ZFS Administration Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–2271
November 2005

051118@13215

Contents

Preface

The *ZFS Administration Guide* provides information about setting up and managing ZFS file systems.

This guide contains information for both SPARC® based and x86 based systems.

Who Should Use This Book

This guide is intended for anyone who is interested in setting up and managing ZFS file systems. Experience using Solaris or another UNIX version is recommended.

How This Book Is Organized

The following table describes the chapters in this book.

Chapter	Description
Chapter 1	An overview of ZFS and its features and benefits. It also covers some basic concepts and terminology.
Chapter 2	Step-by-step instructions for setting up simple ZFS configurations with simple pools and filesystems. It also provides the hardware and software required to create ZFS filesystems.

Chapter	Description
Chapter 3	Identifies important topics that make ZFS significantly different from traditional filesystems. Understanding these key differences will help reduce confusion when using traditional tools to interact with ZFS.
Chapter 4	Detailed description of how to create and administer storage pools.
Chapter 5	Detailed information about managing ZFS filesystems. Included are such concepts as hierarchical filesystem layout, property inheritance, and automatic mount point management and share interactions.
Chapter 6	Detailed description of how to create and administer ZFS snapshots and clones.
Chapter 7	Information about using access control lists (ACLs) to protect your ZFS files by providing more granular permissions then the standard UNIX permissions.
Chapter 8	Description of emulated volumes, using ZFS on a Solaris system with zones installed, and alternate root pools.
Chapter 9	Describes how to identify ZFS failure modes and how to recover from them. Steps for preventing failures are covered as well.

Related Books

Related information about general Solaris system administration topics can be found in the following books:

- *Solaris System Administration: Basic Administration*
- *Solaris System Administration: Advanced Administration*
- *Solaris System Administration: Devices and File Systems*
- *Solaris System Administration: Security Services*
- *Solaris Volume Manager Administration Guide*

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is http://docs.sun.com.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at http://docs.sun.com.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction

This chapter provides an overview of ZFS and its features and benefits. It also covers some basic terminology used throughout the rest of this book.

The following sections are provided in this chapter.

1.1 What is ZFS?

The Zettabyte File System (ZFS) is a revolutionary new filesystem that fundamentally changes the way filesystems are administered, with features and benefits not found in any other filesystem available today. ZFS has been designed from the ground up to be robust, scalable, and simple to administer.

1.1.1 Pooled Storage

ZFS uses the concept of *Storage Pools* to manage physical storage. Historically, filesystems were constructed on top of a single physical device. In order to address multiple devices and provide for data redundancy, the concept of a *Volume Manager* was introduced to provide the image of a single device so that filesystems would not have to be modified to take advantage of multiple devices. This added another layer of complexity, and ultimately prevented certain filesystem advances, since the filesystem had no control over the physical placement of data on the virtualized volumes.

ZFS does away with the volume manager altogether. Instead of forcing the administrator to create virtualized volumes, ZFS aggregates devices into a storage pool. The storage pool describes the physical characteristics of the storage (device layout, data redundancy, etc.) and acts as an arbitrary data store from which filesystems can be created. Filesystems are no longer constrained to individual devices, allowing them to share space with all filesystems in the pool. There is no need to predetermine the size of a filesystem, as they grow automatically within the space allocated to the storage pool. When new storage is added, all filesystems within the pool can immediately make use of the additional space without additional work. In many ways, the storage pool acts as a virtual memory system. When a memory DIMM is added to a system, the operating system doesn't force the administrator to invoke some commands to configure the memory and assign it to individual processes — all processes on the system automatically make use of the additional memory.

1.1.2 Transactional Semantics

ZFS is a transactional filesystem, which means that the filesystem state is always consistent on disk. Traditional filesystems overwrite data in place, which means that if the machine loses power between, say, the time a data block is allocated and when it is linked into a directory, the filesystem will be left in an inconsistent state. Historically, this was solved through the use of the `fsck(1M)` command, which was responsible for going through and verifying filesystem state, making an attempt to repair it in the process. This caused great pain to administrators, and was never guaranteed to fix all possible problems. More recently, filesystems have introduced the idea of *journaling*, which records action in a separate journal which can then be replayed safely in event of a crash. This introduces unnecessary overhead (the data needs to be written twice) and often results in a new set of problems (such as when the journal can't be replayed properly).

With a transactional filesystem, data is managed using *copy on write* semantics. Data is never overwritten, and any sequence of operations is either entirely committed or entirely ignored. This means that the filesystem can never be corrupted through accidental loss of power or a system crash, and there is no need for a `fsck(1M)` equivalent. While the most recently written pieces of data may be lost, the filesystem itself will always be consistent. In addition, synchronous data (written using the `O_DSYNC` flag) is always guaranteed to be written before returning, so it is never lost.

1.1.3 Checksums and Self-Healing Data

With ZFS, all data and metadata is checksummed using a user-selectable algorithm. Those traditional filesystems that do provide checksumming have performed it on a per-block basis, out of necessity due to the volume manager layer and traditional filesystem design. This means that certain failure modes (such as writing a complete block to an incorrect location) can result in properly checksummed data that is

actually incorrect. ZFS checksums are stored in a way such that these failure modes are detected and can be recovered from gracefully. All checksumming and data recovery is done at the filesystem layer, and is transparent to the application.

In addition, ZFS provides for self-healing data. ZFS supports storage pools with varying levels of data redundancy, including mirroring and a variation on RAID-5. When a bad data block is detected, not only does ZFS fetch the correct data from another replicated copy, but it will also go and repair the bad data, replacing it with the good copy.

1.1.4 Unparalleled Scalability

ZFS has been designed from the ground up to be the most scalable filesystem, ever. The filesystem itself is a 128-bit filesystem, allowing for 256 quadrillion zettabytes of storage. All metadata is allocated dynamically, so there is no need to pre-allocate inodes or otherwise limit the scalability of the filesystem when it is first created. All the algorithms have been written with scalability in mind. Directories can have up to 2^{48} (256 trillion) entries, and there is no limit on the number of filesystems or number of files within a filesystem.

1.1.5 Snapshots and Clones

A snapshot is a read-only copy of a filesystem or volume. Snapshots can be created quickly and easily. Initially, snapshots consume no additional space within the pool.

As data within the active dataset changes, the snapshot *consumes* space by continuing to reference the old data, and so, prevents it from being freed back to the pool.

1.1.6 Simplified Administration

Most importantly, ZFS provides a greatly simplified administration model. Through the use of hierarchical filesystem layout, property inheritance, and auto-management of mount points and NFS share semantics, ZFS makes it easy to create and manage filesystems without needing multiple different commands or editing configuration files. The administrator can easily set quotas or reservations, turn compression on or off, or manage mount points for large numbers of filesystems with a single command. Devices can be examined or repaired without having to understand a separate set of volume manager commands. Administrators can take an unlimited number of instantaneous snapshots of filesystems, and can backup and restore individual filesystems.

ZFS manages filesystems through a hierarchy that allows for this simplified management of properties such as quotas, reservations, compression, and mount points. In this model, filesystems become the central point of control. Filesystems

themselves are very cheap (equivalent to a new directory), so administrators are encouraged to create a filesystem for each user, project, workspace, etc. This allows the administrator to define arbitrarily fine-grained management points.

1.2 ZFS Terminology

The following table covers the basic terminology used throughout this book.

checksum	A 256-bit hash of the data in a filesystem block. The checksum function can be anything from the simple and fast fletcher2 (the default) to cryptographically strong hashes such as SHA256.
clone	A filesystem whose initial contents are identical to that of a snapshot. For information about clones, see "6.2 ZFS Clones" on page 80.
dataset	A generic name for ZFS entities: clones, filesystems, snapshots, or volumes. For more information about datasets, see Chapter 5.
filesystem	A dataset that contains a standard POSIX filesystem. For more information about filesystems, see Chapter 5.
mirror	A virtual device that stores identical copies of data on two or more disks. If any disk in a mirror fails, any one of the other disks in that mirror can provide the same data.
pool	A logical group of devices describing the layout and physical characteristics of available storage. Space for datasets is allocated from a pool. For more information about storage pools, see Chapter 4.
RAID-Z	A virtual device that stores data and parity on multiple disks, similar to RAID-5. All traditional RAID-5-like algorithms (RAID-4. RAID-5. RAID-6, RDP, and EVEN-ODD, for example) suffer from a problem known as the "RAID-5 write hole": if only part of a RAID-5 stripe is written, and power is lost before all blocks have made it to disk, the parity will remain out of sync with data – and therefore useless – forever (unless a subsequent full-stripe write overwrites it). In RAID-Z, ZFS uses variable-width RAID stripes so that all writes are full-stripe writes. This is only possible because ZFS integrates filesystem and device management in such a way that the

	filesystem's metadata has enough information about the underlying data replication model to handle variable-width RAID stripes. RAID-Z is the world's first software-only solution to the RAID-5 write hole.
snapshot	A read-only image of a filesystem or volume at a given point in time. For more information about snapshots, see "6.1 ZFS Snapshots" on page 77.
virtual device	A logical device in a pool, which can be a physical device, a file, or a collection of devices. For more information on virtual devices, see "4.1 Virtual Devices" on page 29.
volume	A dataset used to emulate a physical device in order to support legacy filesystems. For more information on emulated volumes, see "8.1 Emulated Volumes" on page 103.

Each dataset is identified by a unique name of the ZFS namespace. Datasets are identified using the following format:

pool/*path*[@*snapshot*]

pool	identifies the name of the storage pool that contains the dataset
path	is a slash-delimited pathname for the dataset object
snapshot	is an optional component that identifies a snapshot of a dataset

1.3 ZFS Component Naming Conventions

Each ZFS component must be named according to the following rules:

- Empty components are not allowed.
- Each component can only be composed of alphanumeric characters plus the following special characters: _, -, :, and .
- Pool names must begin with a letter, except that the beginning sequence c[0–9] is not allowed. The pool names 'raidz' and 'mirror' are reserved names.
- Dataset names must begin with an alphanumeric character.

CHAPTER 2

Getting Started

This chapter provides step-by-step instructions for setting up simple ZFS configurations. By the end of this chapter, you should have a basic idea of how the ZFS commands work, and should be able to create simple pools and filesystems. It is not designed to be a comprehensive overview, and refers to later chapters for more detailed information.

The following sections are provided in this chapter.

2.1 Hardware and Software Requirements

Make sure the following hardware and software requirements are met before attempting to use the ZFS software.

- A SPARC or x86 system that is running the Solaris Nevada release, build 27.
- The minimum disk size is 128 Mbytes. The minimum amount of disk space required for a storage pool is 64 Mbytes.
- A minimum of 128 Mbytes of memory.

If you create a mirrored disk configuration, multiple controllers are recommended.

2.2 Creating a Basic Filesystem

ZFS administration has been designed with simplicity in mind. Among the goals of the commands is to reduce the number of commands needed to create a usable filesystem. Assuming that the whole disk `/dev/dsk/c0t0d0` is available for use, the following sequence of commands will create a filesystem for you to use:

```
# zpool create tank c0t0d0
# zfs create tank/fs
```

This creates a new storage pool with the name *tank*, and a single filesystem in that pool with the name *fs*. This new filesystem can use as much of the disk space on `c0t0d0` as needed, and is automatically mounted at `/tank/fs`:

```
# mkfile 100m /tank/fs/foo
# df -h /tank/fs
Filesystem             size   used  avail capacity  Mounted on
tank/fs                 80G   100M    80G     1%    /tank/fs
```

2.3 Creating a Storage Pool

While the previous example serves to illustrate the simplicity of ZFS, it is not a terribly useful example. In the remainder of this chapter, we will demonstrate a more complete example similar to what we would encounter in a real world environment. The first step is to create a storage pool. The pool describes the physical characteristics of the storage and must be created before any filesystems.

▼ Identifying Storage Requirements

Steps **1. Determine available devices.**

Before creating a storage pool, you must determine which devices will store your data. These must be disks of at least 128 Mbytes in size, and must not be in use by other parts of the operating system. The devices can be individual slices on a pre-formatted disk, or they can be entire disks which ZFS will format to be a single large slice. For this example, we will assume that the whole disks `/dev/dsk/c0t0d0` and `/dev/dsk/c0t0d1` are available for use.

For more information on devices and how they are used and labelled, see "4.1.1 Disks" on page 29.

2. **Choose data replication.**

   ZFS supports multiple types of data replication, which determines what types of hardware failures the pool is able to withstand. ZFS supports non-redundant (striped) configurations, as well mirroring and RAID-Z (a variation on RAID-5). For this example, we will use basic mirroring between the two available disks.

▼ Creating the Pool

Steps

1. **Become root or assume an equivalent role with the appropriate ZFS rights profile.**

   For more information about the ZFS rights profiles, see "8.4 ZFS Rights Profiles" on page 109.

2. **Pick a pool name.**

   The name is used to identify this storage pool when using any of the `zpool(1M)` or `zfs(1M)` commands. Most systems will require only a single pool, so you can pick any name that your prefer, provided it satisfies the naming requirements outlined in "1.3 ZFS Component Naming Conventions" on page 17. For this example, we will use the pool name `tank`.

3. **Create the pool.**

   Execute the following command to create the pool:

   ```
   # zpool create tank mirror c0t0d0 c0t0d1
   ```

   If one or more of the devices contains another filesystem or is otherwise in use, the command will refuse to create the pool.

   For more information on creating storage pools, see "4.4.1 Creating a Pool" on page 32.

   For more information on how devices usage is determined, see "4.4.2.1 Detecting In-Use Devices" on page 33.

4. **Viewing the result.**

   You can see that your pool was successfully created by using the `zpool list` command:

   ```
   # zpool list
   NAME                    SIZE    USED   AVAIL    CAP  HEALTH     ALTROOT
   tank                     80G    137K     80G     0%  ONLINE     -
   ```

   For more information on getting pool status, see "4.6 Querying Pool Status" on page 39.

2.4 Creating a Filesystem Hierarchy

Now that you have created a storage pool to hold your data, you can create your filesystem hierarchy. Name hierarchies are a simple yet powerful mechanism for organizing information. They are also very familiar to anyone who has used a filesystem.

ZFS allows filesystems to be organized into arbitrary hierarchies, where each filesystem has only a single parent. The root of the name hierarchy is always the pool name. ZFS leverages this hierarchy by supporting property inheritance, so that common properties can be set quickly and easily on entire trees of filesystems.

▼ Determining Filesystem Hierarchy

Steps

1. **Pick filesystem granularity.**

   ZFS filesystems are the central point of administration. They are lightweight, and can be created easily. A good model to use is a filesystem per user or project, as this allows properties, snapshots, and backups to be controlled on a per-user or per-project basis. For this example, we will be creating a filesystem for each of two users: `bonwick` and `billm`

   For more information on managing filesystems, see Chapter 5.

2. **Group similar filesystems together.**

   ZFS allows filesystems to be organized into hierarchies so that similar filesystems can be grouped together. This provides a central point of administration for controlling properties and administering filesystems. Similar filesystems should be created under a common name. For this example, we will place our two filesystems under a filesystem named `home`.

3. **Choosing filesystem properties.**

   Most filesystem characteristics are controlled using simple properties. These properties control a variety of behavior, including where the filesystems are mounted, how they are shared, whether they use compression, and if there are any quotas in effect. For this example, we want all home directories to be mounted at `/export/home/`*user*, shared via NFS, and with compression. In addition, we will enforce a quota of 10 Gbytes on `bonwick`.

   For more information on properties, see "5.2 ZFS Properties" on page 54.

▼ Creating Filesystems

Steps

1. **Become root or assume an equivalent role with the appropriate ZFS rights profile.**

   For more information about the ZFS rights profiles, see "8.4 ZFS Rights Profiles" on page 109.

2. **Create desired hierarchy.**

   In this example, we create a filesystem that will act as a container for individual file systems:

   ```
   # zfs create tank/home
   ```

   Now we can group our individual filesystems under the `home` file system in our pool `tank`

3. **Set inherited properties.**

   Now that we have established a filesystem hierarchy, we want to set up any properties that should be shared among all users:

   ```
   # zfs set mountpoint=/export/zfs tank/home
   # zfs set sharenfs=on tank/home
   # zfs set compression=on tank/home
   ```

   For more information on properties and property inheritance, see "5.2 ZFS Properties" on page 54.

4. **Create individual filesystems.**

   Now we can create our individual user filesystems. Note that we could have also created the filesystems first and then changed properties at the `home` level. All properties can be changed dynamically while filesystems are in use.

   ```
   # zfs create tank/home/bonwick
   # zfs create tank/home/billm
   ```

   These filesystems inherit their property settings from their parent, so they be automatically mounted at `/export/zfs/`*user* and shared via NFS. There is no need to edit the `/etc/vfstab` or `/etc/dfs/dfstab` file.

   For more information on creating filesystems, see "5.1.1 Creating a Filesystem" on page 52.

   For more information on mounting and sharing filesystems, see "5.5 Mounting and Sharing File Systems" on page 66.

5. **Set filesystem-specific properties.**

   As mentioned in the previous task, we want to give `bonwick` a quota of 10 Gbytes. This places a limit on the amount of space he can consume, regardless of how much space is available in the pool:

   ```
   # zfs set quota=10G tank/home/bonwick
   ```

6. **View the results.**

   Display available filesystem information with the `zfs list` command:

```
# zfs list
NAME                   USED  AVAIL  REFER  MOUNTPOINT
tank                  92.0K  67.0G   9.5K  /tank
tank/home             24.0K  67.0G     8K  /export/zfs
tank/home/billm          8K  67.0G     8K  /export/zfs/billm
tank/home/bonwick        8K  10.0G     8K  /export/zfs/bonwick
```

   Note that `bonwick` only has 10 Gbytes of space available, while `billm` can use the full pool (67 Gbytes).

   For more information on getting filesystem status, see "5.3 Querying Filesystem Information" on page 60.

   For more information on how space is used and calculated, see "3.2 Space Accounting" on page 26.

CHAPTER 3

Differences from Traditional Filesystems

Before continuing further with ZFS, there are some important topics that differ significantly from traditional filesystems. Understanding these key differences will help reduce confusion when using traditional tools to interact with ZFS.

The following sections are provided in this chapter.

3.1 ZFS Filesystem Granularity

Historically, filesystems were constrained to one device so that the filesystems themselves were constrained to the size of the device. Creating and recreating traditional filesystems because of size constraints are time-consuming and sometimes difficult. Traditional volume management products helped manage this process.

Because ZFS filesystems are not constrained to specific devices, they can be created easily and quickly like directories. They grow automatically within the space allocated to the storage pool.

Instead of creating one filesystem, such as `/export/home`, to manage many user subdirectories, you can create one filesystem per user. In addition, ZFS also provides a filesystem hierarchy so that it is easy to set up and manage many filesystems by applying properties that can be inherited by filesystems contained within the hierarchy.

For an example of creating a filesystem hierarchy, see "2.4 Creating a Filesystem Hierarchy" on page 22.

3.2 Space Accounting

ZFS is based on a concept of pooled storage. Unlike typical file systems, which are mapped to physical storage, all ZFS file systems in a pool share the available storage in the pool. So the available space reported by utilities like `df` may change even when the file systems is inactive, as other file systems in the pool consume or release space. Note that maximum file system size can be limited using quotas (see "5.6.1 Setting Quotas" on page 72), and space can be guaranteed to a file system using reservations (see "5.6.2 Setting Reservations" on page 73). The user experience of this model is very similar to the NFS experience when multiple directories are mounted from the same file systems (consider `/home`).

All metadata in ZFS is allocated dynamically. Most other file systems pre-allocate much of their metadata. As a result, there is an immediate space cost at file system creation for this metadata. This also means that the total number of files supported by the file systems is predetermined. Since ZFS allocates its metadata as it needs it, there is no initial space cost, and the number of files is limited only by the available space. The output from the `df -g` command must be interpreted differently for ZFS than other file systems: the `total files` reported is only an estimate based on the amount of storage available in the pool.

ZFS is a transactional file system. Most file system modifications are bundled into transaction groups and committed to disk asynchronously. Until they are committed to disk, they are termed *pending changes*. The amount of space used, available, and referenced by a file or filesystem does not take into account pending changes. Pending changes are generally accounted for within a few seconds. Even committing a change to disk using `fsync(3c)` or `O_SYNC` does not necessarily guarantee that the space usage information will be updated immediately.

3.3 Out of Space Behavior

File system snapshots (see Chapter 6) are "cheap and easy" to make in ZFS. It is anticipated that they will be common in most ZFS environments. The presence of snapshots can cause some unexpected behavior when attempting to free up space. It is generally expected that, given appropriate permissions, one can always remove a file from a full file system, and that this action results in more space becoming available in

the file system. However, if the file to be removed exists in a snapshot of the file system, then there is no space gained from the file deletion. The blocks making up the file continue to be referenced from the snapshot. In fact, the file deletion can end up consuming more disk space, since a new version of the directory will need to be created to reflect the new state of the namespace. This means that one can get an unexpected `ENOSPC` or `EDQUOT` when attempting to remove a file.

3.4 Mounting Filesystems

ZFS is designed to reduce complexity and ease administration. For example, in existing systems it is necessary to edit `/etc/vfstab` every time a new filesystem is added. ZFS has eliminated this need by automatically mounting and unmounting filesystems according to properties of the dataset. There is no need to add ZFS entries to the `/etc/vfstab` file.

