A Portable Tape Driver Architecture

Curtis Anderson
TurboLinux
2000 Sierra Point Parkway, 4th Floor
Brisbane CA 94005
canderson@turbolinux.com
tel +1 650 244-7777
fax +1 650 244-7766

Abstract
This paper describes a new architecture for device drivers for tape drives attached to UNIX-like systems. The design goals are presented, some current architectures are measured against the requirements, and a new architecture is described along with its advantages and disadvantages. This architecture was developed, and this paper was written, while the author was at Silicon Graphics. The work continues at SGI for IRIX, SGI's variant of UNIX, and for Linux. The resulting Linux software will be licensed as Open Source

1 Introduction

In order to evaluate the architecture presented later, and the design decisions that went into it, the environment and assumptions that existed during the design must be presented as well.

1.1 What Do Tape Drivers Do?

The fundamental purpose of a tape driver is fairly simple; it provides access to the tape drive hardware for use by application programs. Beyond that basic purpose, however, there are secondary characteristics that are controlled by the environment that the driver is running in and by application program expectations.

For example, tape drivers for a UNIX-like environment will attempt to both insulate the application from some of the device specific details of controlling the drive hardware and to homogenize the operational interfaces for different types of drives. One example of the latter is the need to have all drive types end up on the same side of a file mark when reading through a tape, in spite of what the drive's firmware does naturally. A critical result of this is the application having the ability to predict where the tape is going to physically end up after each operation, i.e.: before the tape mark or after it.

Other secondary characteristics include the failure modes of the driver and the failure domain. The failure modes describe how the driver can fail; e.g.: hang, crash, returns an error, silent failure, etc. The failure domain describes how much impact the failure modes have; e.g.: does the current operation fail, the application hang, or the system crash on each type of failure. Some combinations are clearly more desirable than others.

1.2 Traditional Tape Driver Architectures

The normal architecture for a tape driver in a UNIX-like system is an event driven state machine built into the operating system kernel. For most UNIX-like systems building it

into the kernel is required in order for the driver to have access to the hardware control registers for the connection to the drive. That connection will usually be a SCSI bus but FibreChannel attached tape drives are now starting to appear on the market.

The driver must be able to support multiple devices simultaneously. As a result of the interrupt-driven nature of modern I/O busses, it must also be able to coordinate an asynchronous thread of execution with a synchronous one for all such devices simultaneously. In a multiprocessor system it must even be able to protect itself against race conditions between the processors, in both normal code and interrupt level code. The interrupt level code can have impacts on the rest of the kernel's ability to respond to real-time events, and the kernel's scheduler can have an impact on the driver's ability to keep the drive streaming.

The kernel of a UNIX-like system is usually a fairly hostile place to program. The interfaces to supporting code in the rest of the kernel are complex and the dependencies are delicate, and those interfaces and dependencies can change at every release of the kernel. In addition, the debugging tools are usually very primitive.

The driver may also attempt to support in a single monolithic driver source file many, if not all, of the devices the system vendor wants to be able to control. This may lead to either table-driven code or inscrutably complex run-time checking of the drive type. In either case there is the risk that any change to the code requires careful attention and regression testing to ensure no breakage in support for any of the supported devices.

The opposite structure, a separate driver source file for each tape drive make/model to be supported, is used by some system vendors. This option avoids some of the pitfalls of the monolithic driver, but it leads to a risk of different semantics from one tape make/model to the next, and still has a requirement to modify and retest all the supported drivers if the interface to a kernel support routine changes.

The sum of all these forces acting on the design of tape drivers has brought us to a point where a tape driver is a complex and delicate piece of code where any bug includes the risk of impacting the entire computer system.

2 Architectural Requirements

In order to decide what, if anything, is wrong with a traditionally structured tape driver, we need a list of those characteristics that we believe are important to the architecture of a tape driver. Debating which characteristics belong in this list, which do not, and their order of importance is in itself an interesting topic. Here is the list of requirements used for this work, shown in their order of importance:

2.1 Failures Must Be Contained

Failures in a tape driver must be isolated to that one drive, i.e.: it cannot be allowed to crash the system. Limited forms of service interruption are acceptable; for example a bug might affect the use of one device.

