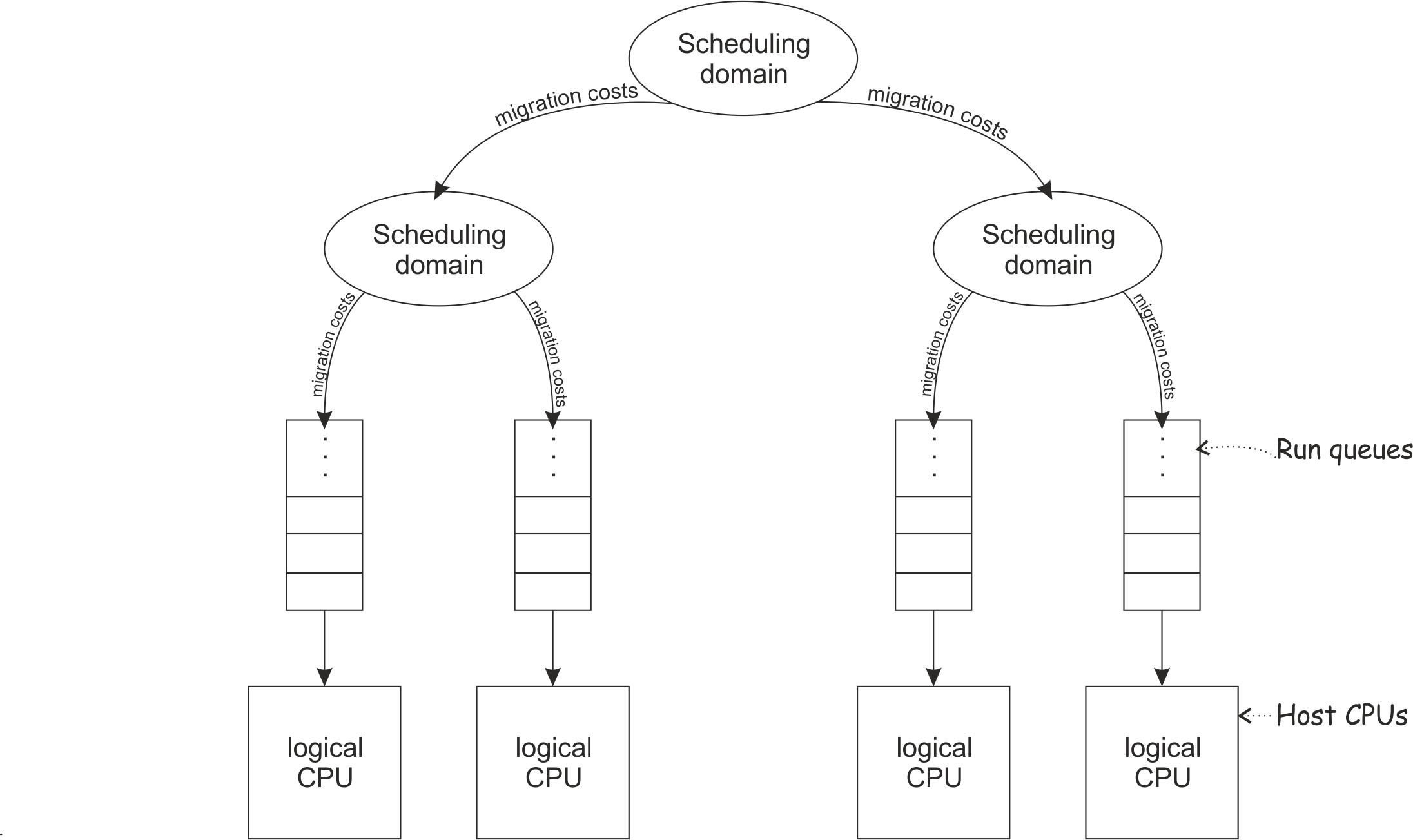
Basic scheduling domains consist of those processes that are run on physically adjacent cores, such as the cores on the same chip. Higher level scheduling domains group physically adjacent scheduling domains, such as the chips on the same book.

The Linux scheduler is a multi-queue scheduler, which means that for each of the logical host CPUs, there is a run queue of processes waiting for this CPU. Each virtual CPU waits for its execution in one of these run queues.

Moving a virtual CPU from one run queue to another is called a (CPU) migration. Be sure not to confuse the term “CPU migration” with a “live migration”, which is the migration of a virtual server from one host to another. The Linux scheduler might decide to migrate a virtual CPU when the estimated wait time until the virtual CPU will be executed is too long, the run queue where it is supposed to be waiting is full, or another run queue is empty and needs to be filled up.