For more information on mounting and sharing filesystems, see "5.5 Mounting and Sharing File Systems" on page 66.

3.5 Volume Management

As described in "1.1.1 Pooled Storage" on page 13, ZFS eliminates the need for a separate volume manager. ZFS operates on raw devices, however, so it's possible to create a storage pool comprised of logical volumes (either software or hardware). This is not a recommended configuration, as ZFS works best when using raw physical devices. Using a logical volumes may sacrifice performance and/or reliability, and should be avoided.

3.6 ACLs

Previous versions of Solaris supported an ACL implementation that was primarily based on the POSIX ACL draft specification. The POSIX-draft based ACLs are used to protect UFS files. A new ACL model that is based on the NFSv4 specification is used to protect ZFS files.

The main differences of the new ACL model are as follows:

- Based on the NFSv4 specification and are similar to NT-style ACLs.
- Much more granular set of access privileges.
- Set and displayed with the `chmod` and `ls` commands rather than the `setfacl` and `getfacl` commands.
- Richer inheritance semantics for designating how access privileges are applied from directory to subdirectories, and so on.

For more information about using ACLs with ZFS files, see Chapter 7.

CHAPTER 4

Managing Storage Pools

This chapter contains a detailed description of how to create and administer storage pools.

The following sections are provided in this chapter.

4.1 Virtual Devices

Before getting into detail about how exactly pools are created and managed, you must first understand some basic concepts about virtual devices. Each storage pool is comprised of one or more virtual devices, which describe the layout of physical storage and its fault characteristics.

4.1.1 Disks

The most basic building block for a storage pool is a piece of physical storage. This can be any block device of at least 128 Mbytes in size. Typically, this is some sort of hard drive visible to the system in the `/dev/dsk` directory. A storage device can be a whole disk (`c0t0d0`) or an individual slice (`c0t0d0s7`). The recommended mode of operation is to use an entire disk, in which case the disk does not need to be specially formatted. ZFS formats the disk using an EFI label to contain a single, large slice. When used in this fashion, the partition table (as displayed by `format(1M)`) looks similar to the following:

```
Current partition table (original):
Total disk sectors available: 71670953 + 16384 (reserved sectors)

Part      Tag    Flag     First Sector         Size         Last Sector
  0        usr    wm                34      34.18GB           71670953
  1 unassigned    wm                 0          0                  0
  2 unassigned    wm                 0          0                  0
  3 unassigned    wm                 0          0                  0
  4 unassigned    wm                 0          0                  0
  5 unassigned    wm                 0          0                  0
  6 unassigned    wm                 0          0                  0
  7 unassigned    wm                 0          0                  0
  8   reserved    wm          71670954       8.00MB           71687337
```

In order to use whole disks, the disks must be named in a standard Solaris fashion (`/dev/dsk/cXtXdXsX`). Some third party drivers use a different naming scheme or place disks in a location other than `/dev/dsk`. In order to use these disks, you must manually label the disk and provide a slice to ZFS. Disks can be labelled with EFI labels or a traditional Solaris VTOC label. Slices should only be used when the device name is non-standard, or when there a single disk must be shared between ZFS and UFS or swap or a dump device. Disks can be specified using either the full path (such as `/dev/dsk/c0t0d0`) or a shorthand name consisting of the filename within `/dev/dsk` (such as `c0t0d0`). For example, the following are all valid disk names:

- `c1t0d0`
- `/dev/dsk/c1t0d0`
- `c0t0d6s2`
- `/dev/foo/disk`

ZFS works best when given whole physical disks. You should refrain from constructing logical devices using a volume manager (SVM or VxVM) or hardware volume manager (LUNs or hardware RAID). While ZFS functions properly on such devices, it may result in less-than-optimal performance.

Disks are identified both by their path and their device ID (if available). This allows devices to be reconfigured on a system without having to update any ZFS state. If a disk is switched between controller 1 and controller 2, ZFS uses the device ID to detect that the disk has moved and should now be accessed using controller 2. The device ID is unique to the drive's firmware. While unlikely, some firmware updates have been known to change device IDs. If this happens, ZFS will still be able to access the device by path (and will update the stored device ID automatically). If you manage to change both the path and ID of the device, then you will have to export and re-import the pool in order to use it.

4.1.2 Files

ZFS also allows for ordinary files within a pool. This is aimed primarily at testing and enabling simple experimentation. It is not designed for production use. The reason is that **any use of files relies on the underlying filesystem for consistency**. If you create a ZFS pool backed by files on a UFS filesystem, then you are implicitly relying on UFS to guarantee correctness and synchronous semantics.

However, files can be quite useful when first trying out ZFS or experimenting with more complicated layouts when not enough physical devices are present. All files must be specified as complete paths, and must be at least 128 megabytes in size. If a file is moved or renamed the pool must be exported and re-imported in order to use it, as there is no device ID associated with files by which they can be located.

4.1.3 Mirrors

ZFS provides two levels of data redundancy: mirroring and RAID-Z.

A mirrored storage pool configuration requires at least two disks, preferrably on separate controllers.

4.1.4 RAID-Z

In addition to a mirrored storage pool configuration, ZFS provides a RAID-Z configuration.

RAID-Z is similar to RAID-5 except that it does full-stripe writes, so there's no write hole as described in the RAID-Z description in "1.2 ZFS Terminology" on page 16.

You need at least two disks for a RAID-Z configuration. Other than that, no special hardware is required to create a RAID-Z configuration.

Currently, RAID-Z provides single parity. If you have 3 disks in a RAID-Z configuration, 1 disk is used for parity.

4.2 Self Healing Data

ZFS provides for self-healing data. ZFS supports storage pools with varying levels of data redundancy, as described above.

When a bad data block is detected, not only does ZFS fetch the correct data from another replicated copy, but it will also go and repair the bad data, replacing it with the good copy.

4.3 Dynamic Striping

For each virtual device added to the pool, ZFS dynamically stripes data across all available devices. The decision where to place data is done at write time, so there is no need to create fixed width stripes at allocation time. Virtual devices can be added to a pool and ZFS gradually allocates data to the new device in order to maintain performance and space allocation policies.

As a result, storage pools can contain multiple "top level" virtual devices. Each virtual device can also be a mirror or a RAID-Z device containing other disk or file devices. This flexibility allows for complete control over the fault characteristics of your pool. Given 4 disks, for example, you could create the following configurations:

- Four disks using dynamic striping
- One four-way RAID-Z configuration
- Two two-way mirrors using dynamic striping

While ZFS supports combining different types of virtual devices within the same pool, it is not a recommended practice. For example, you can create a pool with a two-way mirror and a 3-way RAID-Z configuration, but your fault tolerance is as good as your worst virtual device (RAID-Z in this case). The recommended practice is to use top level virtual devices all of the same type with the same replication level in each.

4.4 Creating and Destroying Pools

By design, creating and destroying pools is fast and easy. But beware, although checks are performed to prevent using devices known to be in-use in a new pool, it is not always possible to know a device is already in use. Destroying a pool is even easier. It is a simple command with significant consequences. Use it with caution.

4.4.1 Creating a Pool

To create a storage pool, use the `zpool create` command. The command takes a pool name and any number of virtual devices. The pool name must satisfy the naming conventions outlined in "1.3 ZFS Component Naming Conventions" on page 17.

4.4.1.1 Basic Pool

The following command creates a new pool named `tank` consisting of the disks `c0t0d0` and `c1t0d0`:

```
# zpool create tank c0t0d0 c1t0d0
```

As described in the previous section, these whole disks are found under the `/dev/dsk` directory and labelled appropriately to contain a single, large slice. Data is dynamically striped across both disks.

4.4.1.2 Mirrored Pool

To create a mirrored pool, use the `mirror` keyword, followed by any number of storage devices comprising the mirror. Multiple mirrors can be specified by repeating the `mirror` keyword on the command line. The following command creates a pool with two, two-way mirrors:

```
# zpool create tank mirror c0d0 c1d0 mirror c0d1 c1d1
```

The second `mirror` keyword indicates that a new top-level virtual device is being specified. Data is dynamically striped across both mirrors, with data being replicated between each disk appropriately.

4.4.1.3 RAID-Z Pool

Creating a RAID-Z pool is identical to a mirrored pool, except that the `raidz` keyword is used instead of `mirror`. The following command creates a pool with a single RAID-Z device consisting of 5 disk slices:

```
# zpool create tank raidz c0t0d0s0 c0t0d1s0 c0t0d2s0 c0t0d3s0
/dev/dsk/c0t0d4s0
```

In the above example, the disk must have been pre-formatted to have an appropriately sized slice zero. The above command also demonstrates that disks can be specified using their full path. `/dev/dsk/c0t0d4s0` is identical to `c0t0d4s0` by itself.

Note that there is no requirement to use disk slices in a RAID-Z configuration. The above command is just an example of using disk slices in a storage pool.

4.4.2 Handling Pool Creation Errors

There are a number of reasons that a pool cannot be created. Some of these are obvious, such as when a specified device doesn't exist, while others are more subtle.

4.4.2.1 Detecting In-Use Devices

Before formatting a device, ZFS first checks to see if it is in use by ZFS or some other part of the operating system. If this is the case, you may see errors such as:

```
# zpool create tank c0t0d0 c1t0d0
invalid vdev specification
use '-f' to override the following errors:
```

```
/dev/dsk/c0t0d0s0 is currently mounted on /
/dev/dsk/c0t0d0s1 is currently mounted on swap
/dev/dsk/c1t1d0s0 is part of active pool 'tank'
```

Some of these errors can be overridden by using the `-f` flag, but most cannot. The following uses cannot be overridden using `-f`, and must be manually corrected by the administrator:

Mounted filesystem — The disk or one of its slices contains a filesystem that is currently mounted. To correct this error, use the umount(1M) command.

Filesystem in /etc/vfstab — The disk contains a filesystem that is listed in `/etc/vfstab`, but is not currently mounted. To correct this error, remove or comment out the line in the `/etc/vfstab` file.

Dedicated dump device — The disk is in use as the dedicated dump device for the system. To correct this error, use the dumpadm(1M) command.

Part of ZFS pool — The disk or file is part of an active ZFS storage pool. To correct this error, use the zpool(1M) command to destroy the pool.

The following in-use checks serve as helpful warnings, and can be overridden using the `-f` flag to create:

Contains a filesystem — The disk contains a known filesystem, though it is not mounted and doesn't appear to be in use.

Part of volume — The disk is part of a SVM or VxVM volume.

Live upgrade — The disk is in use as an alternate boot environment for live upgrade.

Part of exported ZFS pool — The disk is part of a storage pool that has been exported or manually removed from a system. In the latter case, the pool is reported as `potentially active`, as the disk may or may not be a network-attached drive in use by another system. Care should be taken when overriding a potentially active pool.

The following example demonstrates how the `-f` flag is used:

```
# zpool create tank c0t0d0
invalid vdev specification
use '-f' to override the following errors:
/dev/dsk/c0t0d0s0 contains a ufs filesystem
# zpool create -f tank c0t0d0
```

It is recommended that you correct the errors rather than using the `-f` flag.

4.4.2.2 Mismatched Replication Levels

Creating pools with virtual devices of different replication levels is not recommended. The `zpool(1M)` command tries to prevent you from accidentally creating a pool with mismatched replication levels. If you try to create a pool with such a configuration, you will see errors similar to the following:

```
# zpool create tank c0t0d0 mirror c0t0d1 c0t0d2
invalid vdev specification: mirror requires at least 2 devices
# zpool create tank mirror c0t0d0 c0t0d1 mirror c1t0d0 c1t0d1 c1t0d2
use '-f' to override the following errors:
mismatched replication level: 2-way mirror and 3-way mirror vdevs are present
```

These errors can be overridden with the `-f` flag, though doing so is not recommended. The command also warns about creating a mirrored or RAID-Z pool using devices of different sizes. While this is allowed, it results in unused space on the larger device, and requires the `-f` flag to override the warning.

4.4.2.3 Doing a Dry Run

Because there are so many ways that creation can fail unexpectedly, and because formatting disks is such a potentially harmful action, the `zfs create` command has an additional option, `-n`, which simulates creating the pool without actually writing data to disk. This option does the device in-use checking and replication level validation, and reports any errors in the process. If no errors are found, you see output similar to the following

```
# zpool create -n tank mirror c1t0d0 c1t1d1
would create 'tank' with the following layout:

        tank
          mirror
            c1t0d0
            c1t1d0
```

There are some errors that cannot be detected without actually creating the pool. The most common example is specifying the same device twice in the same configuration. This cannot be reliably detected without writing the data itself, so it is possible for `create -n` to report success and yet fail to create the pool when run for real.

4.4.2.4 Default Mount Point for Pools

When a pool is created, the default mount point for the root dataset is */poolname* by default. This directory must either not exist (in which case it is automatically created) or empty (in which case the root dataset is mounted on top of the existing directory). To create a pool with a different default mount point, use `zpool create`'s `-m` option:

```
# zpool create home c0t0d0
default mountpoint '/home' exists and is not empty
use '-m' option to specify a different default
# zpool create -m /export/zfs home c0t0d0
```

This command creates a new pool `home` with the `home` dataset having a mount point of `/export/zfs`.

For more information on mount points, see "5.5.1 Managing Mount Points" on page 66.

4.4.3 Destroying Pools

Pools are destroyed by using the `zpool destroy` command. This command destroys the pool even if it contains mounted datasets.

```
# zpool destroy tank
```

Caution – Currently, once a pool is destroyed, your data is gone. Be very careful when you destroy a pool.

4.4.3.1 Destroying a Pool With Faulted Devices

The act of destroying a pool requires that data be written to disk to indicate that the pool is no longer valid. This prevents the devices from showing up as a potential pool when doing an import. If one or more devices is unavailable, the pool can still be destroyed but the necessary state won't be written to these damaged devices. This means that these devices, when suitably repaired, are reported as 'potentially active' when creating new pools, and appear as valid devices when searching for pools to import. If a pool has enough faulted devices such that the pool itself is faulted (a top-level virtual device is faulted), then the command prints a warning and refuses to complete without the `-f` flag. This is because we cannot open the pool, so we don't know if there is any data stored or not. For example:

```
# zpool destroy tank
cannot destroy 'tank': pool is faulted
use '-f' to force destruction anyway
# zpool destroy -f tank
```

For more information on pool and device health, see "4.6.3 Health Status" on page 43.

For more information on importing pools, see "4.7.5 Importing Pools" on page 49.

4.5 Device Management

Most of the basic information regarding devices is covered in "4.1 Virtual Devices" on page 29. Once a pool has been created, there are a number of tasks to be done when managing the physical devices within the pool.

4.5.1 Adding Devices to a Pool

Space can be dynamically added to a pool by adding a new top-level virtual device. This space is immediately available to all datasets within the pool. To add a new virtual device to a pool, use the `zpool add` command:

```
# zpool add scoop mirror c0t1d0 c1t1d0
```

The format of the virtual devices is the same as for the `zpool create` command, and the same rules apply. Devices are checked to see if they are in use, and the command refuses to change the replication level unless the `-f` flag is given. The command also supports the `-n` option to do a dry run:

```
# zpool add -n scoop mirror c0t2d0 c1t2d0
would update 'scoop' to the following configuration:
      scoop
        mirror
            c0t0d0
            c1t0d0
        mirror
            c0t1d0
            c1t1d0
        mirror
            c0t2d0
            c1t2d0
```

The above command syntax would add mirrored devices `c0t2d0` and `c1t2d0` to pool `scoop`'s existing configuration.

For more information on how virtual device validation is done, see "4.4.2.1 Detecting In-Use Devices" on page 33.

4.5.2 Onlining and Offlining Devices

ZFS allows individual devices to be taken offline or brought online. When hardware is flaky or otherwise bad, ZFS will continue to read or write data to the device, assuming the condition is only temporary. If this is not a temporary condition, it is possible to tell ZFS to ignore the device by bringing it offline. ZFS will no longer send any requests to an offlined device.

Devices do not need to be offlined in order to replace them.

Bringing a device offline is an informational hint to ZFS to avoid sending requests to the device.

4.5.2.1 Taking a Device Offline

To take a device offline, use the `zpool offline` command. The device can be specified by path, or short name (if it is a disk). For example:

```
# zpool offline tank c0t0d0
bringing device 'c0t0d0' offline
```

You cannot offline a pool to the point where it becomes faulted. For example, you cannot offline two devices out of a RAID-Z configuration, nor can you offline a top-level virtual device. Offlined devices show up in the OFFLINE state when querying pool status. If you truly want to offline a device and cause your pool to become faulted, you can do so using the `-f` flag. Note that doing so prevents *any* data from being accessed, and may result in I/O errors and system panics.

```
# zpool offline tank c0t0d0
cannot offline /dev/dsk/c1t2d0: no valid replicas
```

By default, the offline state is persistent; the device remains offline when the system is rebooted.

For more information on device health, see "4.6.3 Health Status" on page 43.

4.5.2.2 Bringing a Device Online

Once a device is taken offline, it can be restored by using the `zpool online` command:

```
# zpool online tank c0t0d0
bringing device 'c0t0d0' online
```

When a device is brought online, any data that has been written to the pool is resynced to the newly available device. Note that you cannot use device onlining to replace a disk. If you offline a device, replace the drive, and try to bring it online, it remains in the faulted state.

For more information on replacing devices, see "9.6 Repairing a Damaged Device" on page 121.

4.5.3 Replacing Devices

You can replace a device in a storage pool by using the `zpool replace` command.

```
# zpool replace tank c0t0d0 c0t0d1
```

In the above example, the previous device, `c0t0d0`, is replaced by `c0t0d1`.

The replacement device must be either equal in size or larger than the previous device. If the replacement device is larger, the pool size is increased when the replacement is complete.

For more information about replacing devices, see "9.5 Repairing a Missing Device" on page 120 and "9.6 Repairing a Damaged Device" on page 121.

4.6 Querying Pool Status

The `zpool(1M)` command provides a number of ways to get information regarding pool status. The information available generally falls into three categories: basic usage information, I/O statistics, and health status. These are covered in this section.

4.6.1 Basic Pool Information

The `zpool list` command is used to display basic information about pools.

4.6.1.1 Listing All Information

With no arguments, the command displays all the fields for all pools on the system:

```
# zpool list
NAME                    SIZE    USED   AVAIL    CAP  HEALTH     ALTROOT
tank                   80.0G   22.3G   47.7G    28%  ONLINE     -
dozer                   1.2T    384G    816G    32%  ONLINE     -
```

The field display the following information:

NAME The name of the pool.

SIZE	The total size of the pool, equal to the sum of the size of all top level virtual devices.
USED	The amount of space allocated by all datasets and internal metadata. Note that this is different from the amount of space as reported at the filesystem level. For more information on determining available filesystem space, see "3.2 Space Accounting" on page 26.
AVAILABLE	The amount of unallocated space in the pool.
CAPACITY (CAP)	The amount of space used, expressed as a percentage of total space.
HEALTH	The current health status of the pool. For more information on pool health, see "4.6.3 Health Status" on page 43.
ALTROOT	The alternate root of the pool, if any. For more information on alternate root pools, see "8.3 ZFS Alternate Root Pools" on page 108.

You can also gather statistics for an individual pool by specifying the pool name:

```
# zpool list tank
NAME                    SIZE    USED   AVAIL    CAP  HEALTH     ALTROOT
tank                   80.0G   22.3G   47.7G    28%  ONLINE     -
```

4.6.1.2 Listing Individual Statistics

Individual statistics can be specified using the `-o` option. This allows for custom reports or a quick way to list pertinent information. For example, to list only the name and size of each pool:

```
# zpool list -o name,size
NAME                     SIZE
tank                    80.0G
dozer                    1.2T
```

The column names correspond to the properties listed in the previous section.

4.6.1.3 Scripting

The default output for the `zpool list` command is designed to be human-readable, and is not easy to use as part of a shell script. In order to aid programmatic uses of the command, the `-H` option can be used to suppress the column headings and separate fields by tabs, rather than space padding. For example, to get a simple list of all pool names on the system:

```
# zpool list -Ho name
tank
dozer
```

Or a script-ready version of the earlier example:

```
# zpool list -H -o name,size
tank   80.0G
dozer  1.2T
```

4.6.2 I/O Statistics

To get I/O statistics for a pool or individual virtual devices, use the `zpool iostat` command. Similar to the `iostat(1M)` command, this can display a static snapshot of all I/O activity so far, as well as updated statistics every specified interval. The following statistics are reported:

USED CAPACITY	The amount of data currently stored in the pool or device. This differs from the amount of space available to actual filesystems by a small amount due to internal implementation details. For more information on the difference between pool space and dataset space, see "3.2 Space Accounting" on page 26.
AVAILABLE CAPACITY	The amount of space available in the pool or device. As with the `used` statistic, this is differs from the amount of space available to datasets by a small margin.
READ OPERATIONS	The number of read I/O operations sent to the pool or device, including metadata requests.
WRITE OPERATIONS	The number of write I/O operations sent to the pool or device.
READ BANDWIDTH	The bandwidth of all read operations (including metadata), expressed as units per second.
WRITE BANDWIDTH	The bandwidth of all write operations, expressed as units per second.