2.2 All Drivers Must Provide The Same Operational Semantics

Any tape driver must provide application programs with the same operational semantics as all other drivers. For example, they must all end up on the same side of the file mark when reading a tape.

An application should be able to reliably predict the behavior of the tape drive no matter who wrote the driver for it. An application should also be able to reliably predict the behavior of the tape drive, with some caveats, no matter which drive type it is. The former ensures that all DLT7000's operate the same, while the latter ensures that (to the extent that it is physically possible) a simplistic application doesn't need to know if it running against an AIT-2 or a DLT7000.

2.3 Drivers Must Be Portable To Multiple Operating System Platforms

Portability of an application that uses tapes to a new operating system can be greatly hindered by a difference in tape access semantics. The best way to avoid such differences is to use the same driver on all of them. Any new driver for a given make/model of tape drive must be source code portable to multiple kernels.

2.4 Distributed Development Of Drivers

The model used in the PC marketplace of bundling the driver software with the drive hardware is a good one. The people who make the drive are the ones in the best position to be able to make that drive perform correctly and reliably. This implies that it must be possible for the drive vendor to be able to build a driver for an operating system without reference to proprietary information from the platform vendor. It must be possible for multiple organizations to develop drivers for different devices in parallel.

2.5 High Performance

Any new tape driver architecture cannot sacrifice the performance of the drive or impose a significantly higher CPU load than a traditional architecture.

2.6 Isolating Support Of A Device From Other Devices

A monolithic tape driver implementing all supported tape drives will become unwieldy as the number of tape drive make/models being supported grows. Regression testing each driver change against all supported devices quickly becomes the dominating factor in the cost of adding support for a new drive. Using separate driver source files for each supported make/model of tape drive avoids this pitfall.

2.7 Differing Levels Of Investment For Each Drive Type

It must be possible in a tape driver architecture for one driver implementer to provide the basic set of operations and error recovery mechanisms while another provides that plus additional error recovery and/or additional device dependent operations. Any new architecture cannot raise the minimum requirements for implementing a tape driver too high, nor can it disallow extensions to take advantage of drive-specific features. Note that a driver providing additional operations risks requirement 2.2 unless it is a pure superset.

3 Evaluating The Traditional Architectures Against The Goals

Before going to the trouble of designing a new tape driver architecture, we need to decide if the above mentioned "traditional" approaches to driver structure are lacking significantly enough to warrant the effort.

The two approaches we've been discussing are both fully inside the kernel so they both will impact the entire computing system if they encounter a severe bug. There is no inherent difference between these two models in terms of their portability across operating systems (they are not) or in terms of their performance.

To successfully create a high performance, highly reliable driver in the hostile environment of an operating system kernel, the implementer must have a great deal of detailed knowledge about the particular kernel they are targeting and be able to use some fairly primitive debugging tools. Both of those requirements imply that writing a driver requires a talented operating systems engineer. It is desirable to eliminate those requirements in favor of allowing an engineer with less specialized experience perform the task.

3.1 Monolithic Drivers

A monolithic driver implementing all supported tape drives easily provides common operational semantics across all drive make/models. It will become unwieldy as the number of tape drive make/models being supported grows, however. Regression testing each driver change against all supported devices quickly becomes the dominating factor in the cost of adding support for a new drive. Changes to the kernel support interfaces that the driver uses also necessitate a full regression test against all supported devices.

A monolithic driver has great difficulty isolating one drive type from another, and is in practice only modifiable by one person at a time, in a serial fashion of implementation then testing. It is also difficult to provide extended error recovery for one drive type while isolating that recovery code from the other drive types.

The monolithic model has significant problems meeting the requirements set out above.

3.2 Separate Driver Source Files

Using separate driver source files for each make/model of drive has a significant risk of allowing variances in the semantics provided by one drive versus another. There is no structural help in the architecture or development model to ensure this, it is just a matter of all the programmers knowing that they have to do the same thing.

Using separate source files for each make/model of drive is quite good at isolating one drive type from another and it lends itself quite well to being worked on by more than one person at a time. Adding extended error recovery code to the driver for one particular type of drive is straightforward and has no impact on the other drive types.

It has the disadvantage of locking the system vendor into providing a static set of support interfaces in their kernel for the tape drivers to use. Those interfaces can become

inefficient and/or difficult to implement as the system vendor makes changes in the structure of the operating system kernel.

