# Removable Media in Solaris

*Howard Alt* – SunSoft, Incorporated

## ABSTRACT

Since the dawn of time (or at least Jan 1, 1970) it has been difficult for the common user to take advantage of removable media under UNIX. The traditional UNIX approach to dealing with removable media has been to let programs name the device containing the media, and to leave it to the operator (or user) to ensure that the right media is in the device named. We have implemented the opposite approach: having the program specify the media and letting the OS take care of the device. Media is referenced by a name in the file system and recognized when it is inserted into a device. Administrators may specify actions to be taken when media is named, recognized, and removed.

## Introduction

The UNIX interface to removable media (floppys, tapes, etc) has traditionally been a minimalist one. The user specifies the physical device in order to gain access to their media. There is no assurance that the expected media is in the device, and the only security is per-device. In addition, there is no general interface for applications to recognize the insertion of media.

Under UNIX, in order to gain access to a file system on a floppy, for example, a user must become root, know what file system type is on the floppy, and execute the mount command (e.g., mount -F pcfs /dev/fd0c /mnt). This is quite a lot for a user to know, especially the users that most workstation companies are trying to attract these days.

In MS-DOS, when a floppy is inserted it is instantly accessible as A: (or B:). The user is not required to know anything about mounting or file systems. The only way to reference a disk, however, is by drive name.

On the Macintosh, users are not forced to deal with drive names, and they do not have to know about file system type. Users of the Macintosh deal with media names, rather than devices.

The UNIX model makes it very difficult to layer applications that use the various forms of removable media. Users of removable media find UNIX systems very difficult to use.

## Media Management

### Goals

Our overall direction has been to create a high level model which will scale to a wide range of removable media, and to provide extensibility at several levels. In effect, we wanted to build a platform on which customers and third parties can build easy to use applications.

We have set forth some high level goals which have guided many of our architectural and design decisions.

*Provide an abstraction for the media*

Applications and users should not refer to a device, instead they should be provided a name to refer to the media.

This is a fundamental change in the existing access model, but it is required to implement any general solution. We have found this to be the biggest stumbling block for people who are used to the "old way".

*Security*

Security should be maintained on a per-media basis, rather than a per-device bases.

Users "own" media, not devices. Protecting the drive from read or write access creates only problems. When media has the notion of an "owner", normal UNIX access semantics can be applied per-media.

*Insertion/ejection paradigm*

There should be a uniform interface for programs to recognize the insertion or ejection of any type of media.

It should be possible to easily implement programs which interact with and take advantage of removable media.

*Operator interface*

Since applications (or users) may reference media that is not in a drive, an interface should be provided to notify operators of media requests.

*Extensible device and label interface*

We cannot predict all the types of devices that users will want to connect to our machines, or the types of media that may exist. Interfaces must exist which customers or vendors can extend the product.

### Design Considerations

*What Media Should be Supported*

Since this mechanism needs to be useful for a broad range of removable media, we had to be careful to not design *out* any particular type of media. We decided to accomplish this by not designing *in*

any particular form of removable media. To provide this level of abstraction, we have moved the details of managing devices and interpreting labels out into dynamic shared objects which are loaded into the user-level media manager at run time. The core of the media manager is only aware of some very basic properties of the devices and labels it is managing.

*Operator Considerations*

Each type of media is potentially used differently. For example, in general, floppies are directly attached to the workstation being used. Magneto-optical disks, however, are generally contained in an autochanger. The mechanism for notifying an "operator" is driven by a configuration file, which allows for directing messages relating to the floppy to the screen and sending an e-mail message to an operator for magneto-optical media.

*Media Names*

A natural UNIX way to implement a new name space is a file system. We have chosen this method to present new block and character special devices that represent media.

For some time, there has been an implementation of a user-level NFS server. It was a fairly trivial matter to take this server and implement our own file system. The user-level NFS server has the advantage of being easy to debug, and all of the interfaces are very well defined.

*Database*

Since we've implemented this name space in terms of a user-level NFS server, we need some way to keep changes to the name space across reboots, and perhaps share the name space between machines. A conflicting goal is for the media manager to "just work" out of the box, with no explicit set-up. We have experimented with two different databases. One that just stores the data in-memory, and another which stores the data in NIS+[2].

The in-memory database is not persistent across reboots, but requires no special configuration. The NIS+ database requires special configuration. We are currently shipping with the in-memory database as the default. In the future, we will be providing a file based database, and we will also be publishing the interface for customers and third partys to craft their own ways to store and share the name space.

