OpenVMS Technical Journal V6

Disk Partitioning on OpenVMS: LDdriver's Secrets

Disk Partitioning on OpenVMS: LDdriver's Secrets

Jur van der Burg, HP OpenVMS Sustaining Engineering

Overview

LDdriver is a device driver that runs under the OpenVMS operating system to allow creation of virtual disks. This article describes many ways to use this driver that are not widely known.

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

History

LDdriver was originally written for VAX/VMS around 1985 by Adrie Sweep, a software engineer in the Netherlands. The initial version worked well, but it was not flexible in configuration and usability. Some time later the driver was changed to use so-called cloned devices to provide a more flexible way to manage the virtual disks. Over the years a number of functions have been added, increasing LDdriver's versatility, and LD has been ported to Alpha. LD has been a part of OpenVMS since VMS V7.3-1, and it is used by CDRECORD. Beginning in V8.2, LD is fully integrated. The procedures described in this article are valid for the latest version of LD (V8.0), which will be available as a separate kit and will also be integrated into the next major OpenVMS release (V8.2-1).

What is LDdriver?

As its name implies, LDdriver is a logical disk driver that allows you to use a file on any type of hard disk as a disk. For example, if you have a file called LDDISK:[VMS]DISK.DAT, you can use that file as a disk by entering the command LD CONNECT LDDISK:[VMS]DISK.DAT LDA1:. After that you have a device LDA1: on the system which you can use as a disk.

The LD system startup procedure is SYS$STARTUP:LD$STARTUP. You can put the startup command the SYS$SYSTEM:SYSTARTUP_VMS.COM file. An optional startup parameter allows you to increase the number of LD devices by specifying the controller letter to use for LD. If the controller letter is not specified, then LDA will be used (allowing up to 9999 LD devices), but if you would like an additional controller, specify B as the first parameter to get LDB.

LDdriver operates in three different modes:

- FILE mode, which allows any arbitrary file to be used as disk.
- LBN mode, which allows you to specify blocks on a disk.
- REPLACE mode, which allows access to the disk.

These modes are explained in the following subsections. If you need any help, just remember the command LD HELP. Every command is described in detail, and a couple of examples are provided as well.

FILE Mode

The most widely-used mode of operation is FILE mode. 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.

The setup is very simple:

1. Create a file on a physical disk.
2. Connect the file to a logical disk.
3. Use it!
4. After use, disconnect it.

For 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
```

This creates 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 an OpenVMS disk that you want to burn on a CD. All you need to do is 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 in any way you like. This can also be done on a PC with a variety of PC software.

Another use of LD is sharing the LD devices across different nodes in a cluster. Notice that LD devices are not MSCP-served (for a number of reasons). If sharing is to be used, the physical disk where LD's container file resides needs to be visible on other cluster members. For example:

```
$ 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, the resulting LD device can be mounted on that node as well. For this to work, the driver imposes several checks.

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

If one of these prerequisites is not met, a specific message will follow. Suppose a device is connected on one node, and when you try to connect it on a second node you see 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
```

This means that you have an allocation class mismatch. This occurs when two nodes in a cluster have a different allocation class. Remember, the default allocation class of an LD device will be the allocation class of the node. You can change this at connect time by specifying the correct allocation class. However, you can only do this if no other LD devices for the same controller are connected.

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

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

By specifying the correct allocation class, you can do what you 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. You have to specify that you want to share the file, or the driver will reject our request. The following command will work:

```
$ 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)
```

You can use different LD devices with different allocation classes on one node. In this case, you need a new controller letter, which can be created by invoking LD's startup procedure and specifying the controller letter as the command parameter, as follows:


```
$ @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

Several 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 you only want to use the first 5000, you can do so by specifying /MAXBLOCK=5000 at connect time. You can also influence the geometry by specifying /CYLINDERS, /TRACKS, and /SECTORS. Although of limited use with modern disks, these qualifiers allow you to configure a disk to exactly mimic another disk. The geometry can be copied from an already mounted disk with /CLONE, which can also be used to create a container file that mimics a real disk, as follows:

```
$ 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 procedure will 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 a subsequent disconnect/connect will restore the same disk parameters. The geometry information can be overridden by specifying /NOAUTOGEOMETRY, in which 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 write-protected using the LD PROTECT command. You can make this protection 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 be restored.

Normally, a file is connected to an LD device by explicitly specifying which device you want to get. If you don't do this, the driver will assign a unit number and inform you about it. If you specify /SYMBOL as well, you will get 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 procedure is useful in command procedures, where the actual device name is unimportant. The symbol makes it easy to reference such a device.

LBN Mode

In LBN (Logical Block Number) mode, a physical disk can be accessed in several parts, as specified by a LBN range. Look at it as partitioning a disk without any file structure on it. For 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 procedure uses two fragments of device 1DGA1 for devices 7LDA1 and 7LDA2. The range of LBNs can be specified in either of the following ways:

- A starting LBN with a block count
- A starting LBN and an ending LBN

The driver will check for an overlap of LBNs, so any attempt to use a range already in use will result in the following 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 is the most efficient way to use the physical device, since the overhead of unused blocks is zero. The physical device can be divided in as many parts as you need, as long as the system's resources will allow them.

Another way to use this is to map a range of blocks from a foreign volume, for example a UNIX file system. Use your imagination!

As with FILE mode, these LD devices can 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 as well, of course. Notice that the range of LBNs in use is checked on all the nodes in the cluster that 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 information about this check, see Internals.

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 with LBN, the physical device will not be available for any other use in the cluster; that is, a mount on any node will fail until the last LD device disconnects. For example:

```
$ 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 to great lengths to protect the user against any errors.