Overall, the separate driver source files model is better than the monolithic driver model, but it still has the critical problem of failure containment as well as some significant application portability risks.

4 New Architecture

The architecture being proposed is composed of a document, two main software components, and a well-defined interface between those components:

- *Tape Access Semantics Document*: specifies the behavior that an application can expect from the tape driver, e.g. which side of the file mark the tape ends up on after a read operation.

- *Tape Support Driver (TSD)*: a piece of code that lives inside the operating system kernel and is uniquely optimized for that kernel but is common to all tape drives supported by that operating system.

- *Personality Daemon*: a piece of user level code that is uniquely optimized for a given make/model of tape drive but is common to all operating systems that support that drive.

- *Personality Interface*: the interface between Tape Support Drivers and Personality Daemons. This is the piece that allows portability.

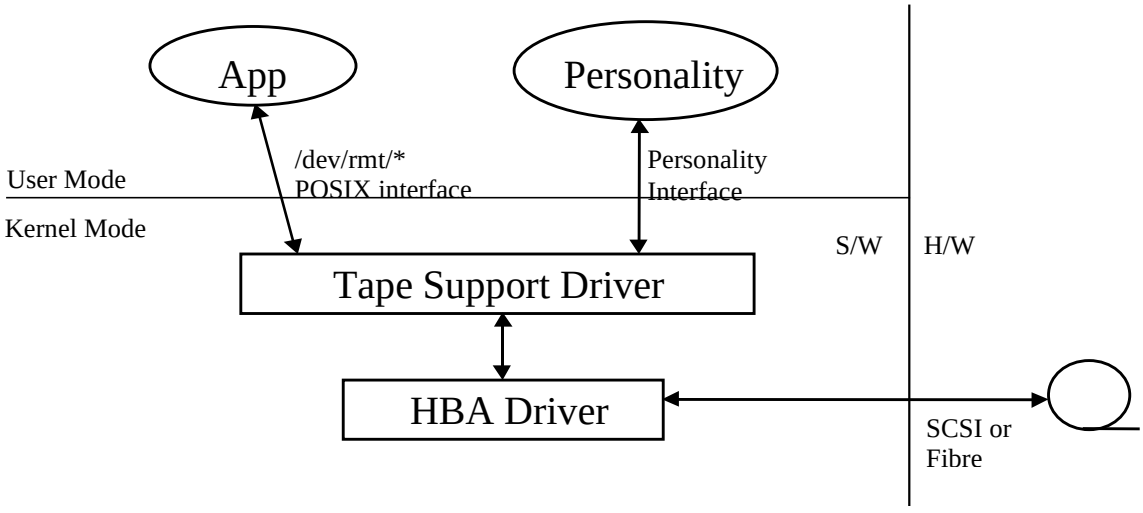

Figure 1. Structure of the new tape driver architecture.

This new architecture will be implemented on IRIX, SGI's variant of UNIX, and on Linux. The source code for the Linux port will be distributed under Open Source licensing terms. It is one of the key characteristics of this architecture that a site be able

to modify or fix an existing Personality Daemon to meet their needs without the involvement of the operating systems vendor.

4.1 Tape Access Semantics Document

One of a tape driver's most important characteristics is the application's ability to predict what the tape drive is physically going to do for each operation the application asks the driver to perform. For example, which side of a tape mark will the drive end up on after a read command runs into a tape mark.

Given the requirement for independently written Personality Daemons, there must be a document that accurately and completely describes the semantics that can be relied upon by application developers and that therefore must be provided by Personality Daemon developers.

In practice, a conformance test suite must also be written. It will need to exercise all of the operations defined in the semantics document in order to verify that a given Personality Daemon correctly implements those operations. The test must check the handling of tape marks, end-of-data, end-of-tape, beginning-of-tape, file-space-forward, short reads, long reads, etc.

The semantics that the document describes, and that the Personality Daemons implement, is not part of this paper. This new architecture is independent of the particulars of the tape access semantics being implemented. In fact, in the Future Work section of this document we talk about the possibility of there being different documents describing different sets of semantics, each matching a de facto industry norm. Examples include Solaris, IRIX, AIX, and HP/UX. A site could have several Personality Daemons available for each drive, one for each common set of tape access semantics. The use and management of multiple Personality Daemons per drive is, again, not part of this paper.

