ADDING MULTIPATHING CAPABILITIES TO LVM

STEFAN BADER

LINUX–KONGRESS 2002

IBM DEVELOPMENT, GERMANY

TABLE OF CONTENT

1 INTRODUCTION 1

2 MULTI−PATH APPROACHES IN LINUX 3

2.1 Extending the SCSI mid−layer 3

2.2 Generic extensions to the block layer 3

2.2.1 MD multi−path personality 4

2.2.2 EVMS multi−path module 4

2.2.3 Multi−path handling in LVM 4

3 LVM MULTI−PATH DESIGN 5

3.1 Requirements 5

3.2 General design 6

3.3 Changed and additional data structures 6

3.4 The IO scheduler 10

3.5 The end IO function 11

3.6 Error handling 12

4 LINKS 13

Figures

Figure 1: Connecting single path devices 1
Figure 2: Connecting multi–path devices 2
Figure 3: Multi–path enhancements to LVM 5
Figure 4: The IO scheduling algorithm 10

1 INTRODUCTION

When looking at the way storage devices are connected to a computer there was one big difference between personal computers and workstations on the one side and mainframes on the other side. For the former this looked like the image below.

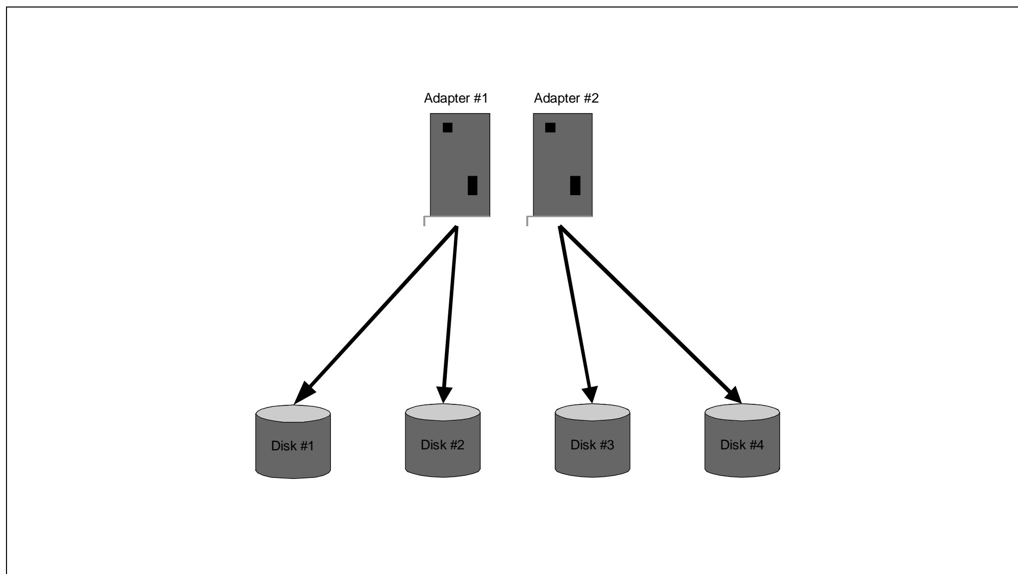

Figure 1: Connecting single path devices

Any connected device can be associated with exactly one host controller. There can be several host controllers and each controller can be responsible for several devices but the path that data traverses on it's way is always the same. This is the way IDE or parallel SCSI works.

While the data on the storage device can be protected using RAID systems this doesn't increase reliability of the transfer between computer and storage. If one controller or cable/bus fails this means all devices that are connected to this chain can't be accessed anymore. The data is safe but this means service outage.

Since downtime in the mainframe world always was a critical issue, alternate approaches to this were established quite a long time ago. Figure 2 shows how the single path example would change if two disks had alternate paths.

Now, should one adapter fail there is still another one left to talk to the device. Main–frame specific device attachments (e.g. ESCON) handle path failures transparently in

hardware. So an operating system won't even notice single path failures. However with the introduction of SCSI it is necessary that the operating system is aware of this redundant hardware layout. This is true not only for the mainframe environment but also for storage area networks (SAN) and also some special parallel SCSI boxes.

If the operating system is aware of multiple data paths it can also use them to increase the throughput to or from the devices (load balancing).