*Devices over the network*

An opportunity we considered was to provide a mechanism for users and applications to access devices on different machines, over the network. The model would be that it doesn't matter which machine your chosen media is on, reads and writes to the name would just get to the right place.

At some level, this would be a nice feature to have. The reason we choose not to implement it at this time is because of the complexity involved in matching up ioctls between different machine architectures and flavors of UNIX.

*Other Work*

There are many implementations of "labeled" media access for various operating systems, including UNIX. To our knowledge, all of these require linking with a special library and many require the use of new interfaces. In addition, these implementations frequently support only one type of media.

We decided early on that our support for removable media must use existing interfaces, and that we needed to be able to support a broad range of media types and devices. We also wanted an implementation that would require very little special support from existing device drivers.

**Architecture**

The media manager is implemented as a kernel driver, and a user level daemon, see Figure 1. The daemon is a user-level NFS server, which automatically mounts a file system on "/vol". The daemon presents names of different media in this name space. Names for the media are block and character special devices. The major number of these devices is that of the vol driver, and the minor number is assigned per virtual media. The vol driver is loaded (via an ioctl) with mappings between the "virtual" minor number and a physical device.

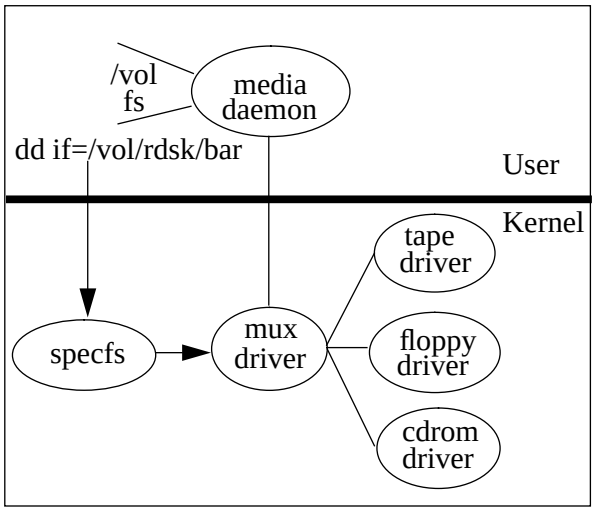


**Figure 1**: Media manager

If a media name is opened and the driver does not contain a mapping for that media, the daemon is notified. At this point, the daemon uses a database to discover where it might be, or might have been, checks the suspected device for the requested media, and loads the mapping into the driver. If the daemon is unable to locate the media, a "notify" event (see below) is generated and someone is told about the problem.

The daemon is controlled by a configuration file, which specifies which devices are being

managed, what sort of media can be expected in a given device, and which labels can be expected on given media. In addition, programs are specified to be executed when certain events happen.

The daemon uses threads [5] to provide concurrency between accesses to the /vol name space, and label checking operations. The daemon is implemented in such a way as to be portable between different threads implementations, making minimum necessary use of the concurrency mechanisms.

*Name Space*

The name space, rooted at /vol, provides access to media names and allows the simple perusal and modification of the names. The file system interface is a very effective tool for presenting and manipulating these names.

In the /vol name space, file system operations work as expected, with the exception that regular files may not be created, and special files may not be explicitly created (via mknod). Other file system operations such as ls, mv, mkdir, chmod, chown, chgrp, ln, and rm work, for the most part, as expected.

The name space consists of a physical portion and a logical portion. The logical part of the name space represents media names independent of their location. A user (or application) can access a piece of media without knowledge of where it is. The "system" can be relied on to provide the correct media at some point in the future (assuming operators are awake and such). The physical part of the name space is provided to allow access to media in a specific drive. For example, media that is unlabeled cannot have a name in the logical part of the name space, but can be accessed through the physical name space.

The name space allows access to both the block and character special interfaces. The logical part of the name space has three directorys: /vol/dsk, /vol/rdsk, and /vol/rmt. /vol/dsk and /vol/rdsk contain the block and character names (respectively) for random access media (disks). /vol/rmt contains the character names for sequential access media (tapes).

The physical part of the name space is contained under /vol/dev. The hierarchy under /vol/dev is intended to mimic /dev to a certain degree. The floppy drive is represented as /vol/dev/fd0, and /vol/dev/rfd0. Media that is inserted in the floppy drive would appear as /vol/dev/fd0/frog, assuming one inserted a floppy named "frog" in fd0. In addition, there are symbolic links that are automatically built to allow the construction of simple programs. The floppy symbolic link would be /vol/dev/aliases/floppy0 -> /vol/dev/rfd0/frog. The "floppy0" alias comes from the configuration file (see below).