4.6.2.1 Pool Wide Statistics

With no options, the `zpool iostat` command displays the accumulated statistics since boot for all pools on the system:

```
# zpool iostat
               capacity     operations    bandwidth
pool         used  avail   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
tank         100G  20.0G   1.2M   102K   1.2M  3.45K
dozer       12.3G  67.7G   132K  15.2K  32.1K  1.20K
```

These statistics are since boot, so bandwidth may appear low if the pool is relatively idle. A more accurate view of current bandwidth usage can be seen by specifying an interval:

```
# zpool iostat tank 2
               capacity     operations    bandwidth
pool         used  avail   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
tank         100G  20.0G   1.2M   102K   1.2M  3.45K
tank         100G  20.0G    134      0  1.34K      0
tank         100G  20.0G     94    342  1.06K   4.1M
```

The above command displays usage statistics only for the pool `tank` every two seconds until the user types Ctrl-C. Alternately, you can specify an additional `count` parameter, which causes the command to terminate after the specified number of iterations. For example, `zpool iostat 2 3` would print out a summary every two seconds for 3 iterations, for a total of six seconds. If there is a single pool, then the statistics is displayed on consecutive lines as shown above. If there is more than one pool, then an additional newline delineates each iteration to provide visual separation.

4.6.2.2 Virtual Device Statistics

In addition to pool-wide I/O statistics, the `zpool iostat` command can also display statistics for individual virtual devices. This can be used to identify abnormally slow devices, or simply observe the distribution of I/O generated by ZFS. To see the complete virtual device layout as well as all I/O statistics, use the `zpool iostat -v` command:

```
# zpool iostat -v
               capacity     operations    bandwidth
tank         used  avail   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
mirror      20.4G  59.6G      0     22      0  6.00K
  c0t0d0        -      -      1    295  11.2K   148K
  c1t1d0        -      -      1    299  11.2K   148K
----------  -----  -----  -----  -----  -----  -----
total       24.5K   149M      0     22      0  6.00K
```

There are a few important things to remember when viewing I/O statistics on a virtual device basis. The first thing you'll notice is that space usage is only available for top-level virtual devices. The way in which space is allocated among mirror and RAID-Z virtual devices is particular to the implementation and not easily expressed as a single number. The other important thing to note is that numbers may not add up exactly as you would expect them to. In particular, operations across RAID-Z and mirrored devices will not be exactly equal. This is particularly noticeable immediately after a pool is created, as a significant amount of I/O is done directly to the disks as part of pool creation that is not accounted for at the mirror level. Over time, these numbers should gradually equalize, although broken, unresponsive, or offlined devices can affect this symmetry as well.

The same set of options (interval and count) can be used when examining virtual device statistics as well.

4.6.3 Health Status

ZFS provides an integrated method of examining pool and device health. The health of a pool is determined from the state of all its devices. This section describes how to determine pool and device health. It does not document how to repair or recover from unhealthy pools. For more information on troubleshooting and data recovery, see Chapter 9.

Each device can fall into one of the following states:

ONLINE	The device is in normal working order. While some transient errors may still be seen, the device is in otherwise working order.
DEGRADED	The virtual device has experienced failure, but is still able to function. This is most common when a mirror or RAID-Z device has lost one or more constituent devices. The fault tolerance of the pool may be compromised, as a subsequent fault in another device may be unrecoverable.
FAULTED	The virtual device is completely inaccessible. This typically indicates total failure of the device, such that ZFS is incapable of sending or receiving data from it. If a top level virtual device is in this state, then the pool is completely inaccessible.
OFFLINE	The virtual device has been explicitly offlined by the administrator.

The health of a pool is determined from the health of all its top-level virtual devices. If all virtual devices are ONLINE, then the pool is also ONLINE. If any one of them is DEGRADED, then the pool is also DEGRADED. If a top level virtual device is FAULTED or OFFLINE, then the pool is also FAULTED. A pool in the faulted state is completely inaccessible — no data can be recovered until the necessary devices are attached or repaired. A pool in the degraded state continues to run, but you may not be getting the same level of data replication level or data throughput you would be if the pool were online.

4.6.3.1 Basic Health Status

The simplest way to get a quick overview of pool health status is with the `zpool status` command:

```
# # zpool status -x
all pools are healthy
```

Particular pools can be examined by specifying a pool name to the command. Any pool not in the ONLINE state should be investigated for potential problems, as described in the next section.

4.6.3.2 Detailed Health Status

A more detailed health summary can be found by using the `-v` option:

```
# zpool status -v tank
  pool: tank
 state: DEGRADED
status: One or more devices could not be opened.  Sufficient replicas exist
        for the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
   see: http://www.sun.com/msg/ZFS-8000-2Q
 scrub: none requested
config:

        NAME                STATE     READ WRITE CKSUM
        tank                DEGRADED     0     0     0
          mirror            DEGRADED     0     0     0
            c0t0d0          FAULTED      0     0     0  cannot open
            c0t0d1          ONLINE       0     0     0
```

This displays a more complete description of why the pool is in its current state, including a human-readable description of the problem and a link to a knowledge article for more information. Each knowledge article provides up-to-date information on the best way to recover from your current situation. Using the detailed configuration information, you should be able to determine which device is damaged and how to repair the pool.

If a pool has a faulted or offlined device, the output of this command identifies the problem pool. For example:

```
# zpool status -x
  pool: tank
 state: DEGRADED
status: One or more devices has been taken offline by the adminstrator.
        Sufficient replicas exist for the pool to continue functioning in a
        degraded state.
action: Online the device using 'zpool online' or replace the device with
        'zpool replace'.
 scrub: none requested
config:

        NAME                STATE     READ WRITE CKSUM
        tank                DEGRADED     0     0     0
          mirror            DEGRADED     0     0     0
            c0t0d0          OFFLINE      0     0     0
            c1t0d0          ONLINE       0     0     0
```

The `READ` and `WRITE` columns gives a count of I/O errors seen on the device, while the `CKSUM` column gives a count of uncorrectable checksum errors seen on the device. Both of these likely indicate potential device failure, and some corrective action is needed. If you see non-zero errors for a top-level virtual device, it may indicate that portions of your data have become inaccessible.

For more information on diagnosing and repairing faulted pools and data, see Chapter 9.

4.7 Storage Pool Migration

It is occasionally necessary to move a storage pool between machines. To do so, the storage devices must be disconnected from the original machine and reconnected to the destination machine. This can be accomplished by physically recabling the devices, or by using multi-ported devices such as those on a SAN. ZFS provides the means to export the pool from one machine and import it on the destination machine, even if the machines are of different endianness. For information on replicating or migrating filesystems between different storage pools (which may be on different machines), see "5.7 Backing Up and Restoring ZFS Data" on page 74.

4.7.1 Preparing for Migration

Storage pools should be explicitly exported to indicate that they are ready to be migrated. This flushes any unwritten data to disk, write data to the disk indicating that the export was done, and remove all knowledge of the pool from the system.

If you do not explicitly export the pool, choosing instead to remove the disks manually, you can still import the resulting pool on another system. However, you may lose the last few seconds of data transactions, and the original machine will think the pool is faulted because the devices are no longer present. The destination machine will also refuse, by default, to import a pool that has not been explicitly exported. This is because the case is indistinguishable from network attached storage that is still in use on another system.

4.7.2 Exporting a Pool

To export a pool, use the `zpool export` command:

```
# zpool export tank
```

Once this command is executed, the pool `tank` is no longer visible on the system. The command attempts to unmount any mounted filesystems within the pool before continuing. If any of the filesystems fail to unmount, you can forcefully unmount them by using the `-f` flag:

```
# zpool export tank
cannot unmount '/export/home/eschrock': Device busy
# zpool export -f tank
```

If devices are unavailable at the time of export, the disks cannot be specified as cleanly exported. If one of these devices is later attached to a system without any of the working devices, it shows up as "potentially active". If there are emulated volumes in the pool that are in use, it cannot be exported, even with the `-f` option. To export a pool with an emulated volume, make sure all consumers of the volume are no longer active first.

For more information on emulated volumes, see "8.1 Emulated Volumes" on page 103.

4.7.3 Determining Available Pools to Import

Once the pool has been removed from the system (either though export or forcefully removing the devices), attach the devices to the target system. Although ZFS can cope with some situations where only a portion of the devices are available, in general all devices within the pool must be moved between the systems. The devices do not necessarily have to be attached under the same device name; ZFS detects any moved or renamed devices and adjusts the configuration appropriately. To discover available pools, run the `zpool import` command with no options:

```
# zpool import
  pool:  tank
    id:  3824973938571987430916523081746329
 state:  ONLINE
 action: The pool can be imported using its name or numeric identifier.  The
         pool may be active on on another system, but can be imported using
         the '-f' flag.
config:

        mirror              ONLINE
          c0t0d0            ONLINE
          c0t0d1            ONLINE
```

In the above example, the pool `tank` is available to be imported on the target system. Each pool is identified by a name as well as a unique numeric identifier. In the event that there are multiple available pools to import with the same name, the numeric identifier can be used to distinguish between them. The command attempts to display as much information as possible about the state of the devices in the pool. For example, if a pool appears to be in use on another system, or was not cleanly exported, a message similar to the following is displayed:

```
# zpool import
  pool: tank
    id: 3824973938571987430916523081746329
 state: DEGRADED
action: This pool appears to be in use or may not have been
        cleanly exported.  Use the '-f' flag to import this
        pool.
   see: http://www.sun.com/msg/ZFS-XXXX-13
config:

        mirror              ONLINE
          c0t0d0            ONLINE
          c0t0d1            ONLINE
```

Similar to `zpool status`, the `zpool import` command refers to a knowledge article available on the web with the most up-to-date information regarding repair procedures for this problem. In this case, the user can force the pool to be imported. Be warned: importing a pool that is currently in use by another system over a storage network can result in data corruption and panics as both systems attempt to write to the same storage. If some of the devices in the pool are not available but there is enough redundancy to have a usable pool, the pool appears in the `DEGRADED` state:

```
# zpool import
  pool: tank
    id: 3824973938571987430916523081746329
action: DEGRADED
  desc: This pool can be imported despite missing some
        devices.  The fault tolerance of the pool may
        be compromised.
   see: http://www.sun.com/msg/ZFS-XXXX-12
config:

        mirror              DEGRADED
          c0t0d0            ONLINE
          c0t0d1            FAULTED
```

In the above case, the second disk is damaged or missing, though you can still import the pool because the mirrored data is still accessible. If there are too many faulted or missing devices, the pool cannot be imported:

```
# zpool import
  pool: dozer
    id: 12938475986103486259473989087556 3
 state: FAULTED
action: This pool cannot be imported because the necessary
        devices are missing or damaged.  Attach the
        unavailable devices and try again.
   see: http://www.sun.com/msg/ZFS-XXXX-11
config:

        raidz               FAULTED
          c0t0d0            ONLINE
          c0t0d1            FAULTED
          c0t0d2            ONLINE
          c0t0d3            FAULTED
```

In this case, there are two disks missing from a RAID-Z virtual device, which means that there isn't sufficient replicated data available to reconstruct the pool. There are even some cases, where not enough devices are present to determine the complete configuration. In this case, ZFS doesn't know what other devices were part of the pool, though it does report as much information as possible about the situation:

```
# zpool import
  pool: tank
    id: 3824973938571987430916523081746329
 state: FAULTED
action: This pool cannot be imported because some
```

```
        devices are missing.  The following is a partial
        configuration.  Attach the additional unknown devices
        and try again.
   see: http://www.sun.com/msg/ZFS-XXXX-12
config:

        mirror              ONLINE
          c0t0d0            ONLINE
          c0t0d1            ONLINE

        Additional devices are known to be part of this pool,
        though their exact configuration cannot be determined.
```

4.7.4 Finding Pools From Alternate Directories

By default, `zpool import` only searches devices within the `/dev/dsk` directory. If you have devices in another directory, or are using pools backed by files, you will need to use the `-d` option to search different directories:

```
# zpool create dozer /file/a /file/b
# zpool export dozer
# zpool import
no pools available
# zpool import -d /file
  pool: dozer
    id: 672153753596386982
 state: ONLINE
action: The pool can be imported using its name or numeric identifier.
config:

        dozer       ONLINE
          /file/a   ONLINE
          /file/b   ONLINE
# zpool import -d /file dozer
```

If you have devices in multiple directories, multiple `-d` options can be specified.

4.7.5 Importing Pools

Once a pool has been identified for import, you can import it simply by specifying the name of the pool or its numeric identifier as an argument to the `zpool import` command:

```
# zpool import tank
```

If you have multiple available pools with the same name, you can specify which one to import using the numeric identifier:

```
# zpool import
  pool: dozer
    id: 2704475622193776801
 state: ONLINE
action: The pool can be imported using its name or numeric identifier.
config:

        dozer       ONLINE
          c1t9d0s0  ONLINE

  pool: dozer
    id: 6223921996155991199
 state: ONLINE
action: The pool can be imported using its name or numeric identifier.
config:

        dozer       ONLINE
          c1t8d0s0  ONLINE
# zpool import dozer
cannot import 'dozer': more than one matching pool
import by numeric ID instead
# zpool import 6223921996155991199
```

If the name conflicts with an existing pool, you can import it under a different name:

```
# zpool import dozer construct
```

This command imports the exported pool `dozer` using the new name `construct`. If the pool was not cleanly exported, ZFS requires the `-f` flag to prevent users from accidentally importing a pool that is still in use on another system:

```
# zpool import dozer
cannot import 'dozer': pool may be in use on another system
use '-f' to import anyway
# zpool import -f dozer
```

Pools can also be imported under an alternate root using the `-R` flag. For more information on alternate root pools, see "8.3 ZFS Alternate Root Pools" on page 108.

CHAPTER 5

Managing Filesystems

This chapter provides detailed information about managing ZFS filesystems. Included are such concepts as hierarchical filesystem layout, property inheritance, and automatic mount point management and share interactions.

A ZFS filesystem is a lightweight POSIX file system built on top of a storage pool. Filesystems can be dynamically created and destroyed without having to allocate or format any underlying space. Because filesystems are so lightweight, and because they are the central point of administration in ZFS, administrators are likely to create many of them.

ZFS filesystems are administered via the `zfs` command. The zfs(1M) command provides a set of subcommands which perform specific operations on filesystems. The following sections of this document describe these subcommands in detail. Snapshots, volumes, and clones are also managed through the use of zfs(1M), but will only be covered briefly in this section. For detailed information about snapshots and clones, see Chapter 6. For detailed information about volumes, see "8.1 Emulated Volumes" on page 103.

Note: The term *dataset* is used in this section as a generic term to refer to a filesystem, snapshot, clone, or volume.

The following sections are provided in this chapter.

5.1 Creating and Destroying Filesystems

Filesystems can be created and destroyed by using the `zfs create` and `zfs destroy` commands.

5.1.1 Creating a Filesystem

Filesystems are created by using the `zfs create` command. When creating a filesystem, the `create` subcommand takes a single argument: the name of the filesystem to create. The filesystem name is specified as a path name starting from the name of the pool: *pool_name/[filesystem_name/]filesystem_name*. The pool name and initial filesystem names in the path identify the location in the hierarchy where the new filesystem should be created. All the intermediate filesystem names must already exist in the pool. The last name in the path, identifies the name of the filesystem to be created. The filesystem name must satisfy the naming conventions defined in "1.3 ZFS Component Naming Conventions" on page 17.

The following example creates a filesystem named `bonwick` at `tank/home`.

```
# zfs create tank/home/bonwick
```

Upon successful creation, ZFS automatically mounts the newly created filesystem. By default, filesystems are mounted as */dataset*, using the path provided for the filesystem name in the `create` subcommand. In the above example, the newly created `bonwick` filesystem would be mounted at `/tank/home/bonwick`. For more information on auto-managed mount points, see "5.5.1 Managing Mount Points" on page 66.

5.1.2 Destroying a Filesystem

To destroy a filesystem, use the `zfs destroy` command. The destroyed filesystem is automatically unmounted and unshared. For more information on auto-managed mounts or auto-managed shares, see "5.5.1.1 Automatic Mount Points" on page 67. The following example destroys the `tabriz` filesystem.

```
# zfs destroy tank/home/tabriz
```

If the filesystem to be destroyed is busy (and so cannot be unmounted), the `zfs destroy` command will not succeed. To force the destruction of an active filesystem, the `-f` option must be used. This option should be used with caution as it can unmount, unshare, and destroy active filesystems, causing unexpected application behavior.

```
# zfs destroy tank/home/ahrens
cannot unmount 'tank/home/ahrens': Device busy

# zfs destroy -f tank/home/ahrens
```

The `zfs destroy` command also fails if a filesystem has children. To force a recursive destruction of a filesystem and all its descendants, the `-r` option must be used. Note that a recursive destroy also destroys snapshots. Great care should be taken when using this option.

```
# zfs destroy tank/ws
cannot destroy 'tank/ws': filesystem has children
use '-r' to destroy the following datasets:
tank/ws/billm
tank/ws/bonwick
tank/ws/maybee

# zfs destroy -r tank/ws
```

If the filesystem to be destroyed has indirect dependants, even the recursive destroy command described above fails. To force the destruction of *all* dependants, including cloned filesystems outside the target hierarchy, the `-R` option must be used. This option should be used with extreme caution.

```
# zfs destroy -r tank/home/schrock
cannot destroy 'tank/home/schrock': filesystem has dependant clones
use '-R' to destroy the following datasets:
tank/clones/schrock-clone

# zfs destroy -R tank/home/schrock
```

For more information about snapshots and clones, see Chapter 6.

5.1.3 Renaming a Filesystem

Filesystems can be renamed by using the `zfs rename` command. A rename can change the name of a filesystem and/or it can relocate the filesystem to a new location within the ZFS hierarchy. The first example below uses the `rename` subcommand to do a simple rename of a filesystem.

```
# zfs rename tank/home/kustarz tank/home/kustarz_old
```

The example above renames the `kustarz` filesystem to `kustarz_old`. The next example shows how to use `zfs rename` to relocate a filesystem.

```
# zfs rename tank/home/maybee tank/ws/maybee
```

In the above example, the `maybee` filesystem was relocated from `tank/home` to `tank/ws`. When relocating through rename, the new location must be within the same pool and it must have enough space to hold this new filesystem. If the new location does not have enough space, possibly because it has reached its quota (see "5.6 Quotas and Reservations" on page 72 for more information), the rename will fail.

The rename operation attempts an unmount/remount sequence for the filesystem and any descendant filesystems. The rename fails if it is unable to unmount an active filesystem. If this occurs, manual intervention is needed to force unmount the filesystem(s).

For information about renaming snapshots, see "6.1.1.1 Renaming ZFS Snapshots" on page 78.

5.2 ZFS Properties

Properties are the main mechanism used to control the behavior of filesystems, volumes, snapshots, and clones. Unless explicitly called out, the properties defined in the section apply to all the dataset types.

Properties are either readonly statistics, or settable properties. Most settable properties are also inheritable, where an inheritable property is one that, when set on a parent, is propagated down to all descendants.

All inheritable properties have an associated source. The source indicates how a property was obtained. It can have the following values:

`local` — A `local` source indicates that this property was explicitly set on this dataset using the `zfs set` command (described in "5.4.1 Setting Properties" on page 63).

`inherited from` *dataset-name* — A value of `inherited from` *dataset-name* means that this property was inherited from the named ancestor.

`default` — A value of `default` means that this property setting was not inherited nor set locally. This source is a result of no ancestor having this property as source `local`.

Name	Type	Default Value	Description
`aclinherit`	string	`secure`	Controls how ACL entries are inherited when files and directories are created. The values are: `discard`, `noallow`, `secure`, and `passthrough`. For a description of these values, see "7.1.3 ACL Property Modes" on page 88.
`aclmode`	string	`groupmask`	Controls how an ACL is modified during a `chmod(2)` operation. The values are: `discard`, `groupmask`, and `passthrough`. For a description of these values, see "7.1.3 ACL Property Modes" on page 88.

Name	Type	Default Value	Description
atime	boolean	on	Controls whether the access time for files is updated when they are read. See the description below.
available	number	N/A	The amount of space available to the dataset and all its children, assuming that there is no other activity in the pool. See the description below.
checksum	string	on	Controls the checksum used to verify data integrity. The default value is on, which automatically selects an appropriate algorithm, currently, fletcher2. The values are fletcher2, fletcher4, and sha256. A value of off disables integrity checking on user data; this is NOT recommended.
compression	string	off	Controls the compression algorithm used for this dataset. There is currently only one algorithm, lzjb, though this might change in future releases. This property can also be referred to by its shortened column name compress.
compressratio	number	N/A	The compression ratio achieved for this dataset, expressed as a multiplier. Compression can be turned on by running zfs set compression=on *dataset*.
creation	number	N/A	The date and time the dataset was created.
devices	boolean	on	Controls whether device nodes can be opened in the filesystem.
exec	boolean	on	Controls whether processes can be executed from within this filesystem.
mounted	boolean	N/A	For filesystems, indicates whether the filesystem is currently mounted. This property can be either yes or no.
mountpoint	string	See below	Controls the mountpoint used for this filesystem. See the description below.
origin	string	N/A	For cloned filesystems or volumes, the snapshot from which the clone was created. The origin cannot be destroyed (even with the -r or -f options) so long as a clone exists.