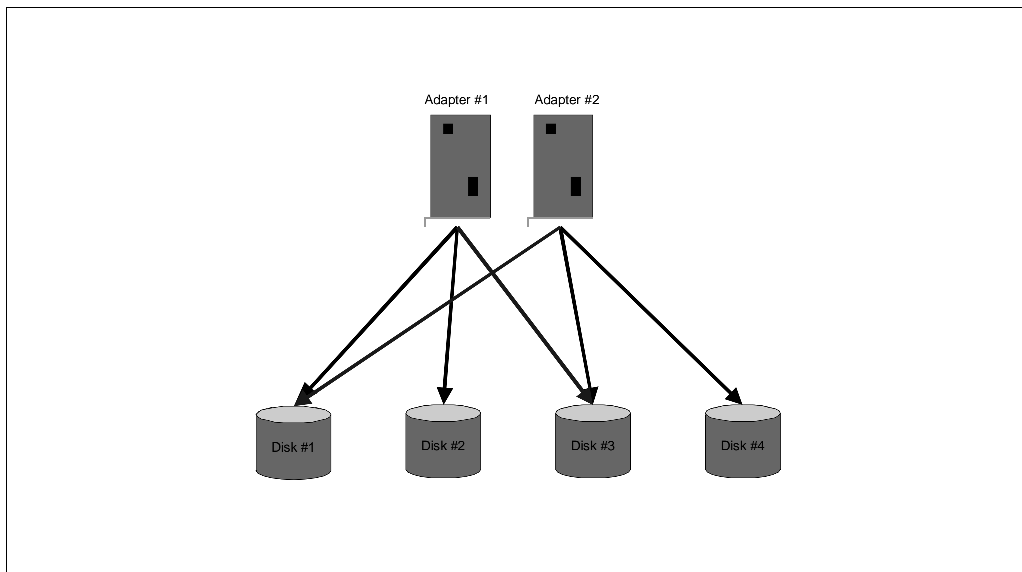

Figure 2: Connecting multi−path devices

However, even with multiple paths, data cannot be transferred faster than the storage device can handle it. So it is unlikely that performance would increase for parallel SCSI boxes but in the SAN environment this could make a difference.

2 MULTI–PATH APPROACHES IN LINUX

There are several approaches that were made to add multi–path capabilities to Linux. The following section is meant to give a general overview on these approaches.

2.1 EXTENDING THE SCSI MID–LAYER

The current SCSI implementation breaks the driver roughly into three parts: the low–level driver, the mid–layer and the device driver. The low–level drivers handle the communication to host adapters and are very hardware specific. On the other side there are device drivers for each “type” of SCSI device (disk, tape, generic). These are specific to their class of devices. Between them the SCSI mid–layer handles every–thing that is common to the SCSI protocol.

Extending the mid–layer would have the advantage of being closer to the supported hardware. Most of the currently available storage devices are SCSI devices. So the missing support for other flavors of storage devices would have little impact. This approach would also hide additional paths from the upper layer, which means that only one device is visible to the block device layer. This saves major/minor[1] numbers and prevents accidentally accessing the same device using different block devices.

The downside is that the SCSI mid–layer is about to be removed in Linux 2.5. And o there is also the problem that a lot of SCSI mid–layer code expects a device to be connected to a specific host.

2.2 GENERIC EXTENSIONS TO THE BLOCK LAYER

At the time of writing there are three approaches that can be counted as block layer approaches. The multi–path personality of the multiple disk (MD) soft–RAID driver, the additions to LVM. And finally a multi–path module for the enterprise volume management system (EVMS) which is, not yet, part of the standard Linux kernel.

1 In 2.4 a block driver has to use a unique major–minor number combination for every partition it provides. Since the range for both numbers is 0 to 255 this might be a problem if there are many block devices.

2.2.1 MD MULTI–PATH PERSONALITY

This has been added to the main kernel stream with Linux 2.4.13 and adds a new personality[2] to MD. Currently there is only "failover" support which means if one path fails the next available path is used. Failed paths stay in the disabled state until they are enabled again with some command line tool. New paths have to be added manually by updating the MD configuration file.

2.2.2 EVMS MULTI–PATH MODULE

EVMS is a device mapper project that aims at integrating everything that is related to device management. They integrate partitioning and file system management support as well as MD or LVM functionality (beside other volume group support). The multi–path module was introduced with the 1.0 release of EVMS. However it currently supports a very host–specific type of storage devices only, which is the major draw–back of that implementation.

2.2.3 MULTI–PATH HANDLING IN LVM

This was the aim of the project described in this document. What made LVM interesting for the multi–path additions was its unique device ID that gets written to devices as a part of the LVM meta–data and the way volume groups are built on startup which is not based on a static configuration file but on the meta–data that is found on the devices available.