Migrating a virtual CPU within the same scheduling domain is less cost intensive than to a different scheduling domain because of the caches being moved from one core to another. The Linux scheduler has detailed information about the migration costs between different scheduling domains or CPUs. Migration costs are an important factor for the decision if the migration of a virtual CPU to another host CPU is valuable.

Figure 1. Linux scheduling



libvirt provides means to assign virtual CPUs to groups of host CPUs in order to minimize migration costs. This process is called CPU pinning. CPU pinning forces the Linux scheduler to migrate virtual CPUs only between those host CPUs of the specified group. Likewise, the execution of the user space process or I/O threads can be assigned to groups of host CPUs.

**Chapter 4**

**Memory Management**

The memory configured for a virtual server appears as physical memory to the guest operating system but is realized as a Linux® virtual address space.

Virtual server memory has the same characteristics as virtual memory used by other Linux processes. For example, it is protected from access by other virtual servers or applications running on the host. It also allows for memory overcommitment, that is, the amount of virtual memory for one or more virtual servers may exceed the amount of physical memory available on the host.

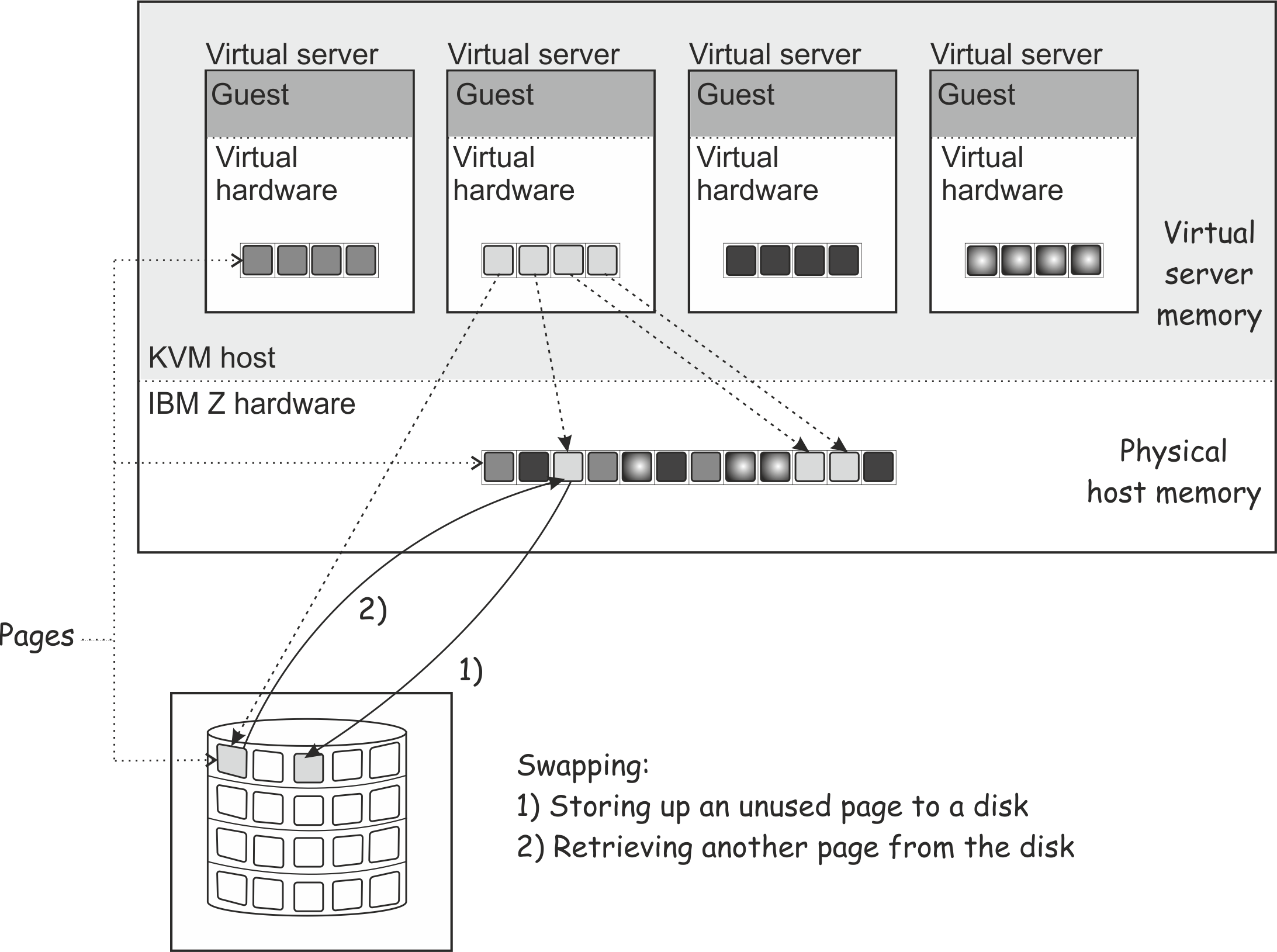
Memory is organized in fixed size blocks called pages. Each virtual server memory page must be backed by a physical page of the host. Since more virtual pages than physical pages can exist, it is necessary that the content of currently unused virtual pages can be temporarily stored on a storage volume (swap device) and retrieved upon access by the guest. The activity of storing pages to and retrieving them from the disk is called swapping.