Name	Type	Default Value	Description
quota	number (or none)	none	Limits the amount of space a dataset and its descendents can consume. See the description below.
readonly	boolean	off	Controls whether dataset can be modified. This property can also be referred to by its shortened column name, rdonly.
recordsize	number	128K	Specifies a suggested block size for files in the file system. See the description below.
referenced	number	N/A	The amount of data accessible by this dataset, which may or may not be shared with other datasets in the pool. See the description below.
reservation	number (or none)	none	The minimum amount of space guaranteed to a dataset and its descendents. See the description below.
sharenfs	string	off	Controls whether the file system is shared via NFS, and what options are used. See the description below.
setuid	boolean	on	Controls whether the setuid bit is honored in the filesystem.
snapdir	string	visible	Controls whether the .zfs directory is hidden or visible in the root of the file system as discussed in the Snapshots section.
type	string	N/A	Identifies dataset type such as filesystem (filesystem/clone), volume, or snapshot.
used	number	N/A	The amount of space consumed by this dataset and all its descendants. See the detailed description below.
volsize	number	See below	For volumes, specifies the logical size of the volume. See the description below.
volblocksize	number	See below	For volumes, specifies the block size of the volume. See the description below.
zoned	boolean	See below	Controls whether the dataset is managed from a non-global zone. The default value is off. See the detailed description below.

5.2.1 Read-Only Properties

Read-only properties are properties that can be retrieved but cannot be set. Read-only properties are not inherited. Some properties are specific to a particular type of dataset; in such cases the particular dataset type is called out in the description.

- `available` — The amount of space available to the dataset and all its children, assuming no other activity in the pool. Because space is shared within a pool, this can be limited by any number of factors, including physical pool size, quotas, reservations, or other datasets within the pool.

  This property can also be referred to by its shortened column name, `avail`.

  For more information about space accounting, see "3.2 Space Accounting" on page 26.

  This property can also be referred to by its shortened column name, `avail`.
- `creation` — Date and time that this dataset was created.
- `mounted` — Indicates whether this filesystem, clone, or snapshot is currently mounted; does not apply to volumes.
- `origin` — For cloned filesystems only; the snapshot from which this clone originated. Non-cloned filesystems have an origin of none. The origin cannot be destroyed so long as a clone exists.
- `compressratio` — The compression ratio achieved on this dataset. Calculated from the logical size of all files and the amount of referenced physical data. Includes explicit savings through the use of the `compression` property.
- `referenced` — The amount of data accessible by this dataset, which may or may not be shared with other datasets in the pool. When a snapshot or clone is created, it initially references the same amount of space as the file system or snapshot it was created from, since its contents are identical. This property can also be referred to by its shortened column name, `refer`.
- `type` — Dataset type such as `filesystem` (filesystem/clone), `volume`, or `snapshot`
- `used` — The amount of space consumed by this dataset and all its descendants. This is the value that is checked against this dataset's quota and reservation. The space used does not include this dataset's reservation, but does take into account the reservations of any descendant datasets. The amount of space that a dataset consumes from its parent, as well as the amount of space that is freed if this dataset is recursively destroyed, is the greater of its space used and its reservation.

  When snapshots are created, their space is initially shared between the snapshot and the file system, and possibly with previous snapshots. As the file system changes, space that was previously shared becomes unique to the snapshot, and counted in the snapshot's space used. Additionally, deleting snapshots can increase the amount of space unique to (and used by) other snapshots. For more information about snapshots and space issues, see "3.3 Out of Space Behavior" on page 26.

The amount of space used, available, or referenced does not take into account pending changes. Pending changes are generally accounted for within a few seconds. Committing a change to a disk using `fsync(3c)` or `O_SYNC` does not necessarily guarantee that the space usage information will be updated immediately.

For more information on space accounting: including the `used`, `referenced`, and `available` properties listed above see "3.2 Space Accounting" on page 26.

5.2.2 Settable Properties

Settable properties are properties whose values can be both retrieved and set. Settable properties are set via the `zfs set` interface described in "5.4.1 Setting Properties" on page 63. With the exceptions of quotas and reservations, settable properties are inherited. For more information on quotas and reservations, see "5.6 Quotas and Reservations" on page 72.

Some settable properties are specific to a particular type of dataset; in such cases the particular dataset type is called out in the description field of the table. If not specifically mentioned, a property applies to all dataset types: filesystems, volumes, clones, and snapshots.

- `atime` — Controls whether the access time for files is updated when read. Turning this property off avoids producing write traffic when reading files and can result in significant performance gains, though it may confuse mailers and other similar utilities.
- `checksum` — Controls the checksum used to verify data integrity. The value `on` automatically selects an appropriate algorithm (currently `fletcher2`, but this may change in future releases). The value `off` disables integrity checking on user data; this is *not* recommended.
- `compression` — Controls the compression algorithm used for this dataset. The value `on` automatically selects an appropriate algorithm. There is currently only one algorithm, `lzjb`, though this may change in future releases.
- `devices` — Controls whether device nodes found within this filesystem can be opened.
- `exec` — Controls whether programs within this filesystem are allowed to be executed. Also, when set to `off`, `mmap(2)` calls with `PROT_EXEC` will be disallowed.
- `mountpoint` — Controls the mount point used for this filesystem. When the `mountpoint` property is changed for a filesystem, the filesystem and any children that inherit the mount point are unmounted. If the new value is `legacy`, then they remain unmounted. Otherwise, they are automatically remounted in the new location if the property was previously `legacy` or `none`, or if they were mounted before the property was changed. In addition, any shared file systems are unshared and shared in the new location.

For more information on using this property, see "5.5.1 Managing Mount Points" on page 66.

- `quota` — Limits the amount of space a dataset and its descendents can consume. This enforces a hard limit on the amount of space used. This includes all space consumed by descendents, including file systems and snapshots. Setting a quota on a descendent of a dataset that already has a quota does not override the ancestor's quota, but rather imposes an additional limit. Quotas cannot be set on volumes, as the `volsize` property acts as an implicit quota.

  For information about setting quotas, see "5.6.1 Setting Quotas" on page 72.

- `readonly` — Controls whether this dataset can be modified. When set to `on`, no modifications can be made to the dataset.

- `recordsize` — Specifies a suggested block size for files in the file system. This property is designed solely for use with database workloads that access files in fixed-size records. ZFS automatically tunes block sizes according to internal algorithms optimized for typical access patterns. For databases that create very large files but access them in small random chunks, these algorithms may be suboptimal. Specifying a `recordsize` greater than or equal to the record size of the database can result in significant performance gains. Use of this property for general purpose file systems is strongly discouraged, and may adversely affect performance. The size specified must be a power of two greater than or equal to 512 and less than or equal to 128 Kbytes. Changing the filesystem's `recordsize` only affects files created afterward; existing files are unaffected.

  This property can also be referred to by its shortened column name, `recsize`.

- `reservation` — The minimum amount of space guaranteed to a dataset and its descendents. When the amount of space used is below this value, the dataset is treated as if it were taking up the amount of space specified by its reservation. Reservations are accounted for in the parent datasets' space used, and count against the parent datasets' quotas and reservations. This property can also be referred to by its shortened column name, `reserv`.

  For more information, see "5.6.2 Setting Reservations" on page 73).

- `sharenfs` — Controls whether the filesystem is shared via NFS, and what options are used. A filesystem with a `sharenfs` property of `off` is managed through traditional tools such as share(1M), unshare(1M), and dfstab(4). Otherwise, the filesystem is automatically shared and unshared with the `zfs share` and `zfs unshare` commands. If the property is set to `on`, the share(1M) command is invoked with no options. Otherwise, the share(1M) command is invoked with options equivalent to the contents of this property. When the `sharenfs` property is changed for a dataset, the dataset and any children are re-shared with the new options, only if the property was previously `off`, or if they were shared before the property was changed. If the new property is `off`, the filesystems are unshared.

  For more information on sharing ZFS filesystems, see "5.5.5 Sharing ZFS File Systems" on page 70.

- `setuid` — Controls whether the set-UID bit is respected for the filesystem.

- `snapdir` — Controls whether the `.zfs` directory is hidden or visible in the root of the file system. For more information on using snapshots, see "6.1 ZFS Snapshots" on page 77.
- `volsize` — The logical size of the volume. By default, creating a volume establishes a reservation for the same amount. Any changes to `volsize` are reflected in an equivalent change to the reservation. These checks are used to prevent unexpected behavior for consumers. A volume which contains less space than it claims is available can result in undefined behavior or data corruption, depending on how the volume is used. These effects can also occur when the volume size is changed while it is in use (particularly when shrinking the size). Extreme care should be used when adjusting the volume size.

  Though not recommended, you can create a *sparse volume* by specifying the `-s` flag to `zfs create -V`, or by changing the reservation once the volume has been created. A sparse volume is defined as a volume where the reservation is not equal to the volume size. For a sparse volume, changes to `volsize` are not reflected in the reservation.

  For more information about using volumes, see "8.1 Emulated Volumes" on page 103.
- `volblocksize` — For volumes, specifies the block size of the volume. The blocksize cannot be changed once the volume has been written, so it should be set at volume creation time. The default blocksize for volumes is 8 Kbytes. Any power of 2 from 512 bytes to 128 Kbytes is valid.

  This property can also be referred to by its shortened column name, `volblock`.
- `zoned` — Indicates whether this dataset has been added to a non-global zone. If this is set, then the mount point is not respected in the global zone, and ZFS refuses to mount such a filesystem when asked. When a zone is first installed, this is set for any added filesystems. For more information on using ZFS with zones installed, see "8.2 Using ZFS on a Solaris System With Zones Installed" on page 104.

5.3 Querying Filesystem Information

The `zfs list` command provides an extensible mechanism for viewing and querying dataset information. Both basic and complex queries are explained in this section.

5.3.1 Listing Basic Information

Basic dataset information can be seen using `zfs list` with no options. This invocation will display the names of all datasets on the system include their `used`, `available`, `referenced`, and `mountpoint` properties. See "5.2 ZFS Properties" on page 54 for more information about these properties.

```
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                   84.0K  33.5G      -  /pool
pool/clone                 0  33.5G  8.50K  /pool/clone
pool/test                 8K  33.5G     8K  /test
pool/home              17.5K  33.5G  9.00K  /pool/home
pool/home/marks        8.50K  33.5G  8.50K  /pool/home/marks
pool/home/marks@snap       0      -  8.50K  /pool/hopme/marks@snap
```

The `zfs list` command can be refined to display specific datasets by providing the dataset name on the command line. Additionally, the `-r` option can be added to recursively display all descendants of that dataset. The example below uses `zfs list` to display `tank/home/chua` and all of its descendant datasets.

```
# zfs list -r tank/home/chua
NAME                         USED  AVAIL  REFER  MOUNTPOINT
tank/home/chua                  26.0K  4.81G  10.0K  /tank/home/chua
tank/home/chua/projects       16K  4.81G   9.0K  /tank/home/chua/projects
tank/home/chua/projects/fs1    8K  4.81G     8K  /tank/home/chua/projects/fs1
tank/home/chua/projects/fs2    8K  4.81G     8K  /tank/home/chua/projects/fs2
```

5.3.2 Complex Queries

The output displayed by `zfs list` can be customized through the use of the `-o`, `-f`, and `-H` options.

The `-o` option provides support for customizing the property value to be output. This option expects a comma separated list of the desired properties where any dataset property can be supplied as a valid value. See "5.2 ZFS Properties" on page 54 for a list of all supported dataset properties. In addition to the properties defined in the Properties section, the `-o` option list can also contain the literal `name` to indicate that the output should display the name of the dataset. The example below uses `zfs list` to display the dataset name along with the `sharenfs` and `mountpoint` properties.

```
# zfs list -o name,sharenfs,mountpoint
NAME                  SHARENFS         MOUNTPOINT
tank                  rw               /export
tank/archives         rw               /export/archives
tank/archives/zfs     rw               /export/archives/zfs
tank/calendar         off              /var/spool/calendar
tank/cores            rw               /cores
tank/dumps            rw               /export/dumps
tank/home             rw               /export/home
tank/home/ahl         rw               /export/home/ahl
tank/home/ahrens      rw               /export/home/ahrens
tank/home/andrei      rw               /export/home/andrei
tank/home/barts       rw               /export/home/barts
tank/home/billm       rw               /export/home/billm
tank/home/bjw         rw               /export/home/bjw
tank/home/bmc         rw               /export/home/bmc
tank/home/bonwick     rw               /export/home/bonwick
```

```
tank/home/brent       rw               /export/home/brent
tank/home/dilpreet    rw               /export/home/dilpreet
tank/home/dp          rw               /export/home/dp
tank/home/eschrock    rw               /export/home/eschrock
tank/home/fredz       rw               /export/home/fredz
tank/home/johansen    rw               /export/home/johansen
tank/home/jwadams     rw               /export/home/jwadams
tank/home/lling       rw               /export/home/lling
tank/home/mws         rw               /export/home/mws
tank/home/rab         rw               /export/home/rab
tank/home/sch         rw               /export/home/sch
tank/home/tabriz      rw               /export/home/tabriz
tank/home/tomee       rw               /export/home/tomee
```

The `-t` option provides the capability of specifying which type(s) of dataset(s) should be output. The valid types can be seen in the table below.

TABLE 5–1 Types of Datasets

Type	Description
`filesystem`	Display filesystem and clone datasets
`volume`	Display volume datasets
`snapshot`	Display snapshot datasets

The `-t` options takes a comma separated list of the types of datasets to be displayed. The example below uses the `-t` and `-o` options simultaneously to show the name and `used` property for all filesystems.

```
# zfs list -t filesystem -o name,used
NAME                  USED
pool                  105K
pool/container        0
pool/home             26.0K
pool/home/tabriz      26.0K
pool/home/tabriz_clone  0
```

The `-H` option can be used to omit the `zfs list` header from the generated output. When using the `-H` option, all white space is output as tabs. This option can be useful when parseable output is needed (for example, when scripting). The example below shows the output generated from an invocation of `zfs list` using the `-H` option.

```
# zfs list -H -o name
pool
pool/container
pool/home
pool/home/tabriz
pool/home/tabriz@now
pool/home/tabriz/container
pool/home/tabriz/container/fs1
pool/home/tabriz/container/fs2
pool/home/tabriz_clone
```

5.4 Managing Properties

Dataset properties are managed through the `set`, `inherit`, and `get` subcommands.

5.4.1 Setting Properties

The `zfs set` command can be used to modify any settable dataset property, see "5.2.2 Settable Properties" on page 58 for a list of settable dataset properties. The `zfs set` command takes a property/value sequence in the format of *property=value* and a dataset name. The example below sets the`atime` property to `off` for `tank/home`. Only one property can be set/modified per `zfs set` invocation.

```
# zfs set atime=off tank/home
```

Numeric properties can be specified using the following human-readable suffixes (in order of magnitude) `BKMGTPEZ`. Any of these suffixes can be followed by an optional `b`, indicating bytes, with the exception of the `B` suffix which already indicates bytes. The following four invocations of `zfs set` are equivalent numeric expressions indicating that the `quota` property should be set to the value of 50 gigabytes on the `tank/home/marks` filesystem.

```
# zfs set quota=50G tank/home/marks
# zfs set quota=50g tank/home/marks
# zfs set quota=50GB tank/home/marks
# zfs set quota=50gb tank/home/marks
```

Non-numeric properties are case sensitive and must be lower case, with the exception of `mountpoint` and `sharenfs` which may have mixed upper and lower case letters.

5.4.2 Inheriting Properties

All settable properties, with the exception of quotas and reservations, inherit their value from their parent, unless explicitly set by the child. If no ancestor has an explicit value set for an inherited property, the default value for the property is used. The `zfs inherit` command is used to clear a property setting, thus causing the setting to be inherited from the parent.

The following example uses`zfs set` to turn on compression, and then `zfs inherit` to unset the `compression` property, thus causing it to inherit the default setting of `off`. Note, since neither `home` nor `pool` have the compression property locally set (via `zfs set`), the default value is used. If both had it set, the value set in the most immediate ancestor would be used (`home` in this example).

```
# zfs set compression=on tank/home/bonwick
# zfs get -r compression tank
NAME             PROPERTY      VALUE                      SOURCE
```

```
tank              compression   off                          default
tank/home         compression   off                          default
tank/home/bonwick compression    on                           local
# zfs inherit compression tank/home/bonwick
# zfs get -r compression tank
NAME              PROPERTY      VALUE                        SOURCE
tank              compression   off                          default
tank/home         compression   off                          default
tank/home/bonwick compression   off                          default
```

The `inherit` subcommand is applied recursively when the `-r` option is specified. The following example causes the value for the compression property to be inherited by `tank/home` and any descendants it may have.

```
# zfs inherit -r compression tank/home
```

Be aware that the use of the `-r` option clears the current property setting for all descendant datasets.

5.4.3 Querying Properties

The simplest way to query property values is `zfs list` (see "5.3.1 Listing Basic Information" on page 60). However, for complicated queries and scripting, the `zfs get` subcommand can provide more detailed information in a customized format.

The `zfs get` subcommand can be used to retrieve any dataset property. The example below shows how to retrieve a single property on a dataset.

```
# zfs get checksum tank/ws
NAME                 PROPERTY      VALUE                     SOURCE
tank/ws              checksum      on                        default
```

The fourth column, `SOURCE`, indicates where this property value has been set from. The table below defines the meaning of the possible source values.

TABLE 5–2 Possible `SOURCE` Values (`zfs get`)

Value	Description
`default`	This property was never explicitly set for this dataset or any of its ancestors. The default value for this property is being used.
`inherited from` *dataset_name*	This property value is being inherited from the parent specified by *dataset_name*.
`local`	This property value was explicitly set, using `zfs set`, for this dataset.

TABLE 5–2 Possible `SOURCE` Values (`zfs get`) *(Continued)*

Value	Description
`temporary`	This property value was set using the `zfs mount -o` option and is only valid for the lifetime of the mount (for more information on temporary mount point properties, see "5.5.3 Temporary Mount Properties" on page 69).
- (none)	This is a read-only property, its value is generated by ZFS.

The special keyword `all` can be used to retrieve all dataset properties. The example below uses the `all` keyword to retrieve all existing dataset properties.

```
# zfs get all pool
NAME              PROPERTY       VALUE                      SOURCE
pool              type           filesystem                 -
pool              creation       Sat Nov 12 11:41 2005      -
pool              used           32K                        -
pool              available      33.5G                      -
pool              referenced     8K                         -
pool              compressratio  1.00x                      -
pool              mounted        yes                        -
pool              quota          none                       default
pool              reservation    none                       default
pool              recordsize     128K                       default
pool              mountpoint     /pool                      default
pool              sharenfs       off                        default
pool              checksum       on                         default
pool              compression    on                         local
pool              atime          on                         default
pool              devices        on                         default
pool              exec           on                         default
pool              setuid         on                         default
pool              readonly       off                        default
pool              zoned          off                        default
pool              snapdir        visible                    default
pool              aclmode        groupmask                  default
pool              aclinherit     secure                     default
```

The `-s` option to `zfs get` provides the ability to specify, by source value, the type of properties to be output. This option takes a comma separated list indicating the source types desired. Any property that does not have the specified source type isn't displayed. The valid source types are: `local`, `default`, `inherited`, `temporary`, and `none`. The example below shows all properties that have been locally set on pool.

```
# zfs get -s local all pool
NAME              PROPERTY      VALUE                     SOURCE
pool              compression   on                        local
```

Any of the above options can be combined with the `-r` option to recursively get the specified properties on all children of the specified dataset. The following example recursively retrieves all temporary properties on all datasets within `tank`.

```
# zfs get -r -s temporary all tank
NAME              PROPERTY       VALUE                      SOURCE
```

```
tank/home           atime          off                    temporary
tank/home/bonwick   atime          off                    temporary
tank/home/marks     atime          off                    temporary
```

5.4.4 Querying Properties for Scripting

The `zfs get` subcommand supports the `-H` and `-o` options. These options are designed for scripting. The `-H` indicates that any header information should be omitted and that all white space should come in the form of tabs; uniform white space allows for easily parseable data. The `-o` option allows the user to customize the output. This option takes a comma separated list of values to be output. All properties, defined in "5.2 ZFS Properties" on page 54, along with the literals `name`, `value`, `property` and `source` can be supplied in the `-o` list.

The following example shows how to retrieve a single value using the `-H` and `-o` options of `zfs get`.

```
# zfs get -H -o value compression tank/home
on
```

A `-p` option is supported that reports numeric values as their exact values. For example, 1 Mbyte would be reported as 1000000. This option can be used as follows:

```
# zfs get -H -o value -p used tank/home
182983742
```

The `-r` option along with any of the above options can be used to recursively get the requested value(s) for all descendants. The following example uses the `-r`, `-o`, and `-H` options to output the dataset name and the value of the `used` property for `export/home` and its descendants, while omitting any header output.

```
# zfs get -H -o name,value -r used export/home
export/home     5.57G
export/home/marks       1.43G
export/home/maybee      2.15G
```

5.5 Mounting and Sharing File Systems

This section describes how mount points and shared filesystems are managed in ZFS.

5.5.1 Managing Mount Points

By default, all ZFS filesystems are mounted by ZFS at boot via the `svc://system/filesystem/local` `smf(5)` service. Filesystems are mounted under */path*, where *path* is the name of the filesystem.

The default mount point can be overridden by setting the `mountpoint` property to a specific path using the `zfs set` command. ZFS automatically creates this mount point, if needed, and automatically mounts this filesystems when `zfs mount -a` is invoked, without having to edit the `/etc/vfstab` file.

The `mountpoint` property is inherited. For example, if `pool/home` has `mountpoint` set to `/export/stuff`, then `pool/home/user` inherits `/export/stuff/user` for `mountpoint`.