2 In MD each personality normally implements one specific type of RAID (striping, mirroring and so on). The idea is that multiple paths are in some way related to mirroring. The difference is that with multiple paths you can write to any device while you have to write on every device to do soft–mirroring.

3 LVM MULTI–PATH DESIGN

This section details the LVM approach mentioned in the previous paragraph. One important goal was to keep the enhanced LVM code compatible to the existing implementation. So that changing the meta–data that gets written to the devices was not an option. This also implies that the only allowed changes to the interface between LVM utilities and the kernel would be additions, so that the using older utilities would still be possible.

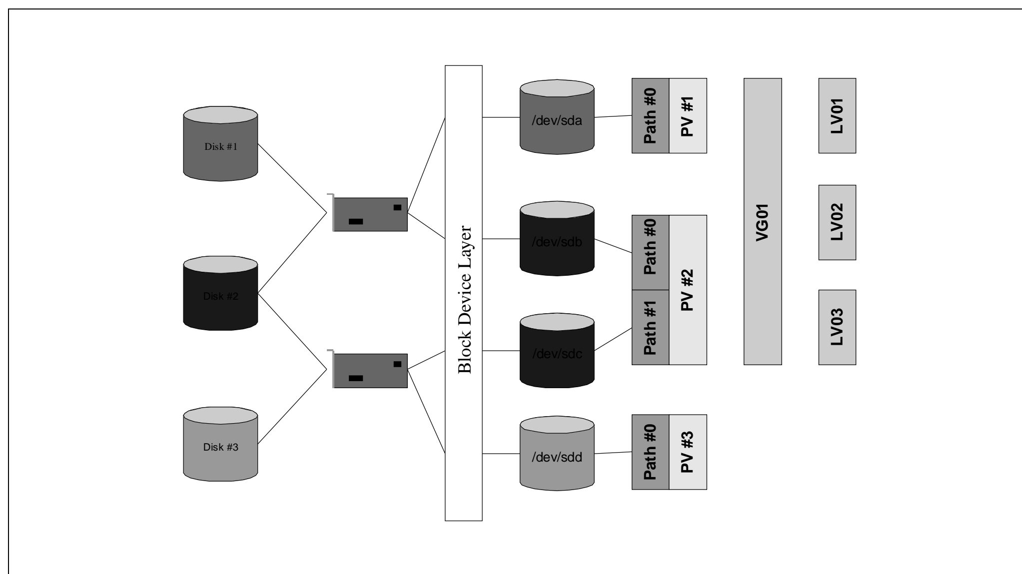

Figure 3: Multi–path enhancements to LVM

3.1 REQUIREMENTS

Since LVM – as any other block device handler – operates on other block devices, there must be a visible device for every path. “Visible” meaning it is visible as an independent device to the block layer (or more technically has its own entry in the generic disk structures).

Since LVM doesn’t know anything about special hardware it is necessary that the hardware, or the specific device driver, can handle requests to any of these path

devices without special intervention. Some devices do but would give very bad performance if there are many path changes. For these devices it should be possible to do only *failover* and no *load balancing*.

3.2 GENERAL DESIGN

To decide which block device is part of a logical volume LVM uses the meta–data written to every device. Part of this data is a unique device ID. This ID is now also used to detect multiple paths to the same device.

The way LVM treats a physical volume is enhanced to support a physical volumes consisting of up to 16 block devices which make up the paths. This adds one level of indirection to the mapping between logical and physical volume.

Without multi–path capabilities a logical volume consists of several physical extends which belong to one of the physical volumes of the volume group. A physical extend is the smallest amount of storage that LVM takes from a physical volume to assign it to a logical volume. For a given offset into a logical volume LVM has to translate that into a logical extend (that is a certain offset into a physical volume) and the offset within that physical extend.

To make this translation faster the internal representation of a physical extend is linked directly to a block device. This would make it very hard to add the multi–path layer. It would make the new lookup much slower if the code has to search through all physical volumes to find the one specified in the mapping and to decide then which path device should be used. To add all the path information to the lookup table would lead to much duplicated information since that would have to be added to every map entry. So now the mapping table holds the link to the physical volume structure and this holds the information about block devices.

3.3 CHANGED AND ADDITIONAL DATA STRUCTURES

One general change that had to be made affects the way LVM translates physical extends into sectors on a physical volume. The direct approach would be to replace the device link with the link to the physical volume. But that would require the tools to know about that change and prevent older tools from working if they would be used on a patched LVM module.