REPLACE Mode

REPLACE mode is not a form of partitioning, but a way to create access to the physical device using LDdriver. You will understand how to use this when you read about I/O Tracing and Watchpoints, which describe how to perform a real-time trace of all I/O requests.

The following example creates a device (7LDA1) that will direct all I/O to the physical device 1DGA1:

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

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 by means of a lock.

I/O Tracing

One of the most powerful and little known 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 shows a timestamp, the elapsed time of the request, and so forth. Various qualifiers allow you to modify the display, such as /IOSB=LONGHEX to show the real contents of the IOSB, and /FUNCTION=HEX to show the function code without any translation.

Another powerful function is to trace FDT (Function Decision Table) calls to the driver. These always happen but are not easy to see. These requests happen in the first part of calling a driver and are used, for example, to validate parameters. But the I/O completion can occur 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 measured 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 (using the LD TRACE command); it is not the default because activating this option may impose a small performance hit. The SCC counter is specific for each processor; therefore, in a multiprocessor system, the driver 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.

If you enable FDT and accurate tracing, you 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
```

Notice that the driver is called more times to do the FDT work. The real I/O request had an elapsed time of 422 microseconds; you 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 data. You need binary data to process the trace data later. You can specify a block size, 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 provides continuous tracing without having to worry about filling up a disk, while still capturing a predictable amount of data.

The trace data can also be viewed in real time with LD SHOW/TRACE/CONTINUOUS. This command reads the trace data from the driver and resets the trace buffer after reading it. As soon as there is new data available, the driver notifies LD driver so that you can get it. The viewer can be stopped either by Control-C or by the LD TRACE/STOP LDA1 command issued 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 byte count quota of the process issuing the command.

Watchpoints

Watchpoints provide another tool that can help you debug various pieces of software. A watchpoint is a logical block on disk that will generate a special action as soon as it is hit by an I/O request. Another form of watchpoints is VIRTUAL watchpoints - virtual block numbers inside a file on disk. You can choose any of the following actions to occur when a watchpoint is hit:

- Error

  A watchpoint with the Error characteristic will return a predetermined error (specified by the user). If you want to simulate an error on a disk, you 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. You can specify any function with /FUNCTION=ALL, or for a unique function by specifying, for example, /FUNCTION=CODE=13.

- Suspend

  The SUSPEND action 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 use SDA to see what is 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
```

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/Os will then complete, and the processes will finish.

- OPCOM

  The OPCOM action allows an OPCOM message to be created when a watchpoint is hit. The message includes the image name, 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.

- Crash

  As its name implies, this action will cause the system to crash when the selected watchpoint is hit. It is a "big hammer" approach for troubleshooting, but it can be handy at times. For example:

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

  This example shows what happens when you do not have the CMKRNL privilege. As crashing a multiuser system is very serious business, a high privilege level is required for this operation to succeed. The driver checks the issuing process for this privilege, and if it is 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
```

The privilege is enabled, but there is no response here. Fortunately, you 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...
```

Note that multiple watchpoints can be created using 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)
```

A useful watchpoints is when a device is put into mount verification:

```
$ 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 the LD commands are also available from a user application program by issuing QIO requests to LDdriver. To make this possible, a C header file (LDDEF.H) that contains all the necessary definitions is included in the LD driver kit. As of OpenVMS version V8.2-1, this definition file will be included in the system library, so an external file is not needed anymore.

The command LD HELP Driver_functions provides the details you need to implement the callable interface.

Internals

At first sight, operation of the driver seems quite straightforward, and in fact, it is. Once a drive is connected to a file there is not much more to do than getting the request and passing 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 simply needed to add an offset to the block in question, and make sure we didn't step outside the capacity of the disk.

The tricky part is to protect innocent users from shooting themselves in the foot. For example, on one node in a cluster you connect file FILE.DSK to device LDA1 with the intent to share it. Then on another node you connect the same file to LDA2. If the driver allowed 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 lock name and the file identification) and one for the LD device itself. This makes it possible to verify all possible combinations for the correct behavior. The lock value blocks 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 needed to determine whether a given range of LBNs was already in use. How could we do this using locks? We needed to be able to ask another node if it is using a range, but there may be dozens of ranges already there, and that will not fit in a lock value block that is used for the communication. A way around it might have been to create a server process that could do the check, but it we knew 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.

Our solution still uses locks. By using one lock and putting the range of blocks we intend to use in the lock value block we can trigger a blocking AST routine on any node that is already using the proposed device. This blocking AST routine can then do the check, and set a go/nogo bit in the lock value block as a return signal. If even just one node objects to the range that we proposed, connecting a drive will fail. Very careful design was required in order to use the right lock modes to prevent loops by repeating blocking AST routine calls. Finally a working model was created that 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 the physical device from being used by anything other 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. If that fails, it will queue the lock so that if another user disconnects, any LD device that was connected to this physical device 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.

Conclusion

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

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

By allowing a great deal of source code transparency between Alpha and I64, there was no need for a special port to IA64. In fact, there is not even one conditional in the driver to make a distinction between the architectures. This shows what a marvelous job the people in OpenVMS engineering have done with the I64 port.

For more information

If you have questions after reading this article, or if you have any suggestions, we are always interested in possible improvements. If you happen to find a bug, just let us know and we will do just about anything to fix it.

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.

The author can be contacted at lddriver@hp.com.