The `mountpoint` property can be set to `none` to prevent the filesystem from being mounted.

If desired, filesystems can also be explicitly managed through legacy mount interfaces by setting the `mountpoint` property to `legacy` via `zfs set`. Doing so prevents ZFS from automatically mounting and managing this filesystem; legacy tools including the `mount` and `umount` commands, and the `/etc/vfstab` file must be used instead. Legacy mounts are discussed in more detail below.

When changing mount point management strategies, the following behaviors apply:

5.5.1.1 Automatic Mount Points

- When changing from `legacy` or `none`, ZFS automatically mounts the filesystem.
- If ZFS is currently managing the filesystem but it is currently unmounted, and the `mountpoint` property is changed, the filesystem remains unmounted.

The default mount point for the root dataset can also be set at creation time by using `zpool create`'s `-m` option. For more information on creating pools, see "4.4.1 Creating a Pool" on page 32.

Any dataset whose `mountpoint` property is not `legacy` is managed by ZFS. The example below creates a dataset that is managed by ZFS.

```
# zfs create pool/filesystem
# zfs get mountpoint pool/filesystem
NAME              PROPERTY      VALUE                      SOURCE
pool/filesystem  mountpoint    /pool/filesystem           default
# zfs get mounted pool/filesystem
NAME              PROPERTY      VALUE                      SOURCE
pool/filesystem  mounted       yes                        -
```

The `mountpoint` property can also be explicitly set as shown in the example below.

```
# zfs set mountpoint=/mnt pool/filesystem
# zfs get mountpoint pool/filesystem
NAME              PROPERTY      VALUE                      SOURCE
pool/filesystem  mountpoint    /mnt                       local
# zfs get mounted pool/filesystem
NAME              PROPERTY      VALUE                      SOURCE
pool/filesystem  mounted       yes                        -
```

When the mountpoint property is changed, the filesystem is automatically unmounted from the old mount point and remounted to the new mount point. Mount point directories are created as needed. If ZFS is unable to unmount a filesystem, due to it being active, an error is reported and a forced manual unmount will be necessary.

5.5.1.2 Legacy Mount Points

Filesystems can be managed via legacy tools by setting the mountpoint property to legacy. Legacy filesystems must be managed through the mount and umount commands and the /etc/vfstab file. ZFS does not automatically mount legacy filesystems on boot, and the ZFS mount and umount command do not operate on datasets of this type. The examples below show how to set up and manage a ZFS dataset in legacy mode.

```
# zfs set mountpoint=legacy tank/home/eschrock
# mount -F zfs tank/home/eschrock /mnt
```

In particular, if you have set up separate ZFS /usr or /var file systems, you will need to indicate that they are legacy file systems and you must mount them by creating entries in the /etc/vfstab file. Otherwise, the system/filesystem/local service enters maintenance mode when the system boots.

To automatically mount a legacy filesystem on boot, an entry to the /etc/vfstab file must be added. The following example shows what the entry in the /etc/vfstab file might look like.

```
#device         device        mount           FS      fsck    mount   mount
#to mount       to fsck       point           type    pass    at boot options
#

tank/home/eschrock -        /mnt           zfs        -        yes        -
```

Note that the device to fsck and fsck pass entries are set to -. This is because the fsck(1M) command is not applicable to ZFS filesystems. See "1.1.2 Transactional Semantics" on page 14 for more information regarding data integrity and the lack of need for fsck in ZFS.

5.5.2 Mounting File Systems

ZFS automatically mounts filesystems on create or boot. Use of the zfs mount subcommand is only necessary when changing mount options or explicitly mounting or unmounting filesystems.

The zfs mount command with no arguments shows all currently mounted filesystems that are managed by ZFS. Legacy managed mount points are not displayed.

```
# zfs mount
tank                            /tank
tank/home                       /tank/home
```

```
tank/home/bonwick                /tank/home/bonwick
tank/ws                          /tank/ws
```

The `-a` option can be used to mount all ZFS managed filesystems. Legacy managed filesystems are not mounted.

```
# zfs mount -a
```

By default, ZFS does not allow mounting on top of a non-empty directory. To force a mount on top of a non-empty directory, the `-O` option must be used.

```
# zfs mount tank/home/alt
cannot mount '/export/home/alt': directory is not empty
use legacy mountpoint to allow this behavior, or use the -O flag
# zfs mount -O tank/home/alt
```

Legacy mount points must be managed through legacy tools. An attempt to use ZFS tools result in an error.

```
# zfs mount pool/home/billm
cannot mount 'pool/home/billm': legacy mountpoint
use mount(1M) to mount this filesystem
# mount -F zfs tank/home/billm
```

When a filesystem is mounted, it uses a set of mount options based on the property values associated with the dataset. The correlation between properties and mount options is as follows:

Property	Mount Options
`devices`	`devices/nodevices`
`exec`	`exec/noexec`
`readonly`	`ro/rw`
`setuid`	`setuid/nosetuid`

The mount option `nosuid` is an alias for `nodevices,nosetuid`.

5.5.3 Temporary Mount Properties

If any of the above options are set explicitly using the `-o` option at mount, the associated property value is temporarily overridden. These property values are reported as `temporary` by `zfs get` and revert back to their original settings when the filesystem is unmounted. If a property value is changed while the dataset is mounted, the change takes effect immediately, overriding any temporary setting.

The following example temporarily sets the read-only option on `tank/home/perrin`.

```
# zfs mount -o ro tank/home/perrin
```

The above example assumes that the filesystem is unmounted. To temporarily change a property on a filesystem that is currently mounted, the special `remount` option must be used. The following example temporarily changes the `atime` property to `off` for a filesystem that is currently mounted.

```
# zfs mount -o remount,noatime tank/home/perrin
# zfs get atime tank/home/perrin
NAME              PROPERTY      VALUE                      SOURCE
tank/home/perrin atime          off                        temporary
```

5.5.4 Unmounting File Systems

Filesystems can be unmounted by using the `zfs unmount` subcommand. The unmount command can take either the mount point or the filesystem name.

Unmounting by filesystem name.

```
# zfs unmount tank/home/tabriz
```

Unmounting by mount point.

```
# zfs unmount /export/home/tabriz
```

The unmount command fails if the filesystem is active or busy. To forceably unmount a filesystem, the `-f` option can be used. Care should be taken when forceably unmounting a filesystem, if its contents are actively being used, unpredictable application behavior can result.

```
# zfs unmount tank/home/eschrock
cannot unmount '/export/home/eschrock': Device busy
zfs unmount -f tank/home/eschrock
```

To provide for backwards compatibility, the legacy umount(1M) command can be used to unmount ZFS filesystems.

```
# umount /export/home/bob
```

5.5.5 Sharing ZFS File Systems

Like mountpoints, ZFS can automatically share filesystems through the use of the `sharenfs` property. Using this method, the administrator does not have to modify the `/etc/dfs/dfstab` file when a new filesystem is added. The `sharenfs` property is a comma-separated list of options to pass to the share(1M) command. The special value `on` is an alias for the default share options, which are `read/write` permissions for anyone. The special value `off` indicates that the filesystem is not managed by ZFS, and can be shared through traditional means such as the `/etc/dfs/dfstab` file. All filesystems whose `sharenfs` property is not `off` are shared during boot.

5.5.5.1 Controlling Share Semantics

By default, all filesystems are unshared. To share a new filesystem, run the following command:

```
# zfs set sharenfs=on tank/home/eschrock
```

The property is inherited, and filesystems are automatically shared on creation if their inherited property is not `off`. For example:

```
# zfs set sharenfs=on tank/home
# zfs create tank/home/bricker
# zfs create tank/home/tabriz
# zfs set sharenfs=ro tank/home/tabriz
```

Both `tank/home/briker` and `tank/home/tabriz` are initially shared writable since they inherit the `sharenfs` property from `tank/home`. Once the property is set to `ro` (readonly), `tank/home/tabriz` is shared readonly regardless of the `sharenfs` property set for `tank/home`.

5.5.5.2 Unsharing Filesystems

While most filesystems are automatically shared and unshared during boot, creation, and destruction, there are times when filesystems need to be explicitly unshared. To do this, use the `zfs unshare` command:

```
# zfs unshare tank/home/tabriz
```

This command unshares the `tank/home/tabriz` filesystem. To unshare all ZFS filesystems on the system, run:

```
# zfs unshare -a
```

5.5.5.3 Sharing Filesystems

As mentioned in the previous section, most of the time the automatic behavior of ZFS (sharing on boot and creation) should be sufficient for normal operation. If, for some reason, you unshare a filesystem, you can share it again with the `zfs share` command:

```
# zfs share tank/home/tabriz
```

You can also share all ZFS filesystems on the system:

```
# zfs share -a
```

5.5.5.4 Legacy Shares

If the `sharenfs` property is `off`, then ZFS does not attempt to share or unshare the filesystem at any time. This allows the filesystem to be administered through traditional means such as the `/etc/dfs/dfstab` file.

Unlike the traditional `mount` command, the traditional `share` and `unshare` commands can still function on ZFS filesystems. This means that it's possible to manually share a filesystem with options that are different from those of the `sharenfs` property. This administrative model is discouraged. You should choose to either manage NFS shares completely through ZFS or completely through the `/etc/dfs/dfstab` file. The ZFS administrative model is designed to be simpler and less work than the traditional model, but there are some cases where you may still want to control shares through the familiar model.

5.6 Quotas and Reservations

ZFS supports quotas and reservations at the filesystem level. Filesystem properties provide the ability to set a limit on the amount of space a filesystem can use by setting the `quota` property as well as the ability to guarantee some amount of space is available to a filesystem by setting the `reservation` property. Both of these properties apply to the dataset they are set on and all descendants of that dataset.

That is, if a quota is set on `tank/home` dataset, the total space used by `tank/home` *and all of its descendants* cannot exceed the quota. Similarly, if `tank/home` is given a reservation, `tank/home` *and all of its descendants* draw from that reservation. The amount of space used by a dataset (and all of its descendents) is reported by the `used` property.

5.6.1 Setting Quotas

ZFS quotas can be set and displayed with the `zfs set` and `zfs get` commands. The following example sets a quota of 10 Gbytes on `tank/home/bonwick`.

```
# zfs set quota=10G tank/home/bonwick
# zfs get quota tank/home/bonwick
NAME              PROPERTY      VALUE                     SOURCE
tank/home/bonwick quota         10.0G                     local
```

ZFS quotas also impact the output of the `zfs list` and `df` commands.

```
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
tank/home              16.5K  33.5G  8.50K  /export/home
tank/home/bonwick      15.0K  10.0G  8.50K  /export/home/bonwick
tank/home/bonwick/ws   6.50K  10.0G  8.50K  /export/home/bonwick/ws
# df -h /export/home/bonwick
Filesystem             size   used  avail capacity  Mounted on
tank/home/bonwick       10G     8K    10G     1%    /export/home/bonwick
```

Note that although `tank/home` has 33.5 Gbytes of space available, `tank/home/bonwick` and `tamk/home/bonwick/ws` only have 10 Gbytes of space available, due to the quota on `tank/home/bonwick`.

It is not possible to set a quota to an amount less than is currently being used by a dataset.

```
# zfs set quota=10K tank/home/bonwick
cannot set quota for 'tank/home/bonwick': size is less than current used or
reserved space
```

5.6.2 Setting Reservations

A ZFS reservation is an allocation of space from the pool that is guaranteed to be available to a dataset. As such, it is not possible to reserve space for a dataset if that space is not currently available in the pool. The total of all outstanding unconsumed reservations cannot exceed the amount of unused space in the pool. ZFS reservations can be set and displayed with the `zfs set` and `zfs get` commands.

```
# zfs set reservation=5G tank/home/moore
# zfs get reservation tank/home/moore
NAME             PROPERTY      VALUE                      SOURCE
tank/home/moore  reservation   5.00G                      local
```

ZFS reservations can influence the output of the `zfs list` command.

```
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
tank/home              5.00G  33.5G  8.50K  /export/home
tank/home/moore        15.0K  10.0G  8.50K  /export/home/moore
```

Note that `tank/home` shows that it is using 5 Gbytes of space, although the total space referred to by `tank/home` and its descendants is much less than 5 Gbytes. The used space reflects the space reserved for `tank/home/moore`. Reservations are accounted for in the used space of the parent dataset and do count against its quota and/or reservation.

```
# zfs set quota=5G pool/filesystem
# zfs set reservation=10G pool/filesysetm/user1
cannot set reservation for 'pool/filesystem/user1': size is greater than
available space
```

A dataset can use more space than its reservation, so long as there is space available in the pool that is unreserved and it is below its quota. A dataset cannot consume space that has been reserved for another dataset.

Reservations are not cumulative. That is, a second invocation of `zfs set` to set a reservation does not add its reservation to the existing one, rather it replaces it.

```
# zfs set reservation=10G tank/home/moore
# zfs set reservation=5G tank/home/moore
# zfs get reservation tank/home/moore
NAME             PROPERTY      VALUE                      SOURCE
tank/home/moore  reservation   5.00G                      local
```

5.7 Backing Up and Restoring ZFS Data

You can backup and restore ZFS filesystem snapshots and the original filesystems by using the `zfs backup` and `zfs restore` commands.

The following ZFS backup and restore solutions are provided:

- Creating ZFS snapshots and rolling back snapshots, if necessary.
- Creating full and incremental backups of ZFS snapshots and restoring the snapshots, if necessary.
- Remotely replicating ZFS file systems by backing up and restoring a ZFS snapshot and file system.

Consider the following when choosing a ZFS backup solution:

- File system snapshots and rolling back snapshots - Use the `zfs snapshot` and `zfs rollback` commands if you want to easily create a copy of a file system and revert back to a previous file system version, if necessary. For example, if you want to restore a file or files from a previous version of a file system.

  For more information about creating and rolling back to a snapshot, see "6.1 ZFS Snapshots" on page 77.
- Backing up file system snapshots - Use the `zfs backup` and `zfs restore` commands to back up and restore a ZFS file system snapshot. You can backup incremental changes between snapshots, but you cannot restore files individually. You must restore the entire file system snapshot.
- Remote replication - Use the `zfs backup` and `zfs restore` commands when you want to copy a file system from one system to another. This process is different from a traditional volume management product that might mirror devices across a WAN. There is no special configuration or hardware required. The advantage of replicating a ZFS file system is that you can recreate a file system on a storage pool

on another system, and specify different levels of configuration for the newly created pool, such as RAID-Z, but with identical file system data.

5.7.1 Backing Up ZFS Filesystems With Other Backup Products

In addition to the `zfs backup` and `zfs restore` commands, you can also use backup utilities, such as the `tar` and `cpio` commands, to back up ZFS files. All of these utilities backup and restore ZFS file attributes and ACLs. Check the appropriate options for both the `tar` and `cpio` commands.

Keep the following issues in mind when using other backup products to back up ZFS files:

- Veritas Backup software – You can use this product to back up ZFS files, but it silently ignores ACLs on ZFS files. (CR 6352899)
- Legato NetWorker™ software – Currently, this product cannot be used to backup ZFS files. (CR 6349974)

5.7.2 Backing Up a ZFS Snapshot

The simplest form of the `zfs backup` command is to backup a snapshot. For example:

```
# zfs backup tank/dana@111505 > /dev/rmt/0
```

You can create an incremental backup by using the `zfs backup -i` option. For example:

```
# zfs backup -i tank/dana@111505 tank/dana@now > /dev/rmt/0
```

Note that the first argument is the earlier snapshot and the second argument is the later snapshot.

If you need to store many backups, you might consider compressing a ZFS backup with `gzip`. For example:

```
# zfs backup pool/fs@snap | gzip > backupfile.gz
```

5.7.3 Restoring a ZFS Snapshot

When you restore a file system snapshot, the file system is restored as well. The file system is unmounted and is inaccessible while it is being restored. In addition, the original file system to be restored must not exist while it is being restored. If there is a conflicting filesystem name, `zfs rename` can be used to rename it. For example:

```
# zfs backup tank/gozer@111105 > /dev/rmt/0

.
.
.
# zfs restore tank/gozer2@today < /dev/rmt/0
# zfs rename tank/gozer tank/gozer.old
# zfs rename tank/gozer2 tank/gozer
```

When you restore an incremental file system snapshot, the most recent snapshot must first be rolled back. In addition, the destination file system must exist. To restore the previous incremental backup for `tank/dana`, for example:

```
# zfs rollback tank/dana@111505
cannot rollback to 'tank/dana@111505': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
tank/dana@now
# zfs rollback -r tank/dana@111505
# zfs restore tank/dana < /dev/rmt/0
```

During the incremental restore process, the filesystem is unmounted and cannot be accessed.

5.7.4 Remote Replication of a ZFS File System

You can also use the `zfs backup` and `zfs restore` commands to remotely copy a file system from one system to another system. For example:

```
# zfs backup tank/cindy@today | ssh newsys zfs restore sandbox/restfs@today
restoring backup of tank/cindy@today
        into sandbox/restfs@today ...
restored 17.8Kb backup in 1 seconds (17.8Kb/sec)
```

The syntax above backs up the`tank/cindy@today` snapshot and restores it into the `sandbox/restored` file system and also creates a `restfs@today` snapshot on the `newsys` system. This syntax assumes that the user has been configured to `ssh` on the remote system.

CHAPTER 6

ZFS Snapshots and Clones

This chapter describes how to create and manage ZFS snapshots and clones.

The following sections are provided in this chapter.

6.1 ZFS Snapshots

A snapshot is a read-only copy of a filesystem or volume. Snapshots can be created almost instantly, and, initially, consume no additional space within the pool. However, as data within the active dataset changes, the snapshot consumes space by continuing to reference the old data and so prevents it from being freed.

ZFS snapshots include the following features:

- Persistence across system reboots.
- Theoretical maximum number of snapshots is 2^{64}.
- Use no separate backing store. They consume space directly from the same storage pool as the file system from which they were created.

Snapshots of volumes cannot be accessed directly, but they can be cloned, backed up, rolled back to, and so on. For information on backing up a ZFS snapshot, see "5.7 Backing Up and Restoring ZFS Data" on page 74.

6.1.1 Creating and Destroying ZFS Snapshots

Snapshots are created by using the `zfs snapshot` command, which takes as its only argument the name of the snapshot to create. The snapshot name is specified as follows:

filesystem@snapname
volume@snapname

The snapshot name must satisfy the naming conventions defined in "1.3 ZFS Component Naming Conventions" on page 17.

The following example creates a snapshot of `tank/home/ahrens` that is named `friday`.

```
# zfs snapshot tank/home/ahrens@friday
```

Snapshots have no modifiable properties. Nor can dataset properties be applied to a snapshot.

```
# zfs set compression=on pool/home/ahrens@tuesday
cannot set compression property for 'pool/home/ahrens@tuesday': snapshot
properties cannot be modified
```

Snapshots are destroyed by using the `zfs destroy` command.

```
# zfs destroy tank/home/ahrens@friday
```

A dataset cannot be destroyed if snapshots of the dataset exists. For example:

```
# zfs destroy pool/home/ahrens
cannot destroy 'pool/home/ahrens': filesystem has children
use '-r' to destroy the following datasets:
pool/home/ahrens@tuesday
pool/home/ahrens@wednesday
pool/home/ahrens@thursday
```

In addition, if clones have been created from a snapshot, then they must be destroyed before the snapshot can be destroyed.

For more information on the `destroy` subcommand, see "5.1.2 Destroying a Filesystem" on page 52.

6.1.1.1 Renaming ZFS Snapshots

You can rename snapshots but they must be renamed within the pool and dataset from which they were created.

```
# zfs rename tank/home/cindys@111205 pool/home/cindys@today
```

The following snapshot rename operation is not supported.

```
# zfs rename tank/home/cindys@111205 pool/home/cindys@saturday
cannot rename to 'pool/home/cindys@today': snapshots must be part of same
dataset
```

6.1.2 Displaying and Accessing ZFS Snapshots

Snapshots of filesystems are accessible in the `.zfs/snapshot` directory within the root of the containing filesystem. For example, if `tank/home/ahrens` is mounted on `/home/ahrens`, then the `tank/home/ahrens@friday` snapshot data is accessible in the `/home/ahrens/.zfs/snapshot/friday` directory.

```
# ls /home/ahrens/.zfs/snapshot
tuesday wednesday thursday friday
```

Currently, the `.zfs/snapshot/` directories can only be accessed locally or over NFSv4. Accessing these directories over earlier NFS versions is not supported.

Snapshots can be listed as follows:

```
# zfs list -t snapshot
NAME                       USED  AVAIL  REFER  MOUNTPOINT
pool/home/ahrens@tuesday  13.3M      -  2.13G  -
```

6.1.2.1 Snapshot Space Accounting

When a snapshot is created, its space is initially shared between the snapshot and the filesystem, and possibly with previous snapshots. As the filesystem changes, space that was previously shared becomes unique to the snapshot, and thus is counted in the snapshot's `used` property. Additionally, deleting snapshots can increase the amount of space unique to (and thus *used* by) other snapshots.

A snapshot's space `referenced` property is the same as the filesystem's was when the snapshot was created.

6.1.3 Rolling Back to a Snapshot

The `zfs rollback` command can be used to discard all changes made since a specific snapshot. The filesystem reverts to its state at the time the snapshot was taken. By default, the command refuses to rollback to a snapshot other than the most recent one. To rollback to an earlier snapshot, all intermediate snapshots must be destroyed. You can destroy earlier snapshots by specifying the `-r` flag.