Some forms of media support partitions. If a piece of media has partitions, its name in the name space becomes a directory, with the special devices appearing under it. The special devices are named using the SVR4 convention: s0, s1, etc. For example, the "toad" CD-ROM might be represented as /vol/rdsk/toad/s0, and /vol/rdsk/toad/s2.

Tapes devices frequently support several different access methods and densities. Tapes are represented as a directory with the access methods represented as files under the directory. A tape named "foo" would have the following access methods: /vol/rmt/foo/d, /vol/rmt/foo/n, /vol/rmt/foo/b, /vol/rmt/foo/bn. The "d" node would yield default System V semantics, "n" is default System V semantics with no-rewind on close, "b" is Berkeley mode, and "bn" is Berkeley with no-rewind on close.

Before a tape has a label, it can be written at any density. There are four densities supported: low, medium, high, and ultra (l, m, h, u). All the permutations of access methods and densities are available on unlabeled tapes. Once a tape has a label, its density is selected and cannot be written at a different density unless the label is "scratched".

*Labels*

Media is expected to have some sort of label, for normal named access to work. Label formats are specified to the media manager in the form of a shared object (e.g., label_dos.so, Figure 2) with several entry points used to identify and interpret the label. The interface to these shared objects will be published and supported so that customers and third parties can provide support for their own labels. The number of lines of C code to implement support for a label type is around 400.

```
# Labels supported
label dos label_dos.so floppy
label cdrom label_cdrom.so cdrom
label sun label_sun.so floppy
```

**Figure 2**: Excerpt from /etc/vold.conf

Not all media is labeled, in particular media that is fresh out of a box, or media that contains data for exchange between machines without labeled access. For this purpose, a special name is provided in the /vol/dev portion of the name space. For example, a floppy which has a cpio or tar file on it would be available as /vol/dev/fd0/unlabeled. A floppy fresh out of the box would be available as /vol/dev/fd0/unformatted. The distinction between unlabeled and unformatted is that unformatted is not readable by the drive, and is assumed to not have any data on it. Unlabeled, on the other hand has data on it, the format of the label is simply not recognizable.

The media manager only looks at labels, not file systems. For example, it is possible to have a floppy that is unlabeled and still have a file system on it. One way to get such a floppy is to write a tar file out to it, then run newfs on it.

Code currently exists to interpret Sun labels, DOS labels, and CD-ROM labels. The label code can try to look at the label and guess at a good name for that media. If the media manager has never "seen" that piece of media before, it will request a name, owner, group, file modes, and other things from the label code. Existing label drivers are only able to derive a name, however future labels (or custom ones) might keep more information.

CD-ROM labels are an interesting case because the standards for labels on them are very weak. There is no requirement that they be unique in any way. For example, two completely different CD-ROM's may have exactly the same Sun label. The same is true, although less likely, with High Sierra. The solution we've chosen is to assume that the first 64K bytes of data on the CD-ROM is unique. We choose that size because the root directories of UNIX file systems, and High Sierra file systems fall in this range. The root directories (and super blocks) contain time stamps and other unique data. A 128 bit digital signature is then generated from this 64K of data, using the "RSA Data Security, Inc. MD4 Message-Digest Algorithm" [3, 4]. This fast algorithm produces a digital signature which is adequate for our identification purposes.

*Devices*

Devices that the media manager knows how to deal with are represented by a shared object, much like a label (Figure 3). The code that manages a device must initialize the device, build some data structures, and be able to generate a message when new media appears or is ejected. Normally, these "device drivers" create a thread which listens for insertion or ejection events in some device dependent way. Device management code is roughly the same size as most label code, about 400 lines.

The media manager uses existing device drivers with very little modification. In fact, all the prototype work was carried out with no changes to the drivers. The one change that we have made is an ioctl which blocks, waiting for media to be inserted or ejected. This keeps the daemon (actually, the shared object that supports the device) from having to poll the device open routine to see if there is new media.

*Databases*

The media manager maintains information about each piece of media, like the name, owner, group, and so on, in a "database". This database provides for sharing the name space over the network, or simply keeping it local. Since there is such a wide variety of data management requirements out there, we've chosen to open up this interface as well.

*Attributes*

Each piece of media has attributes associated with it. The attributes are things like the type of the label, its owner, the number of partitions, and so on. These are system defined attributes. There is also a generalized interface to provide users or applications the ability to set their own attributes. For example on an audio CD, an audio player program might choose to keep the title of tracks in the database. These attributes are manipulated with library functions and are simply ascii "attribute=value" pairs, much like shell environment variables. Attributes are stored in the database, along with other information about the media.