4.2 Tape Support Driver

In order to gain access to the SCSI or FibreChannel controller, and as a practical requirement of getting high performance, we have defined a component inside the kernel. We call that piece the *Tape Support Driver (TSD)*. It is unique to each operating system kernel but is common to all drives supported by that kernel.

The TSD is basically just a data pump. It is highly optimized kernel code that has all the usual dependencies, interrupt level code, multiprocessor locking, etc., and is probably written by a senior operating systems engineer. The implementation will be unique to each operating system and will take advantage of all the optimizations that are available in that environment. The TSD supports the read() and write() system calls from the application as well as the newly defined *Personality Interface*. The TSD does not do anything other than read/write and the Personality Interface, it depends on a Personality Daemon for support of all other operations and for error handling.

There is an exception to the policy that the TSD does no handling of errors. A read() operation where the data transferred is less than the data in the tape block will result in an

Illegal Length Indicator error being reported by the drive. This would appear to be an error to the TSD and would normally be bounced up to the Personality Daemon for processing. This is not considered an error under some circumstances, and we cannot afford to involve the Personality Daemon on every read() operation if we want to maintain streaming, so the architecture allows the TSD to have some parameters and for the Personality Daemon to control them. The number of parameters must be kept to a minimum in order to keep the TSD simple, but they will undoubtedly have to exist.

4.2.1 External Interfaces To The Tape Support Driver

The TSD must support two external interfaces. The first interface is the POSIX compliant tape-access interface that an application uses to control the tape; e.g.: /dev/rmt/tps0d4nrv. The second is the Personality Interface to a Personality Daemon.

Some operating systems layer disk and tape drivers on top of drivers for the specific Host Bus Adapter (HBA) cards, and some provide more formal interfaces to kernel support routines that a driver can make use of. Both of those are examples of interfaces that are not visible in this architecture, they are implementation dependent.

The Personality Interface for a drive is only available to a Personality Daemon, and only one Personality Daemon can be associated with a given physical drive at a time.

4.2.2 Error Injection In The Tape Support Driver

Murphy assures us that the error recovery code in a Personality Daemon will not work correctly unless it has been tested. Making a real drive fail in exactly the required way at the required time in order to test that code is difficult at best. Therefore, it is desirable to be able to inject errors into the operation stream from software. Software based error injection may not be quite the same as if a physical drive were actually failing, but the increased flexibility of testing and code coverage would more than make up for the difference in the nuances.

The error injection is probably best done by a thin layer below the TSD itself. It could watch the sequence of operations as the TSD and Personality Daemon interact with the drive. When the desired position, time sequence, or pattern was found, the layer would return an error rather than the real answer. The patterns, sequences, and resulting error codes should be programmable by a user-level utility program; the level of programmability of the layer would undoubtedly be increased over time.

This layer will be needed in the long run, but is not required for initial support of the new tape driver architecture.

4.3 Personality Daemons

The second major component of the new architecture is called a *Personality Daemon*. It is unique to a given make/model of tape drive but is common to all operating systems.

The Personality Daemon is a piece of user level code that makes use of the Personality Interface to talk to the Tape Support Driver in the kernel. It provides operational control

and semantics for that tape drive. We take it as a base assumption that performance of the drive is less important when recovering from an error or processing a special request from the application (for example, writing a file mark), than when it is doing reads or writes.

There will be a unique implementation of a Personality Daemon for each make and model of tape drive. Making a change to one Personality Daemon, or writing a new one, cannot introduce bugs into another Personality or change its semantics. This allows multiple people or organizations to develop Personality Daemons simultaneously. It also eliminates the need to regression test a Personality until it actually changes.

Having separate Personality Daemons for each drive type means that writing a Personality Daemon for a drive that is fairly self-sufficient is easier that writing one for a drive that requires more hand-holding. The flip side of that coin is also true, that it is possible to invest in a Personality Daemon for one drive more than for another, gaining better error logging or recovery or other operational advantages beyond simply pushing data to and from the drive.