If there are clones of any intermediate snapshots, the `-R` flag must be specified to destroy the clones as well.

Note – The filesystem must be unmounted and remounted, if it is currently mounted. If the filesystem cannot be unmounted, the rollback fails. The `-f` flag forces the filesystem to be unmounted, if necessary.

The following example rolls back the `pool/home/ahrens` filesystem to the `tuesday` snapshot:

```
# zfs rollback pool/home/ahrens@tuesday
cannot rollback to 'pool/home/ahrens@tuesday': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
pool/home/ahrens@wednesday
pool/home/ahrens@thursday
# zfs rollback -r pool/home/ahrens@tuesday
```

6.2 ZFS Clones

A clone is a writable volume or filesystem whose initial contents are the same as another dataset. As with snapshots, creating a clone is nearly instantaneous, and initially consumes no additional space.

Clones can only be created from a snapshot. When a snapshot is cloned, it creates an implicit dependency between the clone and snapshot. Even though the clone is created somewhere else in the dataset hierarchy, the original snapshot cannot be destroyed as long as the clone exists. The `origin` property exposes this dependency, and the `zfs destroy` command lists any such dependencies, if they exist.

Clones do not inherit the properties of the dataset from which they are created. Rather, clones inherit their properties based on where they are created in the pool hierarchy. Use the `zfs get` and `zfs set` commands to view and change the properties of a cloned dataset. For more information about setting ZFS dataset properties, see "5.4.1 Setting Properties" on page 63.

Since a clone initially shares all its space with the original snapshot, its space `used` property is initially zero. As changes are made to the clone, it uses more space. The space `used` property of the original snapshot does not take into account the space consumed by the clone.

6.2.1 Creating a Clone

To create a clone use the `zfs clone` command, specifying the snapshot from which to create it, and the name of the new filesystem or volume. The new filesystem or volume can be located anywhere in the ZFS hierarchy. The type of the new dataset (for example, filesystem or volume) is the same as the snapshot from which it was created. The following example creates a new clone named `pool/home/ahrens/bug123` with the same initial contents as the snapshot `pool/ws/gate@yesterday`.

```
# zfs clone pool/ws/gate@yesterday pool/home/ahrens/bug123
```

The following example creates a cloned work space from the `projects/newproject@today` snapshot for a temporary user as `projects/teamA/tempuser` and then sets properties on the cloned work space.

```
# zfs snapshot projects/newproject@today
# zfs clone projects/newproject@today projects/teamA/tempuser
# zfs set sharenfs=on projects/teamA/tempuser
# zfs set quota=5G projects/teamA/tempuser
```

6.2.2 Destroying a Clone

ZFS clones are destroyed with the `zfs destroy` command.

```
# zfs destroy pool/home/ahrens/bug123
```

Clones must be destroyed before the parent snapshot can be destroyed.

CHAPTER 7

Using ACLs to Protect ZFS Files

This chapter provides information about using access control lists (ACLs) to protect your ZFS files by providing more granular permissions then the standard UNIX permissions.

The following sections are provided in this chapter.

7.1 New Solaris ACL Model

Previous versions of Solaris supported an ACL implementation that was primarily based on the POSIX ACL draft specification. The POSIX-draft based ACLs are used to protect UFS files and are translated by earlier versions of NFS prior to NFSv4.

With the introduction of NFSv4, a new ACL model is needed to fully support the interoperability that NFSv4 hopes to achieve between UNIX and non-UNIX clients. The new ACL implementation, as defined in the NFSv4 specification, provide much richer semantics that are based on NT-style ACLs.

The main differences of the new ACL model are as follows:

- Based on the NFSv4 specification and are similar to NT-style ACLs.
- Much more granular set of access privileges. For more information, see Table 7–2.
- Set and displayed with the `chmod` and `ls` commands rather than the `setfacl` and `getfacl` commands.
- Richer inheritance semantics for designating how access privileges are applied from directory to subdirectories, and so on. For more information, see "7.1.2 ACL Inheritance" on page 87.

The goal of both ACL models is to provide more fine grained access control than is available with the standard file permissions. Much like POSIX-draft ACLs, the new ACLs are made up of multiple Access Control Entries (ACEs).

POSIX-draft style ACLs use a single entry to define what is allowed and also what is denied for the user or group that the entry applies to. The new ACL model has two types of ACEs that play a role in access checking: ALLOW and DENY. This means that you can't infer from any single ACE that defines some set of permissions whether or not the permissions that weren't defined in that ACE are allowed or denied.

Translation between NFSv4-style ACLs and POSIX-draft ACLs is as follows:

- If you use any ACL-aware utility, such as the `cp`, `mv`, `tar`, `cpio`, or `rcp` commands, to transfer UFS files with ACLs to a ZFS file system, the POSIX-draft ACLs are translated into the equivalent NFSv4-style ACLs.
- NFSv4-style ACLs are not translated to POSIX-draft ACLs. You will see a message similar to the following:

```
# cp -p filea /var/tmp
cp: failed to set acl entries on /var/tmp/filea
```

- If you attempt to set a NFSv4–style ACL on a UFS file, you will see a message similar to the following:

```
chmod: ERROR: ACL type's are different
```

- If you attempt to set a POSIX-style ACL on a ZFS file, you will see messages similar to the following:

```
# getfacl filea
File system doesn't support aclent_t style ACL's.
See acl(5) for more information on Solaris ACL support.
```

7.1.1 ACL Format Description

The basic ACL format is as follows:

Syntax A:

ACL-entry-type=owner@, group@, everyone@:*access-permissions*/.../:deny | allow[:*inheritance-flags*]

Syntax B:

ACL-entry-type=user or group:*ACL-entry-ID=username or groupname*:*access-permissions*/.../:deny | allow[:*inheritance-flags*]

ACL-entry-type=owner@, group@, everyone@
Identifies owner@, group@, or everyone@ as displayed in Syntax A. For a description of *ACL-entry-types*, see Table 7–1.

ACL-entry-type=user or group:*ACL-entry-ID=username or groupname*
Identifies *user* or *group* as displayed in Syntax B. The user and group *ACL-entry-type* must also contains the *ACL-entry-ID*: *username* or *groupname*. For a description of *ACL-entry-types*, see Table 7–1.

access-permissions/.../
Identifies the access permissions that are granted or denied. For a description of ACL access permissions, see Table 7–2.

deny | allow
Identifies whether the access permissions are granted or denied.

inheritance-flags
Optional list of ACL inheritance flags. For a description of the ACL inheritance flags, see Table 7–3.

For example:

```
group@:write_data/append_data/execute:deny
```

In the above example, the *ACL-entry-ID* value is not relevant. The following example includes an *ACL-entry-ID* because a specific user (*ACL-entry-type*) is included in the ACL.

```
0:user:gozer:list_directory/read_data/execute:allow
```

When an ACL entry is displayed, it looks similar to the following:

```
2:group@:write_data/append_data/execute:deny
```

The **2** or the <*index-ID*> designation in the above example, identifies the ACL entry in the larger ACL, which might have multiple entries for owner, specific UIDs, group, and everyone. You can identify the *index-ID* with the `chmod` command to identify which part of the ACL you want to modify. For example, you can identify index ID 3 as `A#3` to the `chmod` command, similar to the following:

```
chmod A3=user:venkman:read_acl:allow filename
```

ACL entry types, which are the ACL representations of owner, group, and other, are described in the following table.

TABLE 7–1 ACL Entry Types

ACL Entry Types	Description
owner@	Specifies the access granted to the owner of the object.
group@	Specifies the access granted to the owning group of the object.
everyone@	Specifies the access granted to any user or group that does not match any other ACL entry.

TABLE 7–1 ACL Entry Types *(Continued)*

ACL Entry Types	Description
user	With a user name, specifies the access granted to an additional user of the object. Must include the *ACL-entry-ID*, which contains a *username* or *userID*. If the *username* can't be resolved to a UID, then the entry is assumed to be a numeric UID.
group	With a group name, specifies the access granted to an additional group of the object. Must include the *ACL-entry-ID*, which contains a *groupname* or *groupID*. If the *groupname* can't be resolved to a GID, then the entry is assumed to be a numeric GID.

Access privileges are described in the following table.

TABLE 7–2 ACL Access Privileges

Access Privilege	Description
add_file	Permission to add a new file to a directory.
add_subdirectory	Permission to create a subdirectory in a directory.
append_data	On a directory, permission to create a subdirectory. On a file, permission to modify the contents of a file.
delete	Permission to delete the file.
delete_child	Permission to delete a file or directory within a directory.
execute	Permission to execute a file.
list_directory	Permission to list the contents of a directory.
read_acl	Permission to read the ACL (ls).
read_attributes	The ability to read basic attributes (non-ACLs) of a file. Basic attributes can be thought of as the stat level attributes. Allowing this access mask bit would mean the entity can execute ls(1) and stat(2).
read_data	Permission to read the contents of the file.
read_xattrs	The ability to read the extended attributes of a file or do a lookup in the file's extended attributes directory.
synchronize	Placeholder, unimplemented at this time.

TABLE 7–2 ACL Access Privileges *(Continued)*

Access Privilege	Description
write_xattrs	The ability to create extended attributes or write to the extended attributes directory. Granting this permission to a user means that the user will be able to create an extended attribute directory for the file. The attribute file's permissions control the user's access to the attribute.
write_data	Permission to modify or replace the contents of a file.
write_attributes	Permission to change the times associated with a file or directory to an arbitrary value.
write_acl	Permission to write the ACL or the ability to modify the ACL with chmod(1).
write_owner	Permission to change the file's owner or group. Or, the ability to execute chown(1) or chgrp(1) on the file. Permission to take ownership of a file or the ability to change the group ownership of the file to a group of which the user is a member. If you wish to change the file or group ownership to an arbitrary user or group, then the PRIV_FILE_CHOWN privilege is required.

7.1.2 ACL Inheritance

The purpose of using ACL inheritance is for a newly created file or directory to inherit the ACLs they are intended to inherit, but without disregarding the existing permission bits on the parent directory.

By default, ACLs are not propagated. If you set an explicit ACL on a directory it is not inherited to any subsequent directory. You have to specify the inheritance of an ACL on a file or directory.

The optional inheritance flags are described in the following table.

TABLE 7–3 ACL Inheritance Flags

Inheritance Flag	Description
file_inherit	Inherit the ACL from the parent directory to the directory's files only.
dir_inherit	Inherit the ACL from the parent directory to the directory's subdirectories only.

TABLE 7–3 ACL Inheritance Flags *(Continued)*

Inheritance Flag	Description
inherit_only	Inherit the ACL from the parent directory but only applies to newly created files or subdirectories and not the directory itself. This flag requires either the file_inherit and/or dir_inherit flags to indicate what to inherit.
no_propagate	Inherit the ACL from the parent directory to the first-level contents of the directory only, not the second-level or subsequent contents. This flag requires either the file_inherit and/or dir_inherit flags to indicate what to inherit.

In addition, you can set a default ACL inheritance policy on the file system that is strict or less strict by using the `aclinherit` file system property. For more information, see the next section.

7.1.3 ACL Property Modes

The ZFS file system includes two properties related to ACLs:

- aclinherit – This property determines the behavior of ACL inheritance. Values include the following:
  - discard – For new objects, no ACL entries are inherited when a file or directory is created. The ACL on the file or directory will be equal to the permission mode of the file or directory.
  - noallow – For new objects, only inheritable ACL entries that have an access type of deny are inherited.
  - secure – For new objects, the `write_owner` and `write_acl` permissions are removed when an ACL entry is inherited.
  - passthrough – For new objects, the inheritable ACL entries are inherited with no changes made to the them. This mode, in effect, disables secure mode.

  The default mode is secure.
- aclmode – This property modifies ACL behavior whenever a file or directory's mode is modified by the `chmod` command or when a file is initially created. Values include the following:
  - discard – All ACL entries are removed except for those needed to define the mode of the file or directory.
  - groupmask – User or group ACL permissions are reduced so that they are no greater than the group permission bits unless it is a user entry that has the same UID as the owner of the file or directory. Then, the ACL permissions are reduced so that they are no greater than owner permission bits.
  - passthrough – For new objects, the inheritable ACL entries are inherited with no changes made to the them.

The default mode is groupmask.

7.2 Using ACLs on ZFS Files

As implemented with ZFS, ACLs are composed of an array of ACL entries. ZFS provides a *pure* ACL model, where all files have an ACL. Typically, the ACL is *trivial* in that it only represents the traditional UNIX `owner/group/other` entries.

ZFS files still have permission bits and a mode, but they are more of a cache of what the ACL represents. This means that if you change the permissions of the file, the file's ACL is updated accordingly. In addition, if you remove an explicit ACL that granted a user access to a file or directory, that user could still have access to the file or directory because of the file or directory's permission bits granting access to group or everyone. All access control decisions are governed by the permissions represented in a file or directory's ACL.

The primary rules of ACL access on a ZFS file are as follows:

- Each ACL entry is processed in order by ZFS.
- Only ACL entries that have a "who" that matches the requester of the access are processed.
- Each ACL entry is processed until all the bits of the access request have been allowed.
- Once an allow permission has been granted, it cannot be denied by a subsequent ACL deny entry in the same ACL permission set.
- The owner of the file is granted the `write_acl` permission unconditionally even if it is explicitly denied. Otherwise, any permission left unspecified is denied.

  In the cases of deny permissions or when an access permission is missing, the privilege subsystem determines what access request is granted for the owner of the file or for superuser. This mechanism prevents owners of files from getting locked out of their files and enables superuser to modify files for recovery purposes.

If you set an explicit ACL on a directory, the ACL is not automatically inherited to the directory's children. If you set an explicit ACL and you want it inherited to the directory's children, you have to use the ACL inheritance flags. For more information, see Table 7–3 and "7.3.1 Setting ACL Inheritance on ZFS Files" on page 96.

When a new file is created and depending on the `umask` value, a default trivial ACL is applied, which is similar to the following:

```
% ls -v file.1
-rw-r--r--   1 root     root        2703 Nov  4 12:37 file.1
     0:owner@:execute:deny
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
```

```
        /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
        /write_acl/write_owner:deny
     5:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
        :allow
```

Note that in the above example, each user category (owner@, group@, everyone@) has two ACL entries, which is one entry for deny permissions and one entry is for access permissions.

A description of this file ACL is as follows:

`0:owner@`	Owner is denied execute permissions to the file (`execute:deny`).
`1:owner@`	Owner can read and modify the contents of the file (`read_data/write_data/append_data`) and modify the file's attributes such as time stamps, extended attributes, and ACLs (`write_xattr/write_attributes /write_acl`). In addition, the owner is granted the ability to modify the ownership of the file (`write_owner:allow`)
`2:group@`	Group is denied modify and execute permissions to the file (`write_data/append_data/execute:deny`).
`3:group@`	Group is granted read permissions to the file (`read_data:allow`).
`4:everyone@`	Everyone who is not user or group is denied permission to execute or modify the contents of the file and to modify any attributes of the file (`write_data/append_data/write_xattr/execute/write_attributes/write_acl/write_owner:deny`).
`5:everyone@`	Everyone who is not user or group is granted read permissions to the file and the file's attributes (`read_data/read_xattr/read_attributes/read_acl/synchronize:allow`). The `synchronize` access permission is not currently implemented.

When a new directory is created and depending on the `umask` value, a default directory ACL is similar to the following:

```
$ ls -dv dir.1
drwxr-xr-x   2 root     root           2 Nov  1 14:51 dir.1
     0:owner@::deny
     1:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
        /append_data/write_xattr/execute/write_attributes/write_acl
        /write_owner:allow
     2:group@:add_file/write_data/add_subdirectory/append_data:deny
     3:group@:list_directory/read_data/execute:allow
     4:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
```

```
        /write_attributes/write_acl/write_owner:deny
    5:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
        /read_acl/synchronize:allow
```

A description of this directory ACL is as follows:

`0:owner@`	Owner deny list is empty for the directory (`::deny`).
`1:owner@`	Owner can read and modify the directory contents (`list_directory/read_data/add_file/write_data/add_subdirectory/append_data`), execute the file (`execute`), and modify the file's attributes such as time stamps, extended attributes, and ACLs (`write_xattr/write_attributes/write_acl`). In addition, the owner is granted the ability to modify the ownership of the directory (`write_owner:allow`).
`2:group@`	Group cannot add to or modify the directory contents (`add_file/write_data/add_subdirectory/append_data:deny`).
`3:group@`	Group can list and read the directory contents. In addition, group has execute permission to the directory contents. (`list_directory/read_data/execute:allow`).
`4:everyone@`	Everyone who is not user or group is denied permission to add to or modify the contents of the directory (`add_file/write_data/add_subdirectory/append_data`). In addition, the permission to modify any attributes of the directory is also denied. (`write_xattr /write_attributes/write_acl/write_owner:deny`).
`5:everyone@`	Everyone who is not user or group is granted read and execute permissions to the directory contents and the directory's attributes (`list_directory/read_data/read_xattr/execute/read_attributes/read_acl/synchronize:allow`). The `synchronize` access permission is not currently implemented.

7.3 Setting and Displaying ACLs on ZFS Files

You can use the `chmod` command to modify ACLs on ZFS files. The following `chmod` syntax for modifying ACLs uses *acl-specification* to identify the format of the ACL. For a description of *acl-specification*, see "7.1.1 ACL Format Description" on page 84.

- Adding ACL entries
  - Adding an ACL entry by index-ID

    `% chmod A`*index-ID+acl-specification filename*

    This syntax inserts the new ACL entry at the specified index-ID location.
  - Adding an ACL entry for a user

    `% chmod A+`*acl-specification filename*
- Removing ACL entries
  - Removing an ACL entry by index-ID

    `% chmod A`*index-ID- filename*
  - Removing an ACL entry by user

    `% chmod A-`*acl-specification filename*
  - Removing an ACL from a file

    `% chmod A-` *filename*
- Replacing an ACL entry

  `% chmod A`*index-ID=acl-specification filename*

  `% chmod A=`*acl-specification filename*

Note the space between the # (pound sign) and the *index-ID*.

ACL information can be displayed with the `ls -v` command.

EXAMPLE 7–1 Modifying Trivial ACLs on ZFS Files

The following section provides examples of setting and displaying trivial ACLs.

For example, given the following ACL on `file.1`:

```
# ls -v file.1
-rw-r--r--   1 root     root        2703 Nov  4 12:37 file.1
     0:owner@:execute:deny
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
     5:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

Change the `group@` permissions to `read_data/write_data`. For example:

```
# chmod A3=group@:read_data/write_data:allow file.1
# ls -v filea
-rw-r--r--   1 root     root        2703 Nov  4 12:37 file.1
     0:owner@:execute:deny
```

EXAMPLE 7–1 Modifying Trivial ACLs on ZFS Files *(Continued)*

```
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data/write_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
     5:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

Add `read_data/execute` permissions for the user `gozer` on the `test.dir` directory. For example:

```
# chmod A+user:gozer:read_data/execute:allow test.dir
# ls -dv test.dir
drwxr-xr-x+  2 root     root           2 Nov  4 11:10 test.dir
     0:user:gozer:list_directory/read_data/execute:allow
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

Remove `read_data/execute` permissions for user `gozer`. For example:

```
# chmod A0- test.dir
# ls -dv test.dir
drwxr-xr-x   2 root     root           2 Nov  4 11:10 test.dir
     0:owner@::deny
     1:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     2:group@:add_file/write_data/add_subdirectory/append_data:deny
     3:group@:list_directory/read_data/execute:allow
     4:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     5:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

EXAMPLE 7–2 ACL Interaction With Permissions on ZFS Files

The following ACL scenarios illustrate the interaction between setting explicit ACLs and then changing the file or directory's permission bits.

