OpenVMS Technical Journal V5

Disk partitioning on VMS: LDdriver's secrets

Jur van der Burg

Overview

This article is about LDdriver, a device driver running under the OpenVMS operating system which allows creation of virtual disks. Since there are many more ways to use this driver which are hardly known, this article will show what's possible.

LDdriver was developed a long time ago (around 1985) and has been improved and extended a lot since then. It started as freeware (it still is) but also got integrated in OpenVMS since V7.3-1.

History.

LDdriver was originally written for VAX/VMS around 1985 by a former colleague of mine, Adrie Sweep, a software engineer in The Netherlands. The basic principle worked fine, although it was very inflexible in configuring and usage. Some time later the driver was changed to use so called cloned devices to allow for a very flexible way of managing the virtual disks. Over the years a number of things have been added to allow for far greater versatility, and LD has obviously been ported to Alpha. LD has been a part of OpenVMS since VMS V7.3-1, used by CDRECORD. Beginning in V8.2 LD is fully integrated. The possibilities described in this article are valid for the latest version of LD, V8.0 which will be available as a separate kit, or integrated in the next major VMS release (probably V8.3).

What is it?

LDdriver is, as its name implies, a Logical Disk driver running on OpenVMS. Its main use is to use a file on any type of hard disk as a disk. For example, we have a file called LDDISK:[VMS]DISK.DAT then we can use that file as a disk by using the command LD CONNECT LDDISK:[VMS]DISK.DAT LDA1:. After that we have a device LDA1: on the system which we can use as a disk.

To begin using LD its system startup procedure needs to run: SYS$STARTUP:LD$STARTUP, if needed it can be placed in SYS$SYSTEM:SYSTARTUP_VMS.COM. The normal way to use it is without any parameter, but if 9999 LD devices are not enough a parameter can be specified, and that is the controller letter to use for LD. If not specified then LDA will be used, but if one would like an additional controller like LDB then specifying B as the first parameter should do the trick.

If any help is needed, just remember the command LD HELP. Every command is described in detail, and a couple of examples are provided.

FILE mode.

The first mode of operation is called FILE mode, and is most widely used. It allows an arbitrary file to be used as a disk. The only restriction on the file itself is that it must fit on a real disk, the size may be as big as the physical disk can handle. In the past the file needed to be contiguous, but that restriction has been removed a long time ago.

The setup is very simple:

- Create a file on a physical disk
- Connect the file to a logical disk
- Use it!
- After use: Disconnect it.

Example:

```
$ ld create lddisk:[vms]disk.dsk/size=10000
$ ld connect lddisk:[vms]disk.dsk lda1:
$ ld show lda1:
%LD-I-CONNECTED, Connected $7$LDA1: to $7$DRA6:[VMS]DISK.DSK;1
```

At this time we have a new disk on the system, LDA1: which can do anything a normal disk can do:

```
$ initialize lda1: lddisk
$ mount/system lda1: lddisk
%MOUNT-I-MOUNTED, LDDISK mounted on _$7$LDA1: (LDDRVR)
$ dismount lda1:
$ ld disconnect lda1:
```

This setup allows for very flexible use of disk space. Suppose you want to create a VMS style disk that we want to burn on a CD, then all you need to do is to mount the LD device, copy any files on it the way you want, dismount the disk, disconnect the file from the LD device and burn the file on a cd anyway you like. This can also be done on a PC with a variety of pc software which allows burning of an arbitrary file on a cd.

More complex use of LD can be done by sharing the LD devices across different nodes in a cluster. Notice that LD devices itself are not mscp-served (for a number of reasons), so if sharing is to be used the physical disk where LD's container file resides needs to be visible on other cluster members.

```
$ ld connect lddisk:[vms]disk.dsk lda1:/share/log
%LD-I-CONNECTED, Connected $7$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

If the same command is given on another node in the cluster then the resulting LD device can be mounted on that node as well. For this to work there are several checks imposed by the driver.

First of all, the device name has to match, as well as the allocation class, unit number and controller letter. It must also connect to exactly the same file, and with the same sharing options. The maximum disk size and the device geometry need to be the same as well.

If one of these prerequisites is not met a clear message will follow. Suppose a device is connected on one node, and if we try to connect it on a second node we may face the following error:

```
$ ld connect lddisk:[vms]disk.dsk lda1:
%LD-F-FILEINUSE, File incompatible connected to other LD disk in cluster
-LD-F-ALLOCLASS, Allocation class mismatch
```

In this case we have an allocation class mismatch. This may occur when two nodes in a cluster have a different allocation class, and as a result the default allocation class of an LD device will be the allocation class of the node. This can however be changed at connect

time by specifying the correct allocation class. Notice that this may only be done if no other LD devices for the same controller are connected.

If we look at the other node where the file was first connected we see this:

```
$ ld show/all
%LD-I-CONNECTED, Connected _$12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

So be specifying the correct allocation class we can do what we want:

```
$ ld connect lddisk:[vms]disk.dsk lda1:/alloclass=12
%LD-F-DEVICEINUSE, Device incompatible connected to other LD disk in cluster
-LD-F-NOSHARE, No sharing specified for file on this node
```

Oops, that does not work either. Remember that everything has to be the same, and since we did not specify that we wanted to share the file the driver will reject our request. This will work however:

```
$ ld connect lddisk:[vms]disk.dsk lda1:/alloclass=12/share/log
%LD-I-CONNECTED, Connected $12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

Different LD devices with different allocation classes can be used on one node, but in that case they need a new controller letter which can be created by invoking LD's startup procedure with another parameter like B.

```
$ @sys$startup:ld$startup b
$ ld create disk2
$ ld connect disk2 ldb5
$ ld show/all
%LD-I-CONNECTED, Connected _$12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
%LD-I-CONNECTED, Connected _$7$LDB5: to $7$DRA6:[VMS]DISK2.DSK;1
```

Connect, create and other options.

A number of options can be specified at connect time to influence the appearance of the logical disk. For example, if a file was created with a size of 10000 blocks and we only want to use the first 5000, we can do so by specifying /MAXBLOCK=5000 at connect time. We can also influence the geometry by specifying /CYLINDERS, /TRACKS and/or /SECTORS. Although of limited use with modern disks this allows configuration of a disk to exactly mimic another disk. This geometry can also be copied from an already mounted disk with /CLONE, and it can also be used to create a container file which mimics a real disk:

```
$ ld create/clone=$8$dua14: lddisk:[vms]cloned.dsk
$ ld connect cloned lda2/log
%LD-I-CONNECTED, Connected $12$LDA2: to $7$DRA6:[VMS]CLONED.DSK;1
```

This will effectively create a logical disk LDA2 with exactly the same properties as device 8DUA14:. Notice that the geometry information is stored in the container file, so that a subsequent disconnect/connect will restore the same disk parameters. This can be overridden by specifying /NOAUTOGEOMETRY, in that case the geometry will be calculated by the driver.

The geometry information will also be saved in the container file if /SAVE was specified during connect.

LD devices can be made write-protect with the LD PROTECT command. This protection can be made permanent by adding /PERMANENT which will store the write-protect status in the container file. This way it will be possible for example to emulate a cdrom, on a subsequent connect command the write-protect status will then be restored.

Normally a file is connected to an LD device by explicitly specifying which device we want to get. If we don't do this the driver will assign a unit number, and inform us about it. If we specify /SYMBOL as well we will end up with the assigned unit number in a DCL symbol:

```
$ ld connect disk.dsk/symbol
%LD-I-UNIT, Allocated device is $12$LDA22:
$ show symbol ld_unit
  LD_UNIT = "22"
```

This can be useful in command procedures where the actual device name may be unimportant. The symbol makes it easy to reference such device.

LBN mode.

The second mode of operation is called LBN (Logical Block Number) mode. In this mode a physical disk can be accessed in several parts specified by a LBN range. Look at it as partitioning a disk without any file structure on it. Example:

```
$ ld connect $1$dga1: lda1:/lbn=(start=0,count=1000)/log
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (LBN Mapping: Start=0 End=999)
$ ld connect $1$dga1: lda2:/lbn=(start=1000,end=2000)/log
%LD-I-CONNECTED, Connected $7$LDA2: to _$1$DGA1: (LBN Mapping: Start=1000
End=2000)
```

This will use two fragments of device 1DGA1 for devices 7LDA1 and 7LDA2. The range of LBN's may be specified in two ways: either a starting lbn with a block count, or a starting lbn and an ending lbn. The driver will check for an overlap of lbn's, so any attempt to use a range already in use will result in an error:

```
$ ld connect $1$dga1: lda3:/lbn=(start=600,end=2000)/log
%LD-F-DEVICEINUSE, Device incompatible connected to other LD disk in cluster
-LD-F-RANGEINUSE, LBN range already in use
```

This way of partitioning a disk will be the most efficient way to use the physical device, as the overhead of unused blocks is zero. The physical device may be divided in as many parts as needed (as long as the system's resources will allow).

An alternative way to use this is to map a range of blocks from a foreign volume, for example a Unix filesystem. Use your imagination!

As with the FILE mode, these LD devices may be shared in a cluster:

```
$ ld connect $1$dga1: lda3:/lbn=(start=5000,count=5000)/log/share
%LD-I-CONNECTED, Connected $7$LDA3: to _$1$DGA1: (LBN Mapping: Start=5000
End=9999) (Shared)
```

The same restrictions apply as with FILE sharing, the device name, unit number, controller letter, and allocation class must match the remote node. The range and physical device have to match too of course. Notice that the range of LBN's in use is checked on all nodes in the cluster which have an interest in the physical device, so if any block in the specified range is already in use an error will be generated. This is really a tricky thing to check, for more info about this check the 'internals' chapter.

To be able to use LBN connect, the physical device must not be in use anywhere in the cluster. The driver will enforce a check on this, and if the check fails an error will follow. After the first LD device is connected via LBN the physical device will not be available for any other use in the cluster, i.e. mount on any node will fail until the last LD device disconnects., like this:

```
$ mount/over=id $1$dga1:
%MOUNT-I-OPRQST, device already allocated to another user
%MOUNT-I-OPRQST, device _$1$DGA1: (LDDRVR) is not available for mounting.
```

If any failure to use the physical device occurs on any node make sure that all LD devices are disconnected from this device. The driver will go through great length to protect the user against any errors.

REPLACE mode.

The third mode of operation is called REPLACE mode. It is not a form of partitioning, but a way to create access to the physical device via LDdriver. The use of this will become clear in the chapters about I/O tracing and watchpoints, where we can see that a real-time trace of all I/O requests can be done.

Example:

```
$ ld connect $1$dga1: lda1:/replace/log
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (Replaced)
```

This will create a device 7LDA1 which will direct all I/O to the physical device 1DGA1. Replaced devices can also be shared:

```
$ ld connect $1$dga1: lda1:/replace/log/share
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (Replaced) (Shared)
```

The same restrictions apply for sharing as with FILE and LBN mode: the device name, unit number, controller letter and allocation class must match. The physical device will be made unavailable with a lock.

I/O Tracing.

One of the most powerful and unknown features of LDdriver is I/O tracing. For any mode that an LD device operates a real-time trace of all I/O activity can be created. This is a simple example:

```
$ ld connect disk.dsk lda1
$ ld trace lda1
$ mount lda1: testdisk
%MOUNT-I-MOUNTED, TESTDISK mounted on _$7$LDA1: (LDDRVR)
$ ld show/trace lda1
       I/O trace for device $7$LDA1:
   26-APR-2005 22:18:36.28 on node LDDRVR::

Start Time  Elaps  Pid      Lbn    Bytes  Iosb    Function
-----------------------------------------------------------------
22:18:32.65 00.00 09C00227       0     0 NORMAL  PACKACK|INHERLOG
22:18:32.65 00.01 09C00227       1   512 NORMAL  READPBLK
22:18:32.66 00.00 09C00227    1034   512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227    5007   512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227    5008   512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227    5002   512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227    5002   512 NORMAL  WRITEPBLK
22:18:32.68 00.00 09C00227    5002   512 NORMAL  WRITEPBLK
22:18:32.69 00.00 09C00227    5003  1536 NORMAL  READPBLK
22:18:32.69 00.00 09C00227    5006   512 NORMAL  READPBLK
22:18:32.69 00.00 09C00227    5010   512 NORMAL  READPBLK|EXFUNC
22:18:32.69 00.00 09C00227    5000   512 NORMAL  READPBLK|EXFUNC
22:18:32.69 00.00 09C00227       0     0 NORMAL  PACKACK|BYPASS_VALID_CHK
22:18:32.69 00.00 09C00227    5002   512 NORMAL  READPBLK|EXFUNC
22:18:32.70 00.00 09C00227    5016   512 NORMAL  READPBLK|EXFUNC
22:18:32.70 00.00 09C00227    5023  1024 NORMAL  READPBLK
22:18:32.70 00.00 09C00227    5016   512 NORMAL  WRITEPBLK|EXFUNC|DATACHECK
```

The default display will show a timestamp, the elapsed time of the request, etc. There are various qualifiers available to modify the display, like /IOSB=LONGHEX to show the real contents of the iosb, and /FUNCTION=HEX to show the function code without any translation.

Another powerful feature is to trace FDT calls to the driver (Function Decision Table), which always happen but are not easy to be seen. These requests happen in the first part of calling a driver and are used for example to validate parameters. But it can also happen that the I/O completion occurs directly from these FDT routines, so that the I/O request is normally not noticed. FDT tracing is not implemented on the VAX version of LDdriver.

The timing can be seen not only by the normal timestamps, but also by means of the SCC (System Cycle Counter), which is a hardware register in the processor architecture that can be used for accurate timing measurements. This counter is only available on Alpha and IA64, so accurate tracing is not implemented on the VAX version of LDdriver. Accurate tracing must be enabled at the time that the trace buffer is activated (LD TRACE command), this is done because activating this option may give a small performance hit. The SCC counter is specific for each processor, so in a multi processor system we may need to reschedule the completion of an I/O request to the same processor where the request was initiated to get an accurate measurement.

So if we enable FDT and accurate tracing we get this:

```
$ ld notrace lda1
$ ld trace/fdt/accurate lda1
$ dir lda1:[000000]/out=nl:
```

```
$ ld show/trace/fdt/accurate lda1
        I/O trace for device $7$LDA1:
    26-APR-2005 22:36:35.69 on node LDDRVR::

Start Time  Elaps uSecs  Pid      Lbn    Bytes  Iosb   Function
-------------------------------------------------------------------------
22:36:29.10 00.00     0 09C00227        0     0 FDT     ACCESS
22:36:29.10 00.00     0 09C00227        0     0 FDT     ACCESS | ACCESS | EXFUNC
22:36:29.10 00.00     0 09C00227        0     0 FDT     READVBLK | EXFUNC
22:36:29.10 00.00   422 00000000     5000   512 NORMAL  READPBLK | EXFUNC
22:36:29.10 00.00     0 09C00227        0     0 FDT     DEACCESS | EXFUNC
```

Here we see that the driver is called more times to do the FDT work. Notice that the real I/O request had an elapsed time of 422 microseconds, so we can measure the timing with much finer granularity this way.

Of course all trace data can be saved to disk, either as ASCII or as binary. The latter form is needed to be able to process the trace data at a later stage. A block size can be specified which will force LD to create a new version of the output file once the number of blocks have been reached. Together with a version limit this allows for a continuous form of tracing without having to worry about filling up a disk, and still being able to capture a predictable amount of data.

The trace data can also be viewed in real time with LD SHOW/TRACE/CONTINUOUS. This will read the trace data from the driver and reset the trace buffer after reading. As soon as there's new data available the driver will notify us so that we can get it. The viewer can be stopped either by Control-C or by the LD TRACE/STOP LDA1 command given from another terminal.

The default size of the trace buffer is 512 entries, but this can be as big as non-paged pool allows. The amount of bytes taken for the buffer is charged against the bytecount quota of the process issuing the command.

Watchpoints.

Watchpoints are another tool which can be a great help debugging various pieces of software. Basically, a watchpoint is a logical block on disk which will generate a special action as soon as they are hit by an I/O request. Another form of watchpoints are VIRTUAL watchpoints, these are virtual block numbers inside a file on disk. The action created when a watchpoint is hit may consist of four different flavors:

- Error
- Suspend
- Opcom
- Crash

First of all, /ACTION=ERROR. A watchpoint with this characteristic will return a predetermined error (specified by the user). Suppose we want to simulate an error on a disk, then we can do this:

```
$ ld watch lda1 123/action=error
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1     123  Error     READPBLK          02A4 (BUGCHECK)
```

As soon as logical block 123 is hit by an IO$_READPBLK request the I/O will be terminated with an SS$_BUGCHECK error:

```
$ dump lda1:/block=start=123
%DUMP-E-READERR, error reading LDA1:
-SYSTEM-F-BUGCHECK, internal consistency failure
```

The function code to trigger the watchpoint is by default an IO$_READPBLK. It is possible to specify any function with /FUNCTION=ALL, or for a unique function by using /FUNCTION=CODE=13 (for example).

The next action is SUSPEND. This means that a request will be suspended until it is released by an LD command. If you ever wonder where a request came from you can let it run into a watchpoint, and take a look with SDA what's going on. Multiple suspends will be queued, and can be released by a simple command.