*Insert, Eject, and Notify Events*

A configuration file allows the specification of a program to be run when new media arrives, media ejection is requested, or when media is referenced but isn't in a drive. These are called insert, eject, and notify events respectively.

Figure 4 shows the specification of insert, eject, and notify events. Each event keyword is followed by a regular expression (sh style) which specifies which names in the name space will trigger a particular event. A series of flags follow, and use the

```
# Devices to use
use cdrom drive /dev/dsk/c0t6 dev_cdrom.so cdrom0
use floppy drive /dev/fd0 dev_floppy.so floppy0
```

**Figure 3**: Excerpt from /etc/vold.conf

```
# Events
insert /vol/dev/fd[0-9]/* user=root /usr/sbin/diskovery -D
insert /vol/dev/dsk/* user=root /usr/sbin/diskovery -D
eject /vol/dev/fd[0-9]/* user=root /usr/sbin/diskovery -D
eject /vol/dev/dsk/* user=root /usr/sbin/diskovery -D
notify /vol/rdsk/* group=tty /usr/lib/vold/volmissing -c
```

**Figure 4**: Excerpt from /etc/vold.conf

keyword=value paradigm. Finally, an event line specifies a program to run when the specified event occurs. Each event program is run, by default, with user=daemon and group=other. The diskovery program [1] (see below) must be root, because it needs to mount file systems. The volmissing program must be in the tty group because it needs permission to write to the console.

In the case of the diskovery program, the -D flag turns on debugging messages to the console.

Eject events are unique because the event is generated after the user has typed the eject(1) command, and before the device is actually ejected. This allows the application to clean up things, sync the file system (or unmount it), or whatever. The exit code for the program is also examined, and if it returns 1, the ejection is denied (EBUSY is returned to the eject ioctl(2)).

## Experience

### Automatic Mounting of CD-ROM and Floppy

For several years, Sun customers have been requesting a mechanism by which they can mount media like CD-ROM's and floppies without having to become root. On top of the media manager, we have implemented "diskovery".

Diskovery is executed when CD-ROM's and floppies are inserted or ejected. On insertion, it will discover the file system type, and attempt to mount the partition. On ejection, it will unmount the file system.

```
# File system identification
ident hsfs ident_hsfs.so cdrom
ident ufs ident_ufs.so cdrom floppy
ident pcfs ident_pcfs.so floppy
```

**Figure 5**: Excerpt from /etc/diskovery.conf

Determining the file system type is performed by a function that is kept in a dynamic shared object. This "ident" function decides if the type is right, and also lets the upper level know if it is "clean". A configuration file (Figure 5) specifies which file systems are appropriate to which media, so that long searches can be avoided. The interface to the shared objects will be published so customers or third parties can provide support for their own file system types. The number of lines in these modules average about 60, most of which is boiler plate.

Dirty file systems are cleaned before mounting, or they are mounted read-only if they are not cleanable.

Mounts are performed with the "nosuid" option, which keeps users from carrying around a floppy or CD-ROM's with set-uid programs on them. In addition, the nosuid semantic has been extended to mean that access to block and character special devices is disallowed. This keeps a user from building a file system with devices like memory or disks to which they have privileged access.

Floppies are mounted via the name /floppy/<media_name>; CD-ROM's are mounted as /cdrom/<media_name>. If a floppy or CD-ROM has partitions, they are each mounted (e.g., /cdrom/solaris_2_0/s0, /cdrom/solaris_2_0/s2, etc.).

After mounting (or unmounting) the file system(s), a list of "actions" are run (Figure 6). These actions can do things like notify the file manager program that new media has arrived. In the case of a CD-ROM, an action is provided to check the media for audio tracks and execute the "workman" program.

Diskovery implements a policy which requires that a file system be unmounted before it can be ejected. The media manager does not require this to be the case, however it greatly simplifies the most common user model to do this.

For the future we are considering adding the ability to specify that inserted media be automatically exported upon insertion. An extension to this would be to use the automounter to make this media available in a "known" place all over the network. Another use of the automounter would be to allow it to mount the file systems locally.

### Magneto-optical Autochanger

We have experimented with a magneto-optical autochanger, to convince ourselves that the implementation scales beyond the simple CD-ROM and floppy on a workstation. One difference between an autochanger and one or two drives is that looking at each disk after a reboot is very time consuming, about 15 minutes for a full autochanger. If the database is persistent, it will remember which slot each disk was in, and not check until it needs that disk. If the disk is not found in the "remembered" slot, it is searched for in the autochanger and other drives that could contain that media type.