The biggest drawback to having separate Personality Daemons is that it is more difficult to ensure that they are all supporting the same semantics (e.g.: does the tape end up before or after the file mark) for the application program. A conformance test suite will be required in order to test Personality Daemons. Fortunately, the set of operations and their expected outcomes has already been clearly defined by the *Tape Access Semantics Document* and as part of the definition of the *Personality Interface*.

4.3.1 Personality Daemon Failure Containment

A single Personality Daemon is the unit of failure in this architecture. Since only the most primitive of device support is inside the operating system kernel, and the complexity and potential for bugs comes mostly from error processing and recovery code, most failures should be contained to a single user-level process. That process can be restarted if it fails or hangs, thereby regaining correct operational control of the drive without impacting the system as a whole. Only the application that was accessing the drive would be impacted.

4.3.2 Personality Daemon Responsibilities

A Personality Daemon is responsible for processing both control commands from the application and for handling errors generated by the drive.

When an application issues any control operation on the drive like a rewind or a seek-to-end-of-data, any operation other than a read or a write in fact, the TSD will simply forward that request to the Personality Daemon. The daemon is then responsible for building the correct SCSI command block(s) for the particular drive it is controlling and using the Personality Interface to send those commands to the drive. In this way it interacts with the drive to accomplish the operation that the application wanted. This sounds very indirect, but this mechanism allows for Personality Daemons to work around variations in the native behavior of one drive type versus another in order to accomplish

the goal. In short, this is how we can homogenize the native behavior of drives into the standard semantic model that applications want to depend on. When the daemon has finished its processing of the control operation, it tells the TSD to resume the application with a successful return code.

The Tape Support Driver will also pass virtually any error condition reported by the drive up to the Personality Daemon for processing. This allows the daemon to do device-dependent error diagnosis and recovery operations. The daemon will again use the SCSI pass-through capability of the Personality Interface to interact with the drive before returning control to the application.

4.3.3 Personality Daemon Programming Model

There will be one running instance of a Personality Daemon for each physical drive attached to a system. This allows the programming model inside a Personality Daemon to be single-threaded and fully synchronous. The Personality Daemon will wait on interactions with the Tape Support Driver through the only interface it has to support, the Personality Interface, which is a straightforward event-response style interface. The Personality Daemon does not need to worry about coordinating interrupt level code with non-interrupt level code, multiprocessor locking issues, or asynchronous operations. The reduction in code complexity is one of the significant advantages of the new architecture.

The fact that there is one Personality Daemon per physical drive also implies that the failure scenarios only involve one drive at a time and do not impact the rest of the system. Barring a bug in the Tape Support Driver or a case where the TSD does not adequately protect itself from bad input from the Personality Daemon, the rest of the system will not be impacted if a Personality Daemon crashes or hangs. In fact, the Personality Daemon can simply be killed and restarted to recover from a hang.

The single-threaded, synchronous, user level process programming model allows the use of powerful debuggers and a much more elaborate testing environment. Adding this to the reduced complexity of the code should result in a much higher level of reliability for the driver overall.

4.4 Personality Interface

The *Personality Interface* is common to all Tape Support Drivers and Personality Daemons and defines the relationship between them.

The Personality Interface is used by the Tape Support Driver to tell the Personality Daemon about actions requested by the application, for example: rewind, file-space-forward, and write-a-file-mark. It is also used by the TSD to tell the Personality Daemon about exceptions generated by the tape drive, for example: file mark found, early warning for end-of-tape, read failure, etc. The TSD expects the Personality Daemon to handle those conditions as it sees fit.

The Personality Interface is used by the Personality Daemon to receive notification from the TSD of either application-requested actions or tape drive generated errors. It is also

used to directly interact with the tape drive via a mechanism to send/receive arbitrary SCSI commands, commonly called a pass-through driver, and to control the error indications going back to the application.

The Personality Interface is synchronous in both directions. When the application requests a rewind, for example, the TSD will store the details of who is doing what and will use the Personality Interface to wake up the Personality Daemon. Note that the application will be blocked in the TSD at this point. The Personality Daemon will use the Personality Interface to query the TSD for status. It will then generate the SCSI "rewind media" command and will call the Personality Interface in order to send it down to the TSD for execution. When that has finished and returned to the Personality Daemon, it will call back into the TSD with a command telling the TSD to resume the application with a specific return code.