Figure 1. Swapping

Since disk storage access is significantly slower than memory access, swapping will slow down the execution of a virtual server even though it happens transparently for the guest. Careful planning of virtual server memory handling is therefore essential for an optimal system performance.

Even if the defined virtual server memory exceeds the physical host memory significantly, the actual memory usage of a host may be considerably less than the defined amount. There are multiple techniques allowing the host to efficiently deal with memory overcommitment:

## Using huge pages

You can back the virtual memory of a virtual server with 1 MB huge pages of host memory. Depending on your workload, huge pages can result in performance gains.

**Prerequisites on the host**

* The host must be configured with enough 1 MB pages to satisfy the needs of its guests.
* The kvm module must be loaded with the hpage parameter to support guest-configurations with huge-page memory backing.

For more information, see the large pages (huge pages in recent editions) section in Device Drivers, Features, and Commands.

**Restrictions for guests**

* Transparent huge pages (THP) are not supported.
* The huge page size must be 1 MB.
* Memory overcommitment is not possible.
* Huge pages cannot be freed.
* Collaborative memory management is disabled on guests that use huge pages.
* KVM guests that use huge pages cannot be KVM hosts for higher-level guests.
* You cannot use huge pages for KVM guests that are to run in IBM® Secure Execution mode.

## Collaborative memory management

A guest operating system can mark memory pages as unused or volatile with the IBM Z® Collaborative Memory Management Assist (CMMA) facility. This allows the host to avoid unnecessary disk swapping because unused pages can simply be discarded. Current Linux operating systems make use of CMMA. The subset of the CMMA facility as used by Linux is enabled in KVM, therefore transparently ensuring efficient physical host memory usage, while still allowing the virtual server to use all of the defined virtual memory if needed.

## Ballooning

KVM implements a virtual memory balloon device that serves the purpose of controlling the physical host memory usage of a virtual server. With the balloon device, the host can request that the guest gives up memory. This could be done to re-balance the resource allocations between virtual servers to adapt to changing resource needs.

Whether and to which extent the guest honors the request depends on a few factors not controlled by the host, such as, whether or not a balloon device driver is installed in the guest, or whether there's enough memory that can be freed.

Unlike for CMMA, the memory given up by the balloon device is removed from the virtual server and cannot be reclaimed by the guest. As this can cause adverse effects and even lead to program or operating system failures due to low memory conditions, it should only be used in well-understood situations. By default, you should disable the balloon device by configuring <memballoon model="none"/>.

## Memory tuning

Another way to control virtual server memory usage is by means of the Linux cgroups memory controller. By specifying a soft limit the amount of physical host memory used by a virtual server can be restricted once the host is under high memory pressure, that is, the host is experiencing high swapping activity. Again, this would typically be done to re-balance resource allocations between virtual servers.