Given the following ACL on `file.2`:

EXAMPLE 7–2 ACL Interaction With Permissions on ZFS Files *(Continued)*

```
# ls -v file.2
-rw-r--r--   1 root     root      206663 Nov  4 12:41 file.2
     0:owner@:execute:deny
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
     5:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

Remove ACL allow permissions from everyone@. For example:

```
# chmod A5- file.2
# ls -v file.2
-rw-r-----   1 root     root      206663 Nov  4 12:41 file.2
     0:owner@:execute:deny
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
```

In the above output, the file's permission bits are reset from 655 to 650. You have effectively removed read permissions for other from the file's permissions bits when you removed the ACL allow permissions for everyone@.

Replace the existing ACL with `read_data/write_data` permissions for everyone@. For example:

```
# chmod A=everyone@:read_data/write_data:allow file.23
# ls -v file.3
-rw-rw-rw-+  1 root     root        2703 Nov  4 14:52 file.3
     0:everyone@:read_data/write_data:allow
```

In the above example, the `chmod` syntax effectively replaces the existing ACL with `read_data/write_data:allow` permissions to read/write permissions for owner, group, and other. In this model, everyone@ specifies access to any user or group. Since we do not have an owner@ or group@ ACL entry to override the permissions for owner and group, the permission bits are set to 666.

Replace the existing ACL with read permissions for user `gozer`. For example:

```
# chmod A=user:gozer:read_data:allow file.3
# ls -v file.3
----------+  1 root     root        2703 Nov  4 14:55 file.3
     0:user:gozer:read_data:allow
```

EXAMPLE 7–2 ACL Interaction With Permissions on ZFS Files *(Continued)*

Using the above syntax, the file permissions are computed to be 000 because no ACL entries exist for owner@, group@ or everyone@, which represent the traditional permission components of a file. As the owner of the file, you can resolve this by resetting the permissions (and the ACL) as follows:

```
# chmod 655 file.3
# ls -v file.3
-rw-r-xr-x+  1 root     root        2703 Nov  4 14:55 file.3
     0:user:gozer::deny
     1:user:gozer:read_data:allow
     2:owner@:execute:deny
     3:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     4:group@:write_data/append_data:deny
     5:group@:read_data/execute:allow
     6:everyone@:write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:deny
     7:everyone@:read_data/read_xattr/execute/read_attributes/read_acl
         /synchronize:allow
```

EXAMPLE 7–3 Removing Explicit ACLs on ZFS Files

You can use the `chmod` command to remove all explicit ACLs on a file or directory. For example, given the following ACL:

```
# ls -dv test5.dir
drwxr-xr-x+  2 root     root           2 Nov  4 14:22 test5.dir
     0:user:gozer:read_data:deny:file_inherit
     1:user:lp:read_data:allow:file_inherit
     2:owner@::deny
     3:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     4:group@:add_file/write_data/add_subdirectory/append_data:deny
     5:group@:list_directory/read_data/execute:allow
     6:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     7:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

Remove the explicit ACLs for users `gozer` and `lp`. The remaining ACL contains the default 6 values for owner@, group@, and everyone@.

```
# chmod A- test5.dir
ls -dv test5.dir
drwxr-xr-x+  2 root     root           2 Nov  4 14:22 test5.dir
         2:owner@::deny
     3:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     4:group@:add_file/write_data/add_subdirectory/append_data:deny
```

EXAMPLE 7–3 Removing Explicit ACLs on ZFS Files *(Continued)*

```
     5:group@:list_directory/read_data/execute:allow
     6:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     7:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

7.3.1 Setting ACL Inheritance on ZFS Files

By default, ACLs are not propagated through a directory structure. For example, an explicit ACL of `read_data/write_data/execute` is applied for user gozer on `test.dir`.

```
# chmod A+user:gozer:read_data/write_data/execute:allow test.dir
# ls -dv test.dir
drwxr-xr-x+  2 root     root           2 Nov  4 12:39 test.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute:allow
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

If a `test.dir` subdirectory is created, the ACE for user gozer is not propagated. User gozer would only have access to `sub.dir` if the permissions on `sub.dir` granted him access as the file owner, group member, or other.

```
# mkdir test.dir/sub.dir
# ls -dv test.dir/sub.dir
drwxr-xr-x   2 root     root           2 Nov  4 14:30 test.dir/sub.dir
     0:owner@::deny
     1:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     2:group@:add_file/write_data/add_subdirectory/append_data:deny
     3:group@:list_directory/read_data/execute:allow
     4:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     5:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

The following series of examples identify the file and directory ACEs applied when the `file_inherit` flag is set.

Add read_data/write_data permissions for files in the test.dir directory for user gozer so that he has read access on any newly created files. For example:

```
# chmod A+user:gozer:read_data/write_data:allow:file_inherit test2.dir
# ls -dv test2.dir
drwxr-xr-x+  2 root     root           2 Nov  4 14:33 test2.dir
     0:user:gozer:read_data/write_data:allow:file_inherit
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

Identify user gozer's permissions on the newly created test2.dir/file.2 file. The ACL inheritance granted, read_data:allow:file_inherit, means user gozer can read the contents of any newly created file.

```
# touch test2.dir/file.2
# ls -v test2.dir/file.2
-rw-r--r--+  1 root     root           0 Nov  4 14:33 test2.dir/file.2
     0:user:gozer:write_data:deny
     1:user:gozer:read_data/write_data:allow
     2:owner@:execute:deny
     3:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     4:group@:write_data/append_data/execute:deny
     5:group@:read_data:allow
     6:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
     7:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

Note that because the aclmode for this file is set to the default mode, groupmask, user gozer does not have write_data permission on file.2 because the group permission of the file does not allow it.

Note the inherit_only permission, which is applied when the file_inherit or dir_inherit flags are set, are used to propagate the ACL through the directory structure. This means user gozer is only granted/denied permission from the everyone@ permissions unless he is the owner of the file or a member of the owning group of the file. For example:

```
# mkdir test2.dir/subdir.2
# ls -dv test2.dir/subdir.2
drwxr-xr-x+  2 root     root           2 Nov  4 15:00 test2.dir/subdir.2
     0:user:gozer:read_data/write_data:allow:file_inherit/inherit_only
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
```

```
        /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
        /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
        /read_acl/synchronize:allow
```

The following series of examples identify the file and directory ACLs applied when both the `file_inherit` and `dir_inherit` flags are set.

In the following example, user `gozer` is granted read, write, and execute permissions that are inherited for newly created files and directories.

```
# chmod A+user:gozer:read_data/write_data/execute:allow:file_inherit/
dir_inherit test3.dir
# ls -dv test3.dir
drwxr-xr-x+  2 root     root           2 Nov  4 15:01 test3.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute:allow
        :file_inherit/dir_inherit
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
        /append_data/write_xattr/execute/write_attributes/write_acl
        /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
        /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
        /read_acl/synchronize:allow

 touch test3.dir/file.3
# ls -v test3.dir/file.3
-rw-r--r--+  1 root     root           0 Nov  4 15:01 test3.dir/file.3
     0:user:gozer:write_data/execute:deny
     1:user:gozer:read_data/write_data/execute:allow
     2:owner@:execute:deny
     3:owner@:read_data/write_data/append_data/write_xattr/write_attributes
        /write_acl/write_owner:allow
     4:group@:write_data/append_data/execute:deny
     5:group@:read_data:allow
     6:everyone@:write_data/append_data/write_xattr/execute/write_attributes
        /write_acl/write_owner:deny
     7:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
        :allow
```

In the above examples, since the permission bits of the parent directory for group and other deny write and execute permissions, user gozer is denied write and execute permissions. The default `aclmode` property is secure, which means `write_owner` and `write_acl` permissions are not inherited.

In the following example, user gozer is granted read, write, and execute permissions that are inherited for newly created files, but are not propagated to subsequent contents of the directory.

```
# chmod A+user:gozer:read_data/write_data/execute:allow:file_inherit/
no_propagate test4.dir
# ls -dv test4.dir
drwxr-xr-x+  2 root     root           2 Nov  4 15:04 test4.dir
     0:user:gozer:read_data/write_data/execute:allow:file_inherit/no_propagate
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

When a new subdirectory is created, user gozer's `read_data/write_data/execute` permission for files are not propagated to the new `sub4.dir` directory.

```
# mkdir test4.dir/sub4.dir
# ls -dv test4.dir/sub4.dir
drwxr-xr-x+  2 root     root           2 Nov  4 15:06 test4.dir/sub4.dir
     0:user:gozer:add_file/write_data:deny
     1:user:gozer:list_directory/read_data/add_file/write_data/execute:allow
     2:owner@::deny
     3:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     4:group@:add_file/write_data/add_subdirectory/append_data:deny
     5:group@:list_directory/read_data/execute:allow
     6:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     7:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

EXAMPLE 7–4 ACL Inheritance With ACL Mode Set to Passthrough

If the `aclmode` property on this file system is set to passthrough, then user gozer would inherit the ACL applied on `test4.dir` above for the newly created `file.4` as follows:

```
# zfs set aclmode=passthrough tank/cindy
# touch test4.dir/file.4
# ls -v test4.dir/file.4
-rw-r--r--+  1 root     root           0 Nov  4 15:09 test4.dir/file.4
     0:user:gozer:read_data/write_data/execute:allow
     1:owner@:execute:deny
     2:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     3:group@:write_data/append_data/execute:deny
     4:group@:read_data:allow
     5:everyone@:write_data/append_data/write_xattr/execute/write_attributes
```

EXAMPLE 7–4 ACL Inheritance With ACL Mode Set to Passthrough *(Continued)*

```
        /write_acl/write_owner:deny
     6:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
        :allow
```

The above output illustrates that the `read_data/write_data/execute:allow:file_inherit/dir_inherit` ACL that was set on the parent directory, `test4.dir`, is passed through to user gozer.

EXAMPLE 7–5 ACL Inheritance With ACL Mode Set to Discard

If the `aclmode` property on a file system is set to discard, then ACLs can be potentially discarded when the permission bits on a directory change. For example:

```
# zfs set aclmode=discard tank/cindy
# chmod A+user:gozer:read_data/write_data/execute:allow:dir_inherit test5.dir
# ls -dv test5.dir
drwxr-xr-x+  2 root     root           2 Nov  4 15:10 test5.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute:allow
        :dir_inherit
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
        /append_data/write_xattr/execute/write_attributes/write_acl
        /write_owner:allow
     3:group@:add_file/write_data/add_subdirectory/append_data:deny
     4:group@:list_directory/read_data/execute:allow
     5:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
        /write_attributes/write_acl/write_owner:deny
     6:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
        /read_acl/synchronize:allow
```

If, at a later time, you decide to tighten the permission bits on a directory, the explicit ACL is discarded. For example:

```
# chmod 744 test5.dir
# ls -dv test5.dir
drwxr--r--   2 root     root           2 Nov  4 15:10 test5.dir
     0:owner@::deny
     1:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
        /append_data/write_xattr/execute/write_attributes/write_acl
        /write_owner:allow
     2:group@:add_file/write_data/add_subdirectory/append_data/execute:deny
     3:group@:list_directory/read_data:allow
     4:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
        /execute/write_attributes/write_acl/write_owner:deny
     5:everyone@:list_directory/read_data/read_xattr/read_attributes/read_acl
        /synchronize:allow
```

EXAMPLE 7–6 ACL Inheritance With ACL Inherit Mode Set to Noallow

In the following example, two explicit ACLs with file inheritance are set: one allows `read_data` permission and one denies `read_data` permission.

```
# zfs set aclinherit=noallow tank/cindy
# chmod A+user:gozer:read_data:deny:file_inherit test6.dir
# chmod A+user:lp:read_data:allow:file_inherit test6.dir
# ls -dv test6.dir
drwxr-xr-x+  2 root     root           2 Nov  4 15:13 test6.dir
     0:user:lp:read_data:allow:file_inherit
     1:user:gozer:read_data:deny:file_inherit
     2:owner@::deny
     3:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
         /append_data/write_xattr/execute/write_attributes/write_acl
         /write_owner:allow
     4:group@:add_file/write_data/add_subdirectory/append_data:deny
     5:group@:list_directory/read_data/execute:allow
     6:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
         /write_attributes/write_acl/write_owner:deny
     7:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
         /read_acl/synchronize:allow
```

When a new file is created, the ACL that allows `read_data` permission is discarded.

```
# touch test6.dir/file.6
# ls -v test6.dir/file.6
-rw-r--r--   1 root     root           0 Nov  4 15:14 test6.dir/file.6
     0:user:gozer:read_data:deny
     0:owner@:execute:deny
     1:owner@:read_data/write_data/append_data/write_xattr/write_attributes
         /write_acl/write_owner:allow
     2:group@:write_data/append_data/execute:deny
     3:group@:read_data:allow
     4:everyone@:write_data/append_data/write_xattr/execute/write_attributes
         /write_acl/write_owner:deny
     5:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

CHAPTER 8

Advanced Topics

This chapter describes emulated volumes, using ZFS on a Solaris system with zones installed, and alternate root pools.

The following sections are provided in this chapter.

8.1 Emulated Volumes

An *emulated volume* is a dataset that represents a block device and can be used like any block device. ZFS volumes are identified as devices in the `/dev/zvol/{dsk,rdsk}/path` directory.

The following syntax creates a 5-Gbyte ZFS volume, `tank/vol`:

```
# zfs create -V 5gb tank/vol
```

When you create a volume, a reservation is automatically set to the initial size of the volume. The reservation size continues to equal the size of the volume so that unexpected behavior doesn't occur. For example, if the size of volume shrinks, data corruption might occur. This means you should be careful when changing the size of the volume.

If you are using a Solaris system with zones installed, you cannot create or clone a ZFS volume in a non-global zone. Any attempt to create or clone a volume from within a non-global zone fail. For information about using ZFS volumes in a global zone, see "8.2.3 Adding ZFS Volumes to a Non-Global Zone" on page 106.

8.1.1 Emulated Volumes as Swap or Dump Devices

To set up a swap area, create a ZFS volume of a specific size and then enable swap on that device. Do not swap to a file on a ZFS file system. A ZFS swap file configuration is not supported.

The following syntax adds the 5-Gbyte `tank/vol` volume as a swap device.

```
# swap -a /dev/zvol/dsk/tank/vol
# swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c0t0d0s1   32,33     16 1048688 1048688
/dev/zvol/dsk/tank/vol 254,1      16 10485744 10485744
```

Using a ZFS volume as a dump device is currently unsupported. Use the `dumpadm` command to setup a dump device as you would with a UFS file system.

8.2 Using ZFS on a Solaris System With Zones Installed

ZFS datasets can be added to a zone either as a generic filesystem, or as a delegated dataset.

Adding a filesystem allows the non-global zone to share space with the global zone, though the zone administrator cannot control properties on the underlying dataset or create new filesystems within the dataset. This is identical to adding any other type of filesystem to a zone, and should be used when the primary purpose is solely to share common space.

ZFS also allows datasets to be delegated to a non-global zone, giving complete control over the dataset and all its children to the zone administrator. The zone administrator can create and destroy filesystems within that dataset, and modify properties of the datasets. The zone administrator cannot affect datasets not added to the zone, and cannot exceed any top level quotas set on the exported dataset.

8.2.1 Adding File Systems to a Non-Global Zone

ZFS datasets should be added as generic file systems when the goal is solely to share space with the global zone. You can export a ZFS file system to a non-global zone by using the `add fs` command in zonecfg(1M):

As the global administrator in the global zone:

```
# zonecfg -z zion
zion: No such zone configured
```

```
Use 'create' to begin configuring a new zone.
zonecfg:zion> create
zonecfg:zion> add fs
zonecfg:zion:fs> set type=zfs
zonecfg:zion:fs> set special=tank/zone/zion
zonecfg:zion:fs> set dir=/export/shared
zonecfg:zion:fs> end
```

This syntax adds the ZFS filesystem `tank/zone/zion` to the zone `zion`, mounted at `/export/shared`. The `mountpoint` property of the dataset must be set to `legacy`, and the filesystem cannot already be mounted in another location. The zone administrator can create and destroy files within the filesystem. The filesystem cannot be re-mounted in a different location, nor can the administrator change properties on the filesystem such as atime, readonly, compression, etc. The filesystem appears in the `/etc/mnttab` file, and is present nor visible in zfs(1M) output. The global zone administrator is responsible for setting and controlling properties of the dataset.

For more information about the `zonecfg` command and about configuring resource types with `zonecfg`, see *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

8.2.2 Delegating Datasets to a Non-Global Zone

If the primary goal is to delegate administration of storage to a zone, then ZFS supports adding datasets to a non-global zone through use of the `add dataset` command in zonecfg(1M):

As the global administrator in the global zone:

```
# zonecfg -z zion
zion: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:zion> create
zonecfg:zion> add dataset
zonecfg:zion:dataset> set name=tank/zone/zion
zonecfg:zion:dataset> end
```

Unlike adding a filesystem, this syntax causes the ZFS dataset `tank/zone/zion` to be visible within the zone `zion`. The zone administrator is able to set properties on the dataset, as well as create children. It allows the zone administrator to take snapshots, create clones, and otherwise control the entire namespace below the added dataset.

For more information on what actions are allowed, see "8.2.5 Property Management Within a Zone" on page 106.

8.2.3 Adding ZFS Volumes to a Non-Global Zone

Emulated volumes cannot be added to a zone using `zonecfg`'s `add dataset` subcommand. If an attempt to add an emulated volume is detected, the zone refuses to boot. However, volumes can be added to a zone by using `zonecfg`'s `add dataset` subcommand. For example:

As the global administrator in the global zone:

```
# zonecfg -z zion
zion: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:zion> create
zonecfg:zion> add device
zonecfg:zion:device> set match=/dev/zvol/dsk/tank/vol
zonecfg:zion:device> end
```

This syntax exports the `tank/vol` emulated volume to the zone. Note that adding a raw volume to a zone has implicit security risks, even if the volume doesn't correspond to a physical device. In particular, the zone administrator could create malformed filesystems that would panic the system when a mount was attempted. For more information on adding devices to zones and the related security risks, see "8.2.6 Understanding the `zoned` Property" on page 107.

For more information about adding devices to zones, see *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

8.2.4 Using ZFS Storage Pools Within a Zone

ZFS storage pools cannot be created or modified within a zone. The delegated administration model centralizes control of physical storage devices within the global zone, and control of virtual storage to non-global zones. While a pool-level dataset can be added to a zone, any command that modifies the physical characteristics of the pool, such as creation, deletion, adding or removing devices, is not allowed from within a zone. Even if physical devices are added to a zone via `zonecfg`'s `add device` subcommand, or if files are used, the zpool(1M) command does not allow the creation of any new pools within the zone.

8.2.5 Property Management Within a Zone

Once a dataset is added to a zone, it allows a certain level of control for the zone administrator. When a dataset is added to a zone, all its ancestors are visible as read-only datasets, while the dataset itself is writable as are all its children. For example, if we had the following configuration:

```
global# zfs list -Ho name
tank
tank/home
```

```
tank/data
tank/data/matrix
tank/data/zion
tank/data/zion/home
```

If we added `tank/data/zion` to a zone, each dataset would have the following properties:

Dataset	Visible	Writable	Immutable Properties
tank	yes	no	-
tank/home	no	-	-
tank/data	yes	no	-
tank/data/matrix	no	-	-
tank/data/zion	yes	yes	sharenfs, zoned, quota, reservation
tank/data/zion/home	yes	yes	sharenfs, zoned

Note that every parent of `tank/zone/zion` is visible read-only, all children are writable, and those not part of the parent hierarchy are not visible at all. The zone administrator cannot change the `sharenfs` property, because non-global zones cannot act as NFS servers. Neither can the administrator change the `zoned` property, because it would expose a security risk as described in the next section.

Any other property can be changed, except for the added dataset itself, where the `quota` and `reservation` properties cannot be changed. This allows the global zone administrator to control the space consumption of all datasets used by the non-global zone.

In addition, the `sharenfs` and `mountpoint` properties cannot be changed by the global zone administrator once a dataset has been added to a non-global zone.

8.2.6 Understanding the `zoned` Property

When a dataset is added to a non-global zone, it must be specially marked so that certain properties are not interpreted within the context of the global zone. Once a dataset has been added to a non-global zone under the control of a zone administrator, its contents can no longer be trusted. As with any filesystem, there may be setuid binaries, symbolic links, or otherwise questionable contents that may adverse affect the security of the global zone. In addition, the `mountpoint` property cannot be interpreted in the context of the global zone, or else the zone administrator could affect the global zone's namespace. To address the latter, ZFS uses the `zoned` property to indicate that a dataset has been delegated to a non-global zone at one point in time.

The zoned property is a boolean value that is automatically turned on when a zone containing a ZFS dataset is first booted. An administrator should never need to manually turn this property on. If the zoned property is set, the dataset cannot be mounted or shared in the global zone, and is ignored when the zfs share -a command or the zfs mount -a command is executed. In the following example, tank/zone/zion has been added to a zone, while tank/zone/global has not:

```
# zfs list -o name,zoned,mountpoint -r tank/zone
NAME                  ZONED  MOUNTPOINT
tank/zone/global        off  /tank/zone/global
tank/zone/zion           on  /tank/zone/zion
# zfs mount
tank/zone/global           /tank/zone/global
tank/zone/zion             /export/zone/zion/root/tank/zone/zion
```

Note the difference between the mountpoint property and the directory where the tank/zone/zion dataset is currently mounted. The mountpoint property reflects the property as stored on disk, not where it is currently mounted on the system.

When a dataset is removed from a zone or a zone is destroyed, the zoned property is **not** automatically cleared. This is due to the inherent security risks associated with this tasks. Since an untrusted user has had complete access to the dataset and its children, the mountpoint property may be set to bad values, or setuid binaries may exist on the filesystems.

In order to prevent accidental security risks, the zoned property must be manually cleared by the administrator if you want to reuse the dataset in any way. Before setting the zoned property to off, you should make sure that the mountpoint property for the dataset and all its children are set to reasonable values, and that no setuid binaries exist or turn the setuid property off.

Once you have verified that there are no security vulnerabilities left, the zoned property can be turned off with the zfs set or zfs inherit commands. If the zoned property is turned off while a dataset is in use within a zone, the system might behave in unpredictable ways — only change the property if you are sure the dataset is no longer in use by a non-global zone.

8.3 ZFS Alternate Root Pools

When creating pools, the pool is intrinsically tied to the host system. The host system keeps knowledge about the pool, so that it can detect when the pool is otherwise unavailable. While useful for normal operation, this can prove a hindrance when booting from alternate media, or creating a pool on removable media. To solve this problem, ZFS has the notion of an 'alternate root' pool. An alternate root pool does not persist across system reboots, and all mountpoints are modified to be relative to the root of the pool.