As of the writing of this paper in December of 1999, SGI is now in the implementation phase of the project. The particulars of the Personality Interface will not be finalized until a few Personality Daemons have been written and the Personality Interface's generality has been verified, but listed below is the structure of the interface as it is currently defined.

4.4.1 Interface Initialization

When a Personality Daemon first starts up it needs to set basic parameters for use by the TSD. It must also ensure that the TSD and itself are both using the same version of the Personality Interface and that the drive to be controlled is of the correct type for the Personality Daemon. The *TSD_INIT* ioctl() is used to initialize the interface.

Some of the parameters that are configured when the interface is initialized include: basic device timeouts, whether the drive supports reads followed by writes without an intervening tape positioning command (or writes followed by reads), whether some device-reported error conditions can be ignored by the TSD (e.g.: short length reads), and the list of ioctl() operations that the Personality Daemon supports.

4.4.2 Sleeping And The Use Of Signals

The Personality Daemon will sleep during the time that it is not actively servicing the TSD, waiting for a signal to arrive. The TSD will send the Personality Daemon a *SIGUSR1* signal when it needs help with something.

Once the Personality Daemon has been broken out of its sleep, it will use the *TSD_QUERY* ioctl() to determine the basic situation and will then use the ioctl() operations defined in the following sections to interact with the drive and the TSD.

4.4.3 Type Of Service Required

The TSD_QUERY ioctl() returns the reason for the latest signal from the TSD and the details behind that signal. The possible reasons include an application doing an open(), close(), or ioctl() system call, an error reported by the drive, and a read or write operation on the drive. The structure returned by the TSD_QUERY ioctl() includes all of the fields

required to give the details appropriate to all of the reasons for the Personality Daemon being involved. Combining the various fields into one structure was seen as better than defining a set of query operations, one for each type of service that the TSD requires of the Personality Daemon.

For all operations, the structure includes the process ID of the application. For open()'s, the structure also shows the flags associated with the open() call (rewind-on-close, density, compression, etc). For close()'s, no additional fields are required. For ioctl()'s, the structure contains the ioctl() command code and the ioctl() argument if it is not a pointer to a data structure (see the *TSD_COPYIN* request). For errors reported by the drive, the structure contains the SCSI sense code information, the requested I/O size, and the residual un-transferred byte count. For I/O operations, the structure includes the type of operation and the transfer counts. The structure also contains some statistical counts such as total bytes read/written, total read/write operations performed, current block number on tape, etc.

4.4.4 Application Request Processing

The application will be suspended and the Personality Daemon woken up on all open(), close(), and ioctl() operations and on some read() or write() operations.

The Personality Daemon needs to be involved every time an application opens a tape drive so that it can ensure the drive is ready for the application, validate access modes, etc. The Personality Daemon needs to be involved on every close operation as well so that it can clear and/or set status flags appropriately and to issue the rewind operation for the rewind-on-close semantics that some applications depend on. All control operations that the application performs (e.g.: rewind, write file-mark, etc) will come into the kernel via the ioctl() system call.

When an application issues an ioctl() operation on a tape, it provides the operating system kernel with an operation code and a pointer to a memory buffer. The size and contents of that memory buffer are operation dependent. We do not want to provide direct access from the Personality Daemon into the application's address space, so the contents of the memory buffer must be copied into a buffer inside the TSD. The TSD can then make the contents of that kernel buffer available to the Personality Daemon. The Personality Daemon needs to tell the TSD at initialization time which operation codes it supports, the associated amount of data to be copied in to or out of the kernel for that operation, and whether super-user privileges are requires to perform that operation..

The Personality Daemon will make use of two ioctl() operations when processing ioctl() requests from the application: *TSD_COPYIN* and *TSD_COPYOUT*. TSD_COPYIN returns from the TSD the ioctl() operation code and associated data that the application passed to the TSD. TSD_COPYOUT sends to the TSD the bytes to be copied out to the application as the results of the application's ioctl() operation.

The return code for the ioctl() call the application made comes via a separate ioctl() operation, *TSD_RESUME*, that the Personality Daemon uses. It implies that the TSD should unblock the application and allow it to continue processing.