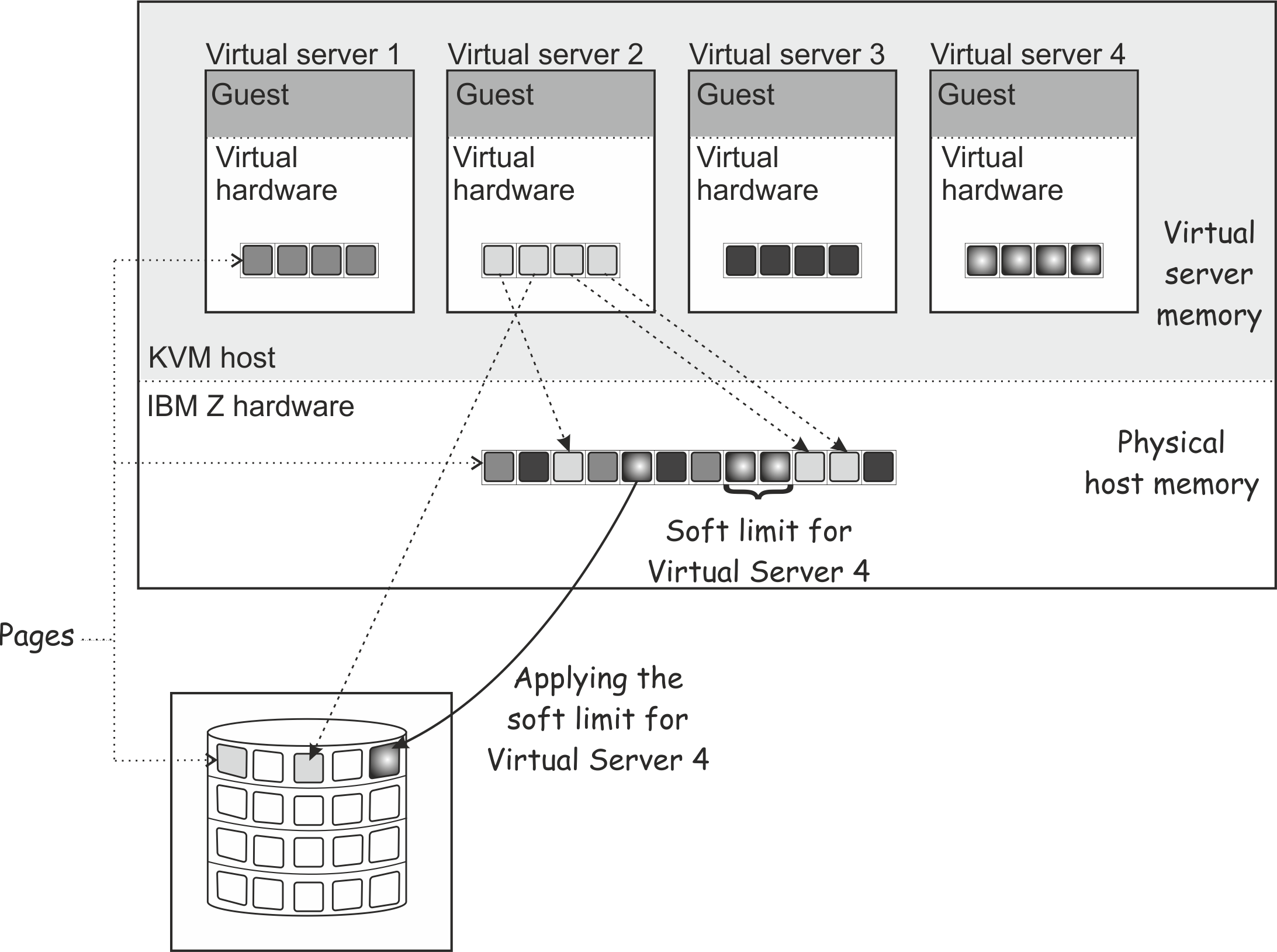
Since the virtual server memory available to the guest is not modified, applying a soft limit is transparent, except for the performance penalty caused by swapping. If swapping becomes excessive, time-critical processes may be affected, causing program failures. Therefore the soft limit should be applied carefully as well.

Figure 2. Applying the soft limit

**Chapter 5**

**Storage Management**

**I/O threads**

To attain good performance of I/O operations, configure each virtual block device to use an I/O thread.

I/O threads are intended to enhance the performance but they consume processing and memory resources. Defining an excessive number of I/O threads can be counterproductive. Do not configure more I/O threads than available host CPUs. Do not configure more I/O threads than virtual block I/O devices that will be available for the virtual server.

You can configure multiple devices to use the same thread. Which mapping of threads to devices yields best results depends on the available resources and on the workload.

**Logical volume management**

Consider these aspects when the virtual server utilizes logical volumes.