```
$ ld watch lda1 10/action=suspend
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1     10  Suspend   READPBLK

$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_50 spawned
$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_9 spawned
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1     10  Suspend   READPBLK
         Suspended process: 09C00247
         Suspended process: 09C00248
```

At this time the I/O requests are suspended by the driver, and the processes are simply waiting for them to be completed. This can be done with one command:

```
$ ld watch/resume lda1
```

These I/O's will then complete, and the processes will finish.

The next action is OPCOM. This allows an OPCOM message to be created once a watchpoint is hit. The message includes the image, process id, device name, I/O function code and logical block number.

```
$ reply/enable
$ ld watch lda1 1/action=opcom
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1       1  Opcom     READPBLK
$ dump lda1:/block=(count=1,start=1)
%%%%%%%%%%%%  OPCOM  26-APR-2005 23:15:30.23  %%%%%%%%%%%%
Message from user SYSTEM on LDDRVR
***** LDdriver detected LBN watchpoint access *****
PID:     09C00227
Image:    DCL
Device:   $7$LDA1: (LDDRVR)
Function: 000C
LBN:      1
```

The OPCOM type may be combined with any other type of watchpoint.

The final type is CRASH. Is its name already suspects, this will crash the system when the selected watchpoint is hit. It's a sort of a big hammer approach in troubleshooting, but it can be handy at times. Example:

```
$ ld watch lda1 1/action=crash
%LD-F-DETECTEDERR, Detected fatal error
-SYSTEM-F-NOCMKRNL, operation requires CMKRNL privilege
```

This is shown for a reason, the lack of CMKRNL privilege. As crashing a multi user system is a very serious business, a heavy privilege is needed for this operation to succeed. The driver will check the issuing process for this privilege, and if it's not there the above error will result.

```
$ set proc/privilege=cmkrnl
$ ld watch lda1 1/action=crash
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1       1  Crash     READPBLK
$ dump lda1:/block=start=1
```

Hmmm, why is there no response here.... Right, I can see something on the console:

```
**** OpenVMS Alpha Operating System V8.2     - BUGCHECK ****
** Bugcheck code = 000008F4: RSVD_LP, Reserved for layered product use
** Crash CPU: 00    Primary CPU: 00    Active CPUs: 00000001
** Current Process =   SysDamager
** Current PSB ID = 00000001
** Image Name = $10$DKA600:[SYS0.SYSCOMMON.][SYSEXE]DUMP.EXE;1

**** Starting selective memory dump at 26-APR-2005 23:29...
```

Another thing about watchpoints is that multiple watchpoints may be created in one command:

```
$ ld watch/action=error=%x84/function=write lda1 1,2,3,4,5
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1       1  Error     WRITEPBLK       0084 (DEVOFFLINE)
 2       2  Error     WRITEPBLK       0084 (DEVOFFLINE)
 3       3  Error     WRITEPBLK       0084 (DEVOFFLINE)
```

```
 4       4  Error     WRITEPBLK           0084 (DEVOFFLINE)
 5       5  Error     WRITEPBLK           0084 (DEVOFFLINE)
```

One of my favorite watchpoints is to put a device into mountverification:

```
$ ld watch lda1 1/action=error=%x84/function=code=%x0808
$ ld watch lda1 10/action=error=%x84/function=read
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
 1          Error     PACKACK|INHERLOG     0084 (DEVOFFLINE)
 2      10  Error     READPBLK             0084 (DEVOFFLINE)
$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_253 spawned
$
%%%%%%%%%%%  OPCOM  26-APR-2005 23:44:41.37  %%%%%%%%%%%
Device $7$LDA1: (LDDRVR) is offline.
Mount verification is in progress.


$ ld nowatch lda1
$
%%%%%%%%%%%  OPCOM  26-APR-2005 23:44:50.62  %%%%%%%%%%%
Mount verification has completed for device $7$LDA1: (LDDRVR)
```

VIRTUAL watchpoints are watchpoints for a given virtual block in a file:

```
$ copy sys$system:copy.exe lda1:[000000]
$ ld watch/file=lda1:[000000]copy.exe lda1: 10
$ ld show/watch lda1
Index LBN     Action     Function        Error return code
------------------------------------------------------------------------
           $7$LDA1:[000000]COPY.EXE;1
 1      10  Error     READPBLK            02A4 (BUGCHECK)
$ dump lda1:[000000]copy.exe/block=(count=1,st=10)
%DUMP-E-READERR, error reading LDA1:[000000]COPY.EXE;1
-SYSTEM-F-BUGCHECK, internal consistency failure
```

As long as a virtual watchpoint is set, it will be impossible to dismount the disk containing the watchpoint:

```
$ dismount lda1:
%DISM-W-CANNOTDMT, LDA1: cannot be dismounted
%DISM-W-USERFILES, 1 user file open on volume
$ ld nowatch lda1/index=1
$ dismount lda1
```

Callable interface.

All commands possible with LD are also available from a user application program, as it basically does nothing more than issuing QIO requests to LDdriver. To make this possible a C header file is included in the LD kit (LDDEF.H) which contains all the necessary definitions.

As of VMS version V8.2-1 this definition file is included in the system library, so an external file is not needed anymore.

The command 'LD HELP Driver_functions' has the details of what's needed.

Internals.

At the first sight, operation of the driver seems quite straightforward, and in fact, it is. Once a drive is connected to a file there's not much more to do than getting the request and pass it in a modified form to the physical disk driver. Of course getting the mapping right and splitting the request into multiple segments if the container file is not contiguous is some extra work, but it's not that difficult. For LBN mode we just need to add an offset to the block in question, and make sure we don't step outside the capacity of the disk.

The real tricky part is to protect the innocent user from shooting itself in the foot. For example, on one node in a cluster we connect file FILE.DSK to device LDA1 with the intent to share it. Then on another node we connect the same file to LDA2. If the driver would allow this it could be a recipe for a disaster, so strict rules are in place for sharing container files. This is all coordinated by using the fork level interface of the distributed lock manager.

For every connected file two locks are taken: one for the file, which includes the volume lockname and the file identification, and one for the LD device itself. This makes it possible to verify all possible combinations for correct behavior. The lock valueblocks of the involved locks are used extensively to pass parameters between the various nodes to check everything.

One extreme case was found during development of LBN mode, where we need to check if a given range of lbn's is already in use. But how can we do this by using locks? We need to be able to ask another node if it's using a range, but there may be dozen's of ranges already there, and that will not fit in a lock value block which is used for the communication. A way around it may be to create a server process which can do the check, but it would be nicer if we could do this without it. Such a process would mean more things to check for, the error handling would be more complicated, and it would take additional resources as well.

The solution chosen is still using locks. By using one lock and putting the range of blocks we intend to use in the lockvalue block we can trigger a blocking ast routine on any node which is already using the proposed device. This blocking ast routine can then do the check, and set a go / nogo bit in the lockvalue block as a return signal. If there's minimal one node objecting to the range that we proposed then connecting a drive will fail. Very careful design was needed to use the right lock modes to prevent loops by repeating blocking ast routine calls. In the end a working model was created which is efficient, does not need an external process and is contained completely within the driver.

For LBN and REPLACE mode, a lock is used to prevent usage of the physical device by anything else than LDdriver. Since connects and disconnects may occur in any order and on multiple nodes the driver will attempt to take out the lock itself, and if that fails it will queue the lock so that if another user disconnects any LD device which was connected to this physical device we will still maintain a lock somewhere in the cluster. For example, if we are connecting on one node and we are the first one we will get the lock. Then when we connect to the same device on another node the lock will not be granted, and the request will be stalled. If we then disconnect on the first node the lock will be granted, so that we continue to be protected against other uses than LDdriver.

Final words.

LDdriver has come a long way. After many hours of midnight puzzling and thinking a lot of functions were added, thanks to a number of people in the outside world who have given me a lot of great suggestions.

As development continued I have been able to keep the VAX version of the driver pretty much in synch with the Alpha/IA64 version. It only misses accurate- and fdt tracing.

By allowing a great deal of source code transparency between Alpha and IA64 there was no need for a port to IA64. In fact, there's not even one conditional in the driver to make a distinction between the architectures. That shows what a marvelous job people in VMS engineering have done with the IA64 port.

If there are any questions left after reading this article, or if you have any suggestions it's good to know that I'm always open for improvements. And if you happen to find a bug, just let me know and I'll do anything (almost…) to fix it. Bugs in LD appear to be rare, but I'm sure there will be some lurking around. It's software after all, and I need a job too ☺.

For more information.

When the V8.0 LD kit is released it will include examples as well as source code and will be posted to the OpenVMS freeware archive at http://www.hp.com/go/openvms/freeware.

I can be contacted at lddriver@hp.com.

Jur van der Burg
HP OpenVMS sustaining Engineering