4.4.5 Support For Device-Dependent Ioctl() Operations

We do not want to artificially limit the ioctl() operations that a Personality Daemon can support. There are many drives that provide unique features that an application might want to use if it was willing to include drive-type-dependent code. The additional operation codes will simply be listed as part of the table of supported ioctl() codes and buffer sizes that is passed into the TSD by the TSD_INIT ioctl().

4.4.6 Drive Error Processing

The TSD_QUERY ioctl() returns to the Personality Daemon essentially all of the information that is needed to start processing the error from the drive. The Personality Daemon will do an initial diagnosis of the problem based on the SCSI sense code information that came back from the last TSD interaction with the drive. It may use the SCSI pass-through support described below to interact with the drive, doing additional error analysis and/or error recovery operations.

When the Personality Daemon has finished all of the processing that it wants to do for the reported error, in addition to its ability to return an error to the application, an option to the TSD_RESUME ioctl() allows the Personality Daemon to ask the TSD to retry the original operation.

4.4.7 I/O Notification Processing

The Personality Daemon needs the ability to tell the TSD to involve it just prior to, or just after, the next read or write SCSI command is issued to the drive.

For example, if the application has opened the drive in fixed-block mode and not set the block size to be used, the tape driver should use the block size that the tape was written with for all subsequent read operations. In order to determine what block size to use, the Personality Daemon will need to get involved just prior to sending the first SCSI read command to the drive.

Some tape drives will not report that the cartridge has physically been marked read-only until some time after the first write operation when the data in the on-drive buffer is actually flushed to the tape. The Personality Daemon for such a drive will arrange to get involved after the first write to a cartridge completes. It can issue a command to force the on-drive buffer to tape, thereby checking the read-only status of the cartridge at a point where the Personality Daemon can still return an error code for application's first write operation indicating that the cartridge is in fact read-only.

This capability can also be used by the Personality Daemon to govern all access to the tape by the application if necessary. After some types of serious I/O errors, further reads or writes to the drive must be disallowed. The Personality Daemon can intercept all I/O operations before they happen and return an error to the application.

The TSD_RESUME ioctl() will include flags that tell the TSD to involve the Personality Daemon just prior to, or just after, the next read or write SCSI command.

4.4.8 SCSI Passthrough Support

When the Personality Daemon wants to send a SCSI command to the drive, it needs to provide to the TSD the following:

- The bytes comprising SCSI command to be sent.
- A pointer to a data buffer used for output to the drive or for input from the drive.
- Flags, including whether the drive will expect to transfer data to or from the host.
- The maximum number of seconds to wait for the command to complete.
- A pointer to a buffer for the SCSI sense code information if the command fails.

If the command was successful, the Personality Daemon can expect that the TSD has filled the data buffer with the results from the drive and has returned the number of valid bytes in that buffer. If the command was not successful, the Personality Daemon can expect that the TSD has filled in the status reported by the HBA, the status reported by the drive (if any), and the SCSI sense code information (if any). The HBA status will include errors such as "parity error on the bus" which will render the other status information invalid.

The *TSD_SEND* ioctl() is the operation used by the Personality Daemon to send SCSI commands to the drive. With one exception, that ioctl() will not return to the Personality Daemon until the SCSI command has either successfully completed, the command has failed and error status has been obtained, or the command has timed out.

In order to support operations such as the ability of an application to continue processing while a tape rewind is in progress, the flags field of the TSD_SEND ioctl() will tell the TSD whether to wait for the SCSI command to finish or to return to the Personality Daemon immediately. The Personality Daemon can then use the TSD_RESUME ioctl() to allow the application to continue processing.

If an application is resumed while a long-running operation is in progress, the Personality Daemon is responsible for managing the application's access to the drive. For example, it can use the flags on the TSD_RESUME ioctl() to intercept all I/O operations from the application before they are sent to the drive, then use the TSD_SEND ioctl() to verify that the drive has finished the long-running operation before retrying the application's I/O operation. An alternative approach relies upon the fact that while the drive is busy rewinding, it will report a "busy" status. All I/O operations that the application issues will generate an "error" that will then involve the Personality Daemon. In either case of the Personality Daemon gaining control, it should probably just sleep for a while and then retry the operation.