**Path redundancy**

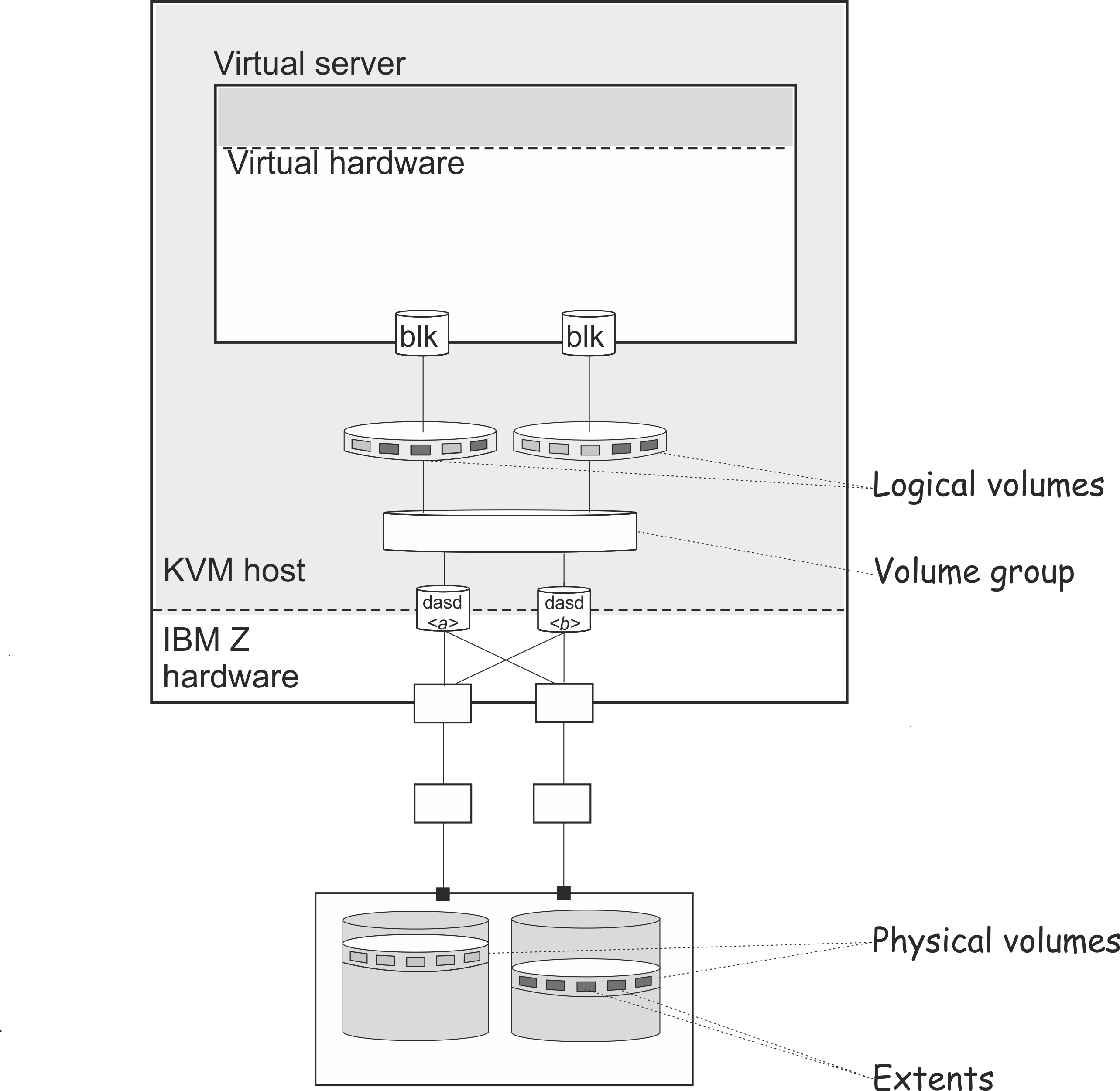
As discussed in Virtual Block Device it is important to ensure that you provide path redundancy for all physical volumes. Especially, all LVM physical volumes on SCSI disks have to be assembled from device mapper-created device nodes.

**Data integrity**

There are two ways to manage logical volumes:

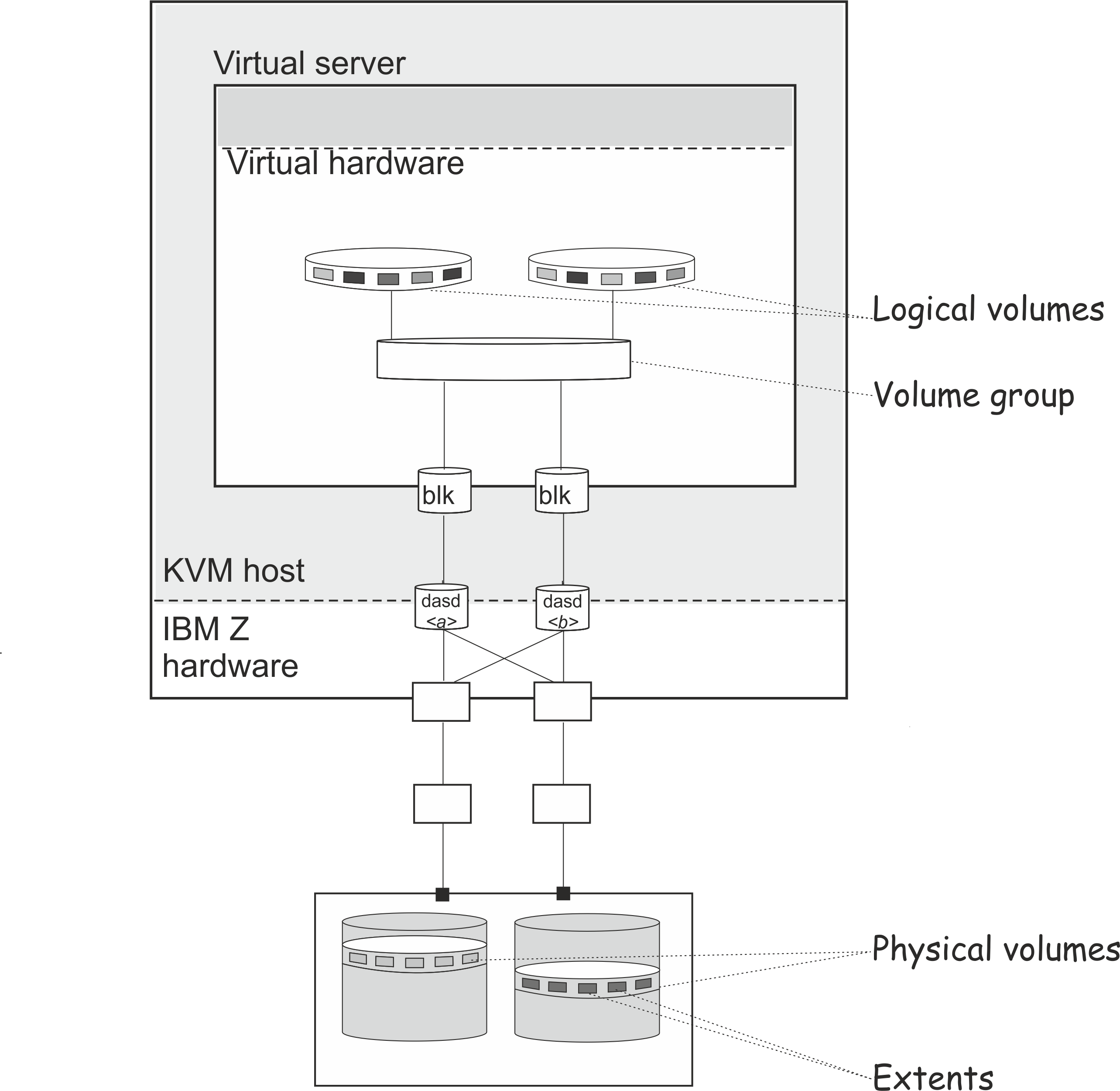
* On the host:

This example shows multipathed DASDs. The logical volumes that are managed on the host are configured as virtual block devices.



* On the virtual server:

When you configure physical volumes as virtual block devices, the logical volumes are managed on the virtual server. In this case you need to prohibit a logical volume management of the configured physical volumes on the host. Else, the host might detect the physical volumes and try to manage them on the host, too. Storing host metadata on the physical volumes might cause a loss of virtual server data.



To prohibit a logical volume management for physical volumes that are managed on the virtual server, provide an explicit allowlist in /etc/lvm/lvm.conf which explicitly contains all disk block devices to be managed on the host, or a blocklist that contains all physical volumes that are to be managed on the virtual server.

The filter section in the device settings allows to specify an allowlist using the prefix “a”, and to specify a blocklist using the prefix “r”.

**CHAPTER 6**

**I/O Systems**

\begin{figure}
\centering
\epsfig{file=EPS/iopath1.eps,width=2in,height=2.4in}\end{figure}Linux uses *request* structures to pass the I/O requests to the devices. All the block devices maintain a list of *request* structures. When a buffer is to be read or written, the kernel calls ll\_rw\_block() routine and passes it an array of pointers to buffer heads. ll\_rw\_block() routine in turn calls make\_request() routine for each buffer. make\_request() first tries to cluster the buffer with the existing buffers in any of the *request* structures present in the device queue. A *request* structure consists of a list of buffers which are adjacent on the disk. This clustering is performed only for the drivers compiled in the kernel and not for loadable modules. If clustering is possible, no new *request* structure is created, otherwise a new *request* is taken from the global pool of structures and initialized with the buffer and is passed to the add\_request() routine. This routine applies the elevator algorithm using insertion sort based on the minor number of the device and the block number of buffer. If the device queue is empty, the kernel calls the strategy routine i.e. the request\_fn() of the driver; otherwise, it is the responsibility of the driver to reinvoke it from the interrupt context (see Figure [4](https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/freenix/full_papers/gopinath/gopinath_html/node14.html#fiopath)). Another requirement for request\_fn() is that it cannot block as it needs to be called from the interrupt context.