Because of this the new mapping is used only within the kernel module. Tools present the same structures as before and the kernel code translates the table at the time the data is read from user space. The way back to user space works similar. The map is translated back and then written to user space.

```
typedef struct {
    kdev_t dev;
    uint32_t pe;        /* to be changed if > 2TB */
    uint32_t reads;
    uint32_t writes;
} pe_t;
/* This structure will be used only in the kernel */
typedef struct {
    uint32_t devno;
    uint32_t pe;        /* to be changed if > 2TB */
    uint32_t reads;
    uint32_t writes;
} le_t;
typedef struct lv_v5 {
         ...
#ifdef __KERNEL__
         le_t *lv_current_pe;
#else
         pe_t *lv_current_pe;    /* HM */
#endif
         ...
} lv_t;
```

This change by itself doesn't add any new functions to the code. To get LVM to recognize multiple paths there are further changes needed. Again this has to be done carefully in order to prevent problems with older tools. This time however it is not enough to hide elements inside the kernel since the path detection and setup has to be done in user space.

The kernel module itself doesn't do much about discovery. That is all done by user space tools (namely **vgscan**). These tools create the complete kernel structures and pass them to the kernel via IOCTL calls. Since the kernel structure for physical volumes is enlarged by the additional path information all IOCTL calls that pass this structure along would get into trouble. This is prevented by adding new versions of these calls (vgcreate, vgextend and pvstatus) which will only be used be the patched tools.

Now information about the paths has to be added to the internal representation of the physical volumes.

```
typedef struct pv_v2 {
         ...
         char                 pv_name[NAME_LEN];
         kdev_t               pv_dev;
         ...
         struct block_device *bd;
         char pv_             uuid[UUID_LEN+1];
         uint32_t             pe_start;

         pv_path_t            pv_path[PV_MAX_PATHS];
         unsigned int         pv_paths;
         unsigned int         pv_current_path;
         int                  pv_path_kept;

#ifdef __KERNEL__
         spinlock_t           pv_lock;
         spinlock_t           pv_path_lock[PV_MAX_PATHS];
         struct block_device *pv_path_bd[PV_MAX_PATHS];
#else
         char padding[512];
#endif
} pv_t;
```

This layout may look strange but is necessary because most of the structure has to be compatible with the original one. So because the utilities pass this structure to the kernel whenever physical volumes are involved, all new elements have to be at the end.

For the new tools to work with unmodified kernels, the structure must still contain sensible values for the block device (pv_name, pv_dev and bd). The tools do this by setting this elements to the same values as the first path. To prevent the first path from changing too often, all paths are ordered by major and minor number, so the first path is always the device with the smallest major–minor combination.

The spinlock elements are hidden from user space since there is no need for the tools to know about them. Instead there is some padding space that must always be larger than the size of the structure in kernel space.

```
typedef struct {
        kdev_t              path_dev;
        unsigned int        path_weight;
        lvm_path_state_t    path_state;
        unsigned int        path_fail_count;
        unsigned int        path_pending_io;
} pv_path_t;
```

The pv_path_t structure is the way a path is represented in the physical volume. The “weight” is used to create groups of paths which will be explained in the scheduler section. The path state can be set to the following values:

```
typedef enum {
        lvm_path_enabled                          = 0,
        lvm_path_disabled_finish_requests,
        lvm_path_disabled_wait_retry,
        lvm_path_disabled_retry,
        lvm_path_disabled,
        lvm_path_invalid                          = -1
} lvm_path_state_t;
```

Three special “disabled states” are used to re–enable paths automatically. A path that fails is put into *disabled_finish_requests* state if there are still some requests processed by the path device. After the last request reports back that state is changed to *disabled_wait_retry*. Now, the path is skipped for a certain amount of requests. After that the state is set to *disabled_retry* and one request is sent to that path. If that request completes successfully the path is *enabled* again. Otherwise the path is put back into *disabled_wait_retry* and the procedure starts again. If the path is put into the *disabled* state there is no automatic recovery.

3.4 THE IO SCHEDULER

For any new request that gets sent to a logical volume the first step is to find out to which physical volume and offset this request would be mapped. This is the standard way LVM maps logical volumes to physical ones.

The difference in the multi–path case is that after finding a physical volume the IO scheduler chooses one of the paths to which the request is sent. The first mapping changes the sector and the device of a request, the second one only the device.