4.4.9 Block Size Control

The Personality Daemon needs to be able to control the size of the read and write operations that the TSD is passing from the application to the drive. The *TSD_BLOCKSIZE* ioctl() tells the TSD the minimum and maximum block sizes that the drive can accept, and if the drive is operating in fixed-block mode, the current block size to use. This information may change during an application's use of a drive as a result of the application asking to change the block size being used for fixed-block mode access.

4.4.10 Stopping In-Progress Operations

The Personality Daemon needs the ability to abort a long-running operation that might be in progress on the drive. The *TSD_ABORT* ioctl() asks the TSD to do just that. Under certain circumstances, the Personality Daemon needs to be able to get control of the drive again after issuing long-running operations such as a rewind.

4.4.11 Personality Interface Summary

SIGUSR1 – Signal when the TSD needs help from the Personality Daemon
TSD_INIT – Initialize the Personality Interface
TSD_QUERY – Show the reason the Personality Daemon needs to get involved
TSD_COPYIN – Copy the application's ioctl() info into the Personality Daemon
TSD_COPYOUT – Copy the Personality Daemon's response to the application's ioctl()
TSD_SEND – Pass a SCSI command through the TSD to the drive
TSD_RESUME – Resume the application or retry the operation that failed
TSD_BLOCKSIZE – Set the allowable block sizes for read() and write() calls
TSD_ABORT – Ask the TSD to abort any in-progress operations with the drive

4.5 Example: Processing A Drive Exception

A critical design issue in the new architecture is how to handle errors reported by the drive. The tools that the Personality Daemon has available to it to handle drive errors have already been described, but walking through the sequence of events in a representative example would be illustrative.

Assume that the application is reading data from the drive and the drive runs into a media defect that has obliterated some of the data.

1. For initial state we assume that the Personality Daemon is sleeping waiting for a Personality Interface signal that the Tape Support Daemon (TSD) needs help. We also assume that the application has issued a read() system call and is blocked waiting for the results.

2. The TSD gets the read request from the application and issues a "read" SCSI command to the drive. The drive encounters a problem and responds to the host with a "check condition" (a SCSI message indicating a problem with the command).

3. The TSD then uses a "request sense" SCSI command to get more information from the drive on what type of error happened and uses the Personality Interface to signal the Personality Daemon that something needs attention.

4. The Personality Daemon uses the Personality Interface to get the sense code information that the drive returned as well as the current status of the application and any other context information it needs such as block counts, operation counts, etc.

5. Based on the sense code information, the Personality Daemon decides to run some diagnostics on the drive. It builds a SCSI command block for the command it wants to send to the drive and calls into the TSD. The Personality Daemon is then blocked waiting for the call to return from the TSD.

6. The TSD sends the SCSI command block to the drive and either collects any resulting output or uses a "request sense" command to collect any error (sense) codes. It allows the call from the Personality Daemon to return with whatever it has collected.

7. The Personality Daemon analyses the results from the TSD and decides that it should log the error, fail the operation, and return an error to the application. The error is logged via the normal SYSLOG facility from the Personality Daemon.

8. The Personality Daemon calls into the TSD asking for the application to be resumed with an "EIO" error code being the return value from the read() system call.

9. The call returns from the TSD into the Personality Daemon and it goes back to sleep waiting for the next signal from the TSD that something needs to be done.

Common variations on the above sequence would include the Personality Daemon issuing more diagnostic and/or error recovery SCSI commands to the drive, doing more detailed error logging or using different modes of notification (e.g. pager or email), interacting with any system management framework that might be desirable, and possibly retrying the operation that failed.

5 Future Work

It is possible to write a range of Personality Daemons for a given operating system providing different legacy-based semantics for the same drive. Since a Personality Daemon can be stopped and another one started for a given drive, it is possible for a Media Management System such as IEEE 1244 to offer different sets of semantics to an application and let the application choose at run-time which it wants to use.

6 Conclusions

Under this new architecture, the operating system vendor's job is to write a Tape Support Driver that can control their HBA, interact gracefully with the rest of their kernel, and implement the Personality Interface. Having done so, then they benefit from the available Personality Daemons. The drive vendor's job (or whoever provides a Personality Daemon) is to accurately control the drive and to conform to the Personality Interface. Having done so, then their drive will be supported on a wide range of operating systems.