|  |
| --- |
| **Figure 4:** I/O flow in Linux |
|  |

\begin{figure}
\centering
\epsfig{file=EPS/appds.eps}
\end{figure}

|  |
| --- |
| **Figure 5:** Data structure passed to ll\_rw\_block() |
|  |

To allow the accumulation of requests in the device queue, a plug is used. When the request comes in and the device queue is empty, the plug is put at the head of the device queue, and a task comprising of the unplug function is registered in the disk task queue. Thus the requests keep on accumulating for some time and then the task queue executes the unplug routine which removes the plug and calls the request\_fn() to service the requests.

**CHAPTER 7**

**File Systems**

File systems are the operating system's method of ordering data on persistent storage devices like disks. They provide an abstracted interface to access data on these devices in such a way that it can be read or modified efficiently. Which file system is convenient depends on the target application of the operating system. For example, Windows uses the FAT32 or NTFS file system. If a disk has a large capacity, FAT32 is inconvenient, because the FAT system was designed considering the smaller disks available at that time. At the same time, an NTFS file system is not convenient on a tiny disk, because it was designed to work with large volumes of data - there would be excessive overhead when using devices such as a 1.44 MB floppy disk.

**File System Theory**

A filesystem provides a generalized structure over persistent storage, allowing the low-level structure of the devices (e.g., disk, tape, flash memory storage) to be abstracted away. Generally speaking, the goal of a filesystem is allowing logical groups of data to be organized into files, which can be manipulated as a unit. In order to do this, the filesystem must provide some sort of index of the locations of files in the actual secondary storage. The fundamental operations of any filesystem are:

* Tracking the available storage space
* Tracking which block or blocks of data belong to which files
* Creating new files
* Reading data from existing files into memory
* Updating the data in the files
* Deleting existing files

(Perceptive readers will note that the last four operations - Create, Read, Update, and Delete, or CRUD - are also applicable to many other data structures, and are fundamental to databases as well as filesystems.)

Additionally, there are other features which go along with a practical filesystem:

* Assigning human-readable names to files, and renaming files after creation
* Allowing files to be divided among non-contiguous blocks in storage, and tracking the parts of files even when they are fragmented across the medium
* Providing some form of hierarchical structure, allowing the files to be divided into directories or folders
* Buffering reading and writing to reduce the number of actual operation on the physical medium
* Caching frequently accessed files or parts of files to speed up access
* Allowing files to be marked as 'read-only' to prevent unintentional corruption of critical data
* Providing a mechanism for preventing unauthorized access to a user's files

Additional features may be found on some filesystems as well, such as automatic encryption, or journalling of read/write activity.

**Indexing Methods**

There are several methods of indexing the contents of files, with the most commonly used being i-nodes and File Allocation Tables.

**Inodes**

inodes (information nodes) are a crucial design element in most Unix file systems: Each file is made of data blocks (the sectors that contains your raw data bits), index blocks (containing pointers to data blocks so that you know which sector is the nth in the sequence), and one inode block.

The inode is the root of the index blocks, and can also be the sole index block if the file is small enough. Moreover, as Unix file systems support hard links (the same file may appear several times in the directory tree), inodes are a natural place to store Metadata such as file size, owner, creation/access/modification times, locks, etc.

**FAT**

The File Allocation Table (FAT) is the primary indexing mechanism for MS-DOS and it's descendants. There are several variants on FAT, but the general design is to have a table (actually a pair of tables, one serving as a backup for the first in case it is corrupted) which holds a list of blocks of a given size, which map to the whole capacity of the disk.

**Workings of File Systems**

There are several common approaches to storing disk information. However, in comparison to memory management, there are some key differences in managing disk media:

* Data can only be written in fixed size chunks.
* Access times are different for different locations on the disk. Seeking is usually a costly operation.
* Data throughput is very small compared to RAM
* Data commonly has to be maintained

Hence some file systems have specialized structures, algorithms, or combinations thereof to improve speed ratings.

**Allocation Table**

The allocation table is comparable to the bitmap approach. However instead of just having a field free or occupied, it may contain other information. Advantage is that the use of this structure is simple, Disadvantage is that this approach is relatively slow, and a separate set of data is needed to define the which sections are used, and in which order. FAT for example, combines a linked list and an allocation table in the same structure.