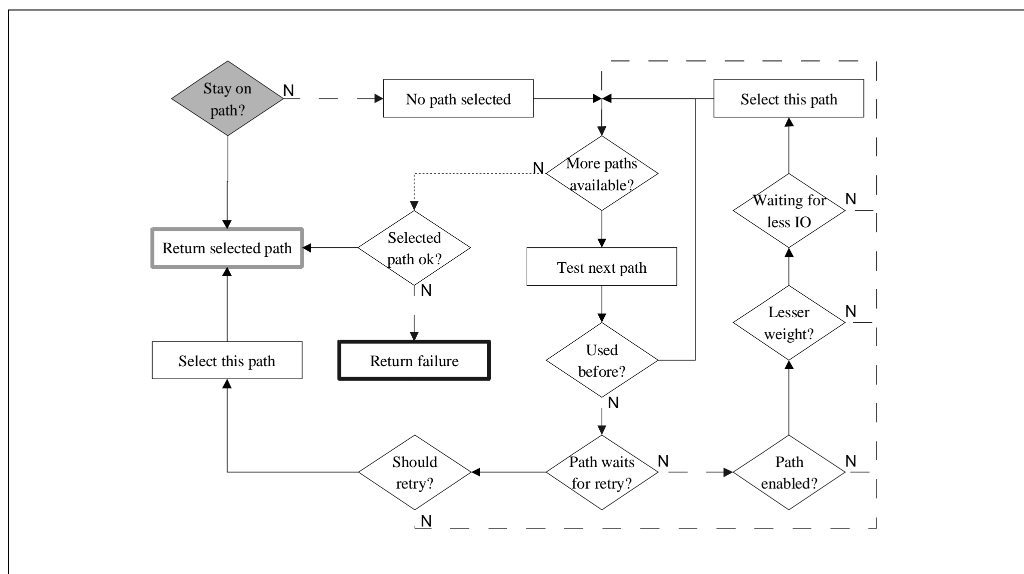

Figure 4: The IO scheduling algorithm

Figure 4 shows the implemented scheduling algorithm. The first step is to check whether the path that was used the last time has not been used for a certain number of requests. The goal is to cluster requests which makes it more likely that the device can merge them into bigger request blocks for more efficient handling.

If the path already has been used more often than defined in the scheduler logic, a new the path will be chosen. There are several requirements the new path has to fulfill. First of all, in case of error recovery, the path may not have been used for that request before. Then the algorithm looks at the *weight* and the number of unfinished requests.

The *weight* must be less or equal to the previously used path as long as there are any working paths left that match that criteria. Otherwise the path may have a higher weight.

For paths with the same weight the new path is the one that has the lowest number of unfinished requests. This results in slower paths being used less often than faster paths because these will process request at a slower rate.

If there is an eligible path found the multi–path mapper adds a private area that stores information for the callback and error handling to the buffer head. Then it replaces the "end IO" function with its own version (the old function pointer is stored in the private area as well). Finally the number of unfinished request for the selected path is increased by one and the request is sent to the path device.

3.5 THE END IO FUNCTION

For every buffer head that a block device is done with, the driver calls the *b_end_io* function. This is now pointing to the multi–path handler, which first decrements the number of unfinished requests and then checks whether the IO completed success–fully. If that is the case it may re–*enable* the path if the current state is *disabled_retry*. After that it restores the saved b_end_io pointer, removes the private data and then calls the original b_end_io function.

For requests that where not successful the handler puts it into an internal error handling queue, triggering the error handler which is implemented as an independent kernel thread. After that the end IO handler returns without any further action.

3.6 ERROR HANDLING

The error handler is a kernel thread that sleeps as long as there are no entries in the internal error handling queue. For every entry (failed request) it first tests the state of the used path. If it is *enabled* then it increments the failure count and if that reaches a certain value sets the state to either *disabled_finish_requests* or *disabled_wait_retry* depending whether there are still requests pending. If the state is *disabled_retry* then that is the result of a failed path test so the state will be *disabled_wait_retry*, again.

The handler then tries to find a working path that hasn't been used, yet and if such a path is found sends the request down that path. Should there be no path left the private data is removed and the end IO function is restored. It is then called to mark the request as unsuccessfully terminated.

4 LINKS

Logical Volume Manager: http://www.sistina.com/lvm

The EVMS project: http://sourceforge.net/projects/evms

The kernel: http://www.kernel.org