For the magneto-optical autochanger, we chose to have the driver provide a full autochanger

```
# Actions
action cdrom action_filemgr.so
action floppy action_filemgr.so
action cdrom action_workman.so /home/rmtc/halt/bin/workman
```

**Figure 6**: Excerpt from /etc/diskovery.conf

interface. Each slot in the autochanger looks to the outside world as a separate device. Access to the devices are scheduled using algorithms implemented in the driver.

### Tapes

We are still early in the development of our media manager support of tapes. Tapes, of course, are sequential access devices and require a slightly different model in some areas. For example, the media manager cannot simply read the tape label after it's written without waiting until the device is no longer busy.

One of the biggest problems with tapes is that they degrade over time. Often this isn't discovered until the desired data can't be read. Tape drives commonly keep all sorts of statistics about error rates. These statistics are almost never collected. Collecting statistics about how much a drive or tape has been used, and what sort of error rates are being seen is a critical part of our tape support.

Currently, there are several tape autochangers available. One way to implement support for these devices is to provide an interface like we did with the magneto-optical autochanger. In other words, hide the autochanger mechanics in the kernel driver. Another way to implement support for these is to build the selection and scheduling of media in to the media manager driver, and use ioctls to "pick" the media that is placed in a drive. We are currently investigating ways to provide "generic" autochanger support using this method.

### Performance

There are several ways to characterize the performance of the media manager. The most interesting is the amount of time it takes between insertion of media in a drive and the time the insert program is executed. This is the most critical for CD-ROM and floppy recognition. Unfortunately, this also poses the biggest problem.

The CD-ROM drive on the SPARCstation is a SCSI device. Being a SCSI device, it is limited to responding to querys, rather than initiating action. In particular, we must ask the device every so often if it has a new piece of media, or if the media has been removed from it. In addition, it takes about 3 seconds for it to scan the surface of the disk upon media insertion. We add another two seconds on top of that to read data off the disk so a unique signature can be generated. If the CD-ROM is being automatically mounted, the act of mounting takes around one second per-partition. We are working along several paths to improve the performance.

Like the CD-ROM drive, the floppy drive doesn't generate interrupts, it just responds to querys. The good news, however, is that once the floppy is detected, reading the label is fairly fast.

The time to eject a CD-ROM or floppy is primarily gated by the speed of fork/exec and umount(2). Currently around two seconds, the time depends on how many partitions were mounted.

The next set of interesting performance metrics is read and write. In particular, does the media manager generate any overhead during normal read and write operations. We have measured the throughput of the vol driver, using a data generator on one end, and dd(1) on the other and the effect of having the vol driver interposed was not measurable.

Operations on names in the /vol name space (e.g., mv, rm, ln, ...) are viewed to be infrequent, and hence their performance is not critical to us. Our performance goal for this was to have operations be on par with NFS operations between like machines. The speed of these operations is affected by which type of database is in use. For the "in memory" database, we exceed the goal by a wide margin. For the experimental NIS+ database, we meet the goal.

## Availability

### Binary in Solaris

We are delivering the media manager into Solaris in phases, starting with minimal functionality required for the automatic mounting of CD-ROM's and floppies. This will be delivered in a release in the near future. The next phase will include support for tape devices and will have all the interfaces documented.

### Source

For support of new devices or labels on machines running Solaris, example code for devices and labels can be provided at no cost.

The product should be fairly portable to other UNIX implementations that support threads. Inquiries for full source to this product are welcome.

## References

[1] Alt, H., "CD-ROM's and Floppies in Solaris", *Proc. Jan 1993 UK UNIX User Group/Sun UK User Group Conference*, January, 1993.

[2] McManis, Chuck, "Naming Systems: A Replacement for NIS", *Proc. September 1991 Sun UK User Group Conference*, September, 1991.

[3] Rivest, R., "The MD4 message digest algorithm", in A.J. Menezes and S.A. Vanstone, editors, Advances in Cryptology - CRYPTO '90 Proceedings, pages 303-311, Springer-Verlag, 1991.

[4] Rivest, R., "The MD4 Message Digest Algorithm", RFC 1186, MIT, October 1990.

[5] D. Stein, D. Shah, "Implementing Lightweight Threads", Proc. 1992 USENIX Summer Conference, June, 1992.

### Author Information

Howard started his career at a startup, administering their machines. He then went to SRI International to do networking and kernel development. Next, he went to Sun Microsystems to do kernel porting for a few years. After getting tired of the bay area, he returned to Austin to work for Tandem on various kernel development projects. Howard currently works for SunSoft's Rocky Mountain Technology Center in the Storage Management group. Howard can be reached at (719) 528-4614, or by mail at halt@central.sun.com.