**Separate file and system areas**

Some filesystems keep metadata and actual contents in separate areas on the disk. This makes specific sorts of data easy to find, as well as allowing for a special 'free' area. Downside is that you have to keep track of the boundary or boundaries between these areas, as the usage can differ and the disk has to adapt for that (big files: more data, less metadata; small files: less data, more metadata). This method is used prominently in SFS.

**Network File Systems**

All these file systems are a way to create a large, distributed storage system from a collection of "back end" systems. That means you cannot (for instance) format a disk in 'NFS' but you instead mount a 'virtual' NFS partition that will reflect what's on another machine. Note that a new generation of file systems is under heavy research, basing on latest P2P, cryptography and error correction techniques (such as the Ocean Store Project or Archival Intermemory.)

**File systems for OSDevers**

While you could pick <insert favorite filesystem here> as your OS main filesystem, you might want to consider all options. Commonly, you'll want to have a filesystem that is operational very quickly so that you can concentrate on the rest before implementing a 'real filesystem'.

**"Beginners" filesystems**

There are only five filesystems that are both relatively easy to implement and worth to consider. There is no general recommendation as the choice depends largely on style and OS design. Instead you can read the comparison and make your own educated decision.

USTAR

* + Of these beginner "filesystems", this is the simplest by far to implement
* + Uses 512 byte sectors, just like floppies and disks
* + Incredibly simple: a sector with metadata followed by data sectors
* + Widely used: a utility to create tar images are available for every mainstream OS
* + Supports special files (like devices and symlinks)
* + Supports Unix permissions
* Not a filesystem in the common understanding of the term
* Generally read-only, was never designed for in-place modifications
* No support for fragmentation
* No standard partition type for it (not that you should even consider using USTAR as a disk partition format)
* Not actually the format used for ramdisks by things like Linux - that's CPIO

**RAMFS/TMPFS**

* + High flexibility of implementation
* + Fast
* + Will allow you to test out your VFS API without having to rely on filesystem specifics
* + \*Highly\* recommended as a starter filesystem to avoid morphing your VFS interface around a specific filesystem
* + Ideal to unpack a USTAR or CPIO injtrd image into
* Changes are, obviously, not persistent, and only in memory, to be wiped after a reboot

FAT

* + Can be read and written by virtually all OSes
* + The 'standard' for floppies
* + Relatively easy to implement
* - Part of it involving long filenames and compatibility is patented by microsoft
* - Large overhead
* - No support for large (>4 GB) files
* - No support for Unix permissions

**Ext2**

* + Supports large files (with an extension)
* + Supports Unix permissions
* + Can be put on floppies
* + Can be read and written from Linux
* - Can not natively be read and written from Windows (but drivers are available)
* - Very large overhead
* - Of these beginner filesystems, this is the most complex

**BMFS**

* + Supports large files
* + Implementation Available as static library
* + Comes with utility program for creating disk images on Linux and Windows
* + Comes with FUSE bindings, allowing it to be mounted on Linux systems
* + Contains source code documentation
* - Does not support fragmentation
* - Less control over the source code

**References:**

[Talha Saif Malik](https://linuxhint.com/author/talha_saif_malik/), (2021) – Everything You Want to Know About CentOS as Linux Distribution

About CentOS. <https://linuxhint.com/everything-you-want-to-know-about-centos-as-linux-distribution/#1>

[Talha Saif Malik](https://linuxhint.com/author/talha_saif_malik/), (2021) - Everything You Want to Know About CentOS as Linux Distribution

History of CentOS. <https://linuxhint.com/everything-you-want-to-know-about-centos-as-linux-distribution/#1>

Linux Admin (N/A) - Process Management

Work with process. <https://www.tutorialspoint.com/linux_admin/linux_admin_process_management.htm>

Linux on IBM Systems, (2023)

Linux Scheduling. <https://www.ibm.com/docs/en/linux-on-systems?topic=management-linux-scheduling>

Letzte Aktualisierung, (2023)

Memory Management. <https://www.ibm.com/docs/en/linux-on-systems?topic=performance-memory-management>

Copyright IBM Corporation 2016,(Updated 2023) -

Storage Management. <https://www.ibm.com/docs/en/linux-on-systems?topic=performance-storage-management>

Dr K Gopinath, (2000)

I/O Systems. <https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/freenix/full_papers/gopinath/gopinath_html/node14.html>

OSDev.org, (2022) -

File Systems. https://wiki.osdev.org/File\_Systems