8.3.1 Creating ZFS Alternate Root Pools

The most common use for creating an alternate root pool is for use on removable media. In these circumstances, the user typically wants a single filesystem, and they want it to be mounted wherever they choose on the target system. When an alternate root pool is created using the `-R` option, the mount point of the root filesystem automatically is set to /,, which is the equivalent of the alternate root itself.

```
# zpool create -R /mnt morpheus c0t0d0
# zfs list morpheus
NAME                    USED  AVAIL  REFER  MOUNTPOINT
morpheus               32.5K  33.5G     8K  /mnt/morpheus
```

Note that there is a single filesystem (morpheus) whose mount point is the alternate root of the pool, `/mnt`. It is important to note that the mount point as stored on disk is really /, and that the full path to `/mnt` is interpreted only by nature of the alternate root. This filesystem can then be exported and imported using under an arbitrary alternate root on a different system.

8.3.2 Importing Alternate Root Pools

Pools can also be imported using an alternate root. This allows for recovery situations, where the mount points should not be interpreted in context of the current root, but under some temporary directory where repairs can be performed. This also can be used when mounting removable media as described above. The usage is similar to the create case:

```
# zpool import -R /mnt morpheus
# zpool list morpheus
NAME                     SIZE    USED   AVAIL    CAP  HEALTH     ALTROOT
morpheus                33.8G   68.0K   33.7G     0%  ONLINE     /mnt
# zfs list morpheus
NAME                    USED  AVAIL  REFER  MOUNTPOINT
morpheus               32.5K  33.5G     8K  /mnt/morpheus
```

8.4 ZFS Rights Profiles

If you want to perform ZFS management tasks without using the superuser (root) account, you will need to assume a role with either of the following profiles to perform ZFS administration tasks:

- ZFS Storage Management - Ability to create, destroy, and manipulate devices within a ZFS storage pool
- ZFS Filesystem Management - Ability to create, destroy, and modify ZFS filesystems

For more information about creating or assigning roles, see *System Administration Guide: Security Services*.

CHAPTER 9

Troubleshooting and Data Recovery

This chapter describes how to identify ZFS failure modes and how to recover from them. Steps for preventing failures are covered as well.

The following sections are provided in this chapter.

9.1 ZFS Failure Modes

As a combined file system and volume manager, there are many different failure modes that ZFS can exhibit. Before going into detail about how to identify and repair specific problems, it is important to describe some of the failure modes and how they manifest themselves under normal operation. This chapter will begin by outlining the various failure modes, then discuss how to identify them on a running system, and finally how to repair the problems. There are three basic types of errors. It is important to note that a single pool can be suffering from all three errors, so a complete repair procedure will involve finding and correcting one error, proceeding to the next, etc.

9.1.1 Missing Devices

If a device is completely removed from the system, ZFS detects that it cannot be opened and places the device in the `FAULTED` state. Depending on the data replication level of the pool, this may or may not result in the entire pool becoming unavailable. If one disk out of a mirror or RAID-Z device is removed, the pool will continue to be accessible. If all components of a mirror are removed, more than one device in a RAID-Z device is removed, or a single-disk, top-level device is removed, the pool will become `FAULTED`, and no data will be accessible until the device is re-attached.

9.1.2 Damaged Devices

The term 'damaged' covers a wide variety of possible errors. Examples include transient I/O errors due to a bad disk or controller, on-disk data corruption due to cosmic rays, driver bugs resulting in data being transferred to/from the wrong location, or simply another user overwriting portions of the physical device by accident. In some cases these errors are transient, such as a random I/O error while the controller was having problems. In other cases the damage is permanent, such as on-disk corruption. Even still, whether or not the damage is permanent does not necessarily indicate that the error is likely to occur again. For example, if an administrator accidentally overwrote part of a disk, it does not indicate any type of hardware failure has occurred, and the device should not be replaced. Identifying exactly what went wrong with a device is not an easy task, and is covered in more detail in a later section.

9.1.3 Corrupted Data

Data corruption occurs when one or more device errors (missing or damaged devices) affects a top level virtual device. For example, one half of a mirror can experience thousands of device errors without ever causing data corruption. If an error is encountered on the other side of the mirror in the exact same location, it will result in corrupted data. Data corruption is always permanent, and requires special consideration when repairing. Even if the underlying devices are repaired or replaced, the original data is lost forever. Most often this will require restoring data from backups. Data errors are recorded as they are encountered, and can be controlled through regular disk scrubbing, explained below. When a corrupted block is removed, the next scrubbing pass will notice that the corruption is no longer present and remove any trace of the error from the system.

9.2 Checking Data Integrity

There is no fsck(1M) equivalent for ZFS. This utility has traditionally served two purposes:

9.2.1 Data Repair

With traditional filesystems, the way in which data is written is inherently vulnerable to unexpected failure causing data inconsistencies. Since the filesystem is not transactional, it is possible to have unreferenced blocks, bad link counts, or other inconsistent data structures. The addition of journalling does solve some of these problems, but can introduce additional problems when the log cannot be rolled. With ZFS, none of these problems exist. The only way for there to be inconsistent data on disk is through hardware failure (in which case the pool should have been replicated), or a bug in the ZFS software. Given that fsck(1M) is designed to repair known pathologies specific to individual filesystems, it is not possible to write such a utility for a filesystem with no known pathologies. Future experience may prove that certain data corruption problems are common enough and simple enough such that a repair utility can be developed, but these problems can always be avoided by using replicated pools.

If your pool is not replicated, there is always the chance that data corruption can render some or all of your data inaccessible.

9.2.2 Data Validation

The other purpose that fsck(1M) serves is to validate that there are no problems with the data on disk. Traditionally, this is done by unmounting the filesystem and running the fsck(1M) utility, possibly bringing the system down to single user mode in the process. This results in downtime that is proportional to the size of the filesystem being checked. Instead of requiring an explicit utility to do the necessary checking, ZFS provides a mechanism to do regular checking of all data in the background, while the filesystem is in use. This functionality, known as *scrubbing*, is commonly used in memory and other systems as a method of detecting and preventing errors before it results in hardware or software failure.

9.2.3 Controlling Data Scrubbing

Whenever ZFS encounters an error, either through scrubbing or when accessing a file on demand, the error is logged internally so that the administrator can get a quick overview of all known errors within the pool. By default, no background scrubbing is

done due to the increased I/O load that may negatively impact performance. Only those errors encountered during normal operation are recorded. But ZFS does allow the administrator to control background scrubbing such that additional errors can be detected and either automatically repaired or reported in the error logs.

9.2.3.1 Explicit Scrubbing

The simples way to perform a check of your data integrity is to initiate an explicit scrub of all data within the pool. This will traverse all the data in the pool exactly once and verify that all blocks can be read. It will proceed as fast as the devices allow, though the priority of any I/O will remain below that of normal operations. This may negatively impact performance, though the filesystem should remain usable and nearly as responsive while the scrub is happening. To kick off an explicit scrub, use the `zpool scrub` command:

```
# zpool scrub tank
```

The status of the current scrub can be seen in `zpool status` output:

```
# zpool status -v
  pool: tank
 state: ONLINE
 scrub: scrub completed with 0 errors on Tue Nov 15 14:31:51 2005
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror    ONLINE       0     0     0
            c1t0d0  ONLINE       0     0     0
            c1t1d0  ONLINE       0     0     0
```

Note that there can only be one active scrubbing operation per pool. If you initiate an explicit scrub while a background scrub (explained next) is in progress, it will bump the priority of the existing scrubbing operation to complete as soon as possible.

For more information on interpreting `zpool status` output, see "4.6 Querying Pool Status" on page 39.

9.2.3.2 Background Scrubbing

In addition to administrator-requested scrubbing operations, ZFS also supports the ability to perform regular data scrubbing as a background operation. This accomplishes two basic tasks:

1. It automatically initiates any self-healing process for any corrupted data in the pool. This will pro-actively prevent any such errors from propagating further, by reducing the time window in which a parallel error could occur on another device. It will catch cases of bit rot and driver bugs that are not detected when the data is written.

2. It allows the administrator to instantly know all corrupted data in a pool without having to schedule an explicit scrub, which takes up I/O bandwidth and requires the administrator to wait for it to complete.

To enable background scrubbing, use the `zpool set` command:

```
# zpool set scrub=2w
```

The parameter is a time duration indicating how often a complete scrub should be performed. In this case, the administrator is requesting that the pool be scrubbed once every two weeks. ZFS automatically tries to schedule I/O to even distribute the work over the two week period, decreasing the performance impact an explicit scrub would have on the system.

Determining an appropriate scrub interval can be difficult and requires evaluation of I/O bandwidth and data integrity requirements. A good first estimate is probably once a month. Abnormal data corruption, while it occurs often enough to be dealt with, should not be a common phenomenon unless multiple devices in the pool are experience hardware failures. A relatively low priority scrub across a long period of time is probably sufficient to catch errors in a reasonably sized pool.

You can view the effectiveness of any background scrubbing through the use of the `zpool status` command. This commands show the time taken to do the last scrub.

```
# zpool status tank
  pool: tank
 state: ONLINE
 scrub: scrub completed with 0 errors on Tue Nov 15 14:31:51 2005
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror    ONLINE       0     0     0
            c1t0d0  ONLINE       0     0     0
            c1t1d0  ONLINE       0     0     0
```

Performing regular scrubbing also guarantees continuous I/O to all disks on the system. Regular scrubbing has the side effect of preventing power management from placing idle disks in low-power mode. If the system is generally performing I/O all the time, or if power consumption is not a concern, then this can safely be ignored. If the system is largely idle, and you want to conserve power to the disks, you should consider a cron(1M) scheduled explicit scrub rather than background scrubbing. This will still perform complete scrubs of data, though it will only generate a large amount of I/O until the scrubbing is finished, at which point the disks can be power managed as normal. The downside (besides increased I/O) is that there will be large periods of time when no scrubbing is being done at all, potentially increasing the risk of corruption during those periods.

For more information on setting scrubbing intervals, see "9.2.3.2 Background Scrubbing" on page 114.

9.2.3.3 Scrubbing and Resilvering

When a device is replaced, it initiates a resilvering operation to move data from the good copies to the new device. This is a form of disk scrubbing, and therefore only one such action can happen at a given time in the pool. If a scrubbing operation (either explicit or background) is in progress, a resilvering operation will suspend the current scrub, and start again after the resilvering is complete.

9.3 Identifying Problems

All ZFS troubleshooting is centered around the `zpool status` command. This command analyzes the various failures seen in the system and identify the most severe problem, presenting the user with a suggested action and a link to a knowledge article for more information. It is important to note that the command only identifies a single problem with the pool, though multiple problems can exist. For example, data corruption errors always imply that one of the devices has failed. Replacing the failed device will not fix the corruption problems.

This chapter describes how to interpret `zpool status` output in order to diagnose the type of failure and direct the user to one of the following sections on how to repair the problem. While most of the work is done automatically by the command, it is important to understand exactly what problems are being identified in order to diagnose the type of failure.

9.3.1 Determining if Problems Exist

The easiest way to determine if there are any known problems on the system is to use the `zpool status -x` command. This command only describes pools exhibiting problems. If there are no bad pools on the system, then the command displays a simple message:

```
# zpool status -x
all pools are healthy
```

Without the `-x` flag, the command displays complete status for all pools (or the requested pool if specified on the command line), even if the pools are otherwise healthy.

For more information on command line options to the `zpool status` command, see "4.6 Querying Pool Status" on page 39.

9.3.2 Understanding `zpool status` Output

A complete `zpool status` output looks similar to the following:

```
# zpool status tank
  pool: tank
 state: ONLINE
reason: Data corruption detected.
action: Remove corrupted data or restore from backup.
   see: http://www.sun.com/msg/ZFS-XXXX-09
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  ONLINE       0     0     0
          mirror              ONLINE       0     0     0
            c0t0d2            ONLINE       0     0     0
            c0t0d1            ONLINE       0     0     0

 scrub: ...
 errors: 4 errors detected.  Use '-v' for a complete list.
```

This output is divided into 4 basic sections:

9.3.2.1 Overall Status Information

This header section contains the following fields, some of which are only displayed for pools exhibiting problems:

pool — The name of the pool

state — The current health of the pool. This refers only to the ability of the pool to provide the necessarily replication level. Pools which are `ONLINE` may still have failing devices or data corruption present.

reason — A human readable description of what is wrong with this pool. This field is omitted if there are no problems.

action — A recommended action for repairing the errors. This is an abbreviated form directing the user to one of the following sections. This field is omitted if no problems are found.

see — A reference to a knowledge article containing detailed repair information. This online article is updated more often than this guide can be updated, and should always be referenced for the for the most up to date repair procedures. This field is omitted if no problems are found.

9.3.2.2 Configuration Information

The `config` field describes the configuration layout of the devices comprising the pool, as well as their state and any errors seen from the devices. The state can be one of `ONLINE`, `FAULTED`, `DEGRADED`, or `OFFLINE`. If it is anything but `ONLINE`, it indicates that the fault tolerance of the pool has been compromised.

The second section of the configuration output displays error statistics. These errors are divided into three categories:

READ I/O error while issuing a read request.

WRITE I/O error while issuing a write request.

CKSUM Checksum error. The device returned corrupted data as the result of a read request.

These errors can be used to determine if the damage is permanent. A small number of I/O errors may indicate a temporary outage, while a large number may indicate a permanent problem with the device. These errors do not necessarily correspond to data corruption as seen by applications. If the device is in a redundant configuration, the disk devices may show uncorrectable errors, while no errors appear at the mirror or RAID-Z device level. If this is the case, then ZFS successfully retrieved the good data, and attempted to heal the damaged data from existing replicas. For more information on interpreting these errors to determine device failure, see "9.6.1 Determining Type of Failure" on page 121.

Finally, additional auxiliary information is displayed in the last column of output. This expands on the `state` field, aiding in diagnosis of failure modes. If a device is `FAULED`, this field indicates whether it is because the device is inaccessible or whether the data on the device is corrupted. If the device undergoing resilvering, this field displays the current progress. For more information on monitoring resilvering progress, see "9.6.3.4 Viewing Resilvering Status" on page 124.

9.3.2.3 Scrubbing Status

The third section of output describes the current status of any explicit or background scrubs. This information is orthogonal to whether any errors are detected on the system, though it can be used to determine how accurate the data corruption error reporting is. If the last scrub ended recently, it is fairly certain that any known data corruption has been discovered.

For more information on data scrubbing and how to interpret this information, see "9.2 Checking Data Integrity" on page 113.

9.3.2.4 Data Corruption Errors

The status also shows whether there are any known errors associated with the pool. These errors may have been found while scrubbing the disks, or during normal operation. ZFS keeps a persistent log of all data errors associated with the pool. This log is rotated whenever a complete scrub of the system finishes. These errors are always fatal — their presence indicates that at least one application experienced an I/O error due to corrupt data within the pool. Device errors within a replicated pool do not result in data corruption, and are not recorded as part of this log. By default, only the number of errors found is displayed. A complete list of errors and their specifics can be found using the `-v` option.

For more information on interpreting data corruption errors, see "9.7.1 Identifying Type of Data Corruption" on page 126.

9.3.3 System Messaging

In addition to persistently keeping track of errors with the pool, ZFS also displays syslog messages when events of interest occur. The following scenarios generate events to notify the administrator:

- Device state transition — If a device becomes `FAULTED`, ZFS logs a message indicating that the fault tolerance of the pool may be compromised. A similar message is sent if the device is later brought online, restoring the pool to health.
- Data corruption — If any data corruption is seen, ZFS logs a message describing when and where the corruption was seen. This message is only logged the first time it is seen; subsequent accesses do not generate a message.

It is important to note that device errors are not mentioned here. If ZFS detects a device error and automatically recovers from it, there is no notification for the administrator. This is because such errors do not constitute a failure in the pool redundancy or data integrity, and because they are typically the result of a driver problem accompanied by its own set of error messages.

9.4 Damaged Configuration

ZFS keeps a cache of active pools and their configuration on the root filesystem. If this file is corrupted or somehow becomes out of sync with what is stored on disk, the pool can no longer be opened. ZFS does everything possible to avoid this situation, though arbitrary corruption is always possible given the qualities of the underlying filesystem and storage. This typically results in a pool 'disappearing' from the system when it should otherwise be available, though it can also manifest itself as a 'partial' configuration which is missing an unknown number of top level virtual devices. In either case, the configuration can be recovered by exporting the pool (if its visible at all), and re-importing it.

For more information on importing and exporting pools, see "4.7 Storage Pool Migration" on page 45.

9.5 Repairing a Missing Device

If a device cannot be opened, it displays as `UNAVAILABLE` in the `zpool status` output. This means that ZFS was unable to open the device when the pool was first accessed, or the device has since become unavailable. If the device causes a top level virtual device to be unavailable, then nothing in the pool can be accessed. Otherwise, the fault tolerance of the pool may be compromised. In either case, the device simply needs to be reattached to the system in order to restore normal operation.

9.5.1 Physically Reattaching the Device

Exactly how to reattach a missing device depends completely on the device in question. If the device is a network attached drive, connectivity should be restored. If the device is a USB or other removable media, it should be reattached to the system. If the device is a local disk, a controller may have died such that the device is no longer visible to the system. In this case, the controller should be replaced at which point the disks should again be available. Other pathologies exist and depend on the type of hardware and its configuration. If a drive fails such that it is no longer visible to the system (an unlikely event), it should be treated as a damaged device and follow the procedures outlined in "9.6 Repairing a Damaged Device" on page 121.

9.5.2 Notifying ZFS of Device Availability

Once the device is reattached to the system, ZFS may or may not automatically detect its availability. If the pool was previously faulted, or the system is rebooted as part of the attach procedure, then it automatically rescans all devices when it tries to open the pool. If the pool was degraded and the device was replaced while the system was up, the administrator will need to notify ZFS that the device is now available and should be reopened. This can be done using the `zpool online` command:

```
# zpool online tank c0t1d0
```

For more information on onlining devices, see "4.5.2.2 Bringing a Device Online" on page 38.

9.6 Repairing a Damaged Device

This section describes how to determine device failure types, clearing transient errors, and replacing a device.

9.6.1 Determining Type of Failure

The term 'damaged device' is rather vague, and can describe a number of possible situations:

- Bit rot — Over time, random events (such as magnetic influences and cosmic rays) can cause bits stored on disk to flip in unpredictable events. These events are relatively rare, but common enough to cause potential data corruption in large or long-running systems. These errors are typically transient.
- Misdirected reads or writes — Firmware bugs or hardware faults can cause reads or writes of entire blocks to reference the incorrect location on disk. These errors are typically transient, though a large number may indicate a faulty drive.
- Administrator error — Administrators can unknowingly overwrite portions of the disk with bad data (such as dd(1)ing /dev/zero over portions of the disk) that cause permanent corruption on disk. These errors are always transient.
- Temporary outage — A disk may become unavailable for a period time, causing I/Os to fail. This is typically associated with network attached devices, though local disks can experience temporary outages as well. These errors may or may not be transient.
- Bad or flaky hardware — This is a catch-all for the various problems that bad hardware exhibits. This could be consistent I/O errors, faulty transports causing random corruption, or any number of failures. These errors are typically permanent.
- Offlined device — If the device is offline, it is assumed that the administrator placed it in this state because it is presumed faulty. Whether or not this is true can be determined by the administrator who placed it in this state.

Determining exactly what is wrong can be a difficult and arduous process. The first step is to examine the error counts in `zpool status` output:

```
# zpool status pool -v
```

The errors are divided into I/O errors and checksum errors, which may indicate the possible failure type. Typical operation predicts a very small number of errors (just a few over long periods of time). If you are seeing large numbers of errors, then it probably indicates impending or complete device failure (although the pathology for

administrator error can result in large error counts). The other source of information is the system log. If there are a large number of messages from the SCSI or fibre channel driver, then it probably indicates serious hardware problems. If there are no syslog messages whatsoever, then the damage is likely transient.

The goal is to answer the following question:

Is another error likely to be seen on this device?

Those errors that were one-of-a-kind are considered "transient", and do not indicate potential failure. Those errors which are persistent, or severe enough to indicate potential hardware failure, are considered "fatal". The act of determining the type of error is beyond the scope of any automated software currently available with ZFS, and so much be done manually by the administrator. Once the determination is made, the appropriate action can be taken: either clear the transient errors or replace the device due to fatal errors. These repair procedures are described in the next sections.

Note even if the device errors are considered transient, it still may have caused uncorrectable data errors within the pool. These errors require special repair procedures, even if the underlying device is deemed healthy or otherwise repaired. For more information on repairing data errors, see "9.7 Repairing Damaged Data" on page 126.

9.6.2 Clearing Transient Errors

If the errors seen are deemed transient, in that they are unlikely to effect the future health of the device, then the device errors can be safely cleared to indicate that there was no fatal error. To clear a device of any errors, simply online the device using the `zpool online` command:

```
# zpool online tank c1t0d0
```

This syntax clears any errors associated with the device.

For more information on onlining devices, see "4.5.2.2 Bringing a Device Online" on page 38.

9.6.3 Replacing a Device

If device damage is permanent, or future permanent damage is likely, the device needs to be replaced. Whether or not the device can be replaced depends on the configuration.

9.6.3.1 Determining if a Device can be Replaced

In order for a device to be replaced, the pool must be in the `ONLINE` state, and the device must be part of a replicated configuration, or it must be healthy (in the `ONLINE` state). If the disk is part of a replicated configuration, there must be sufficient replicas from which to retrieve good data. If two disks in a four-way mirror are faulted, then either can be replaced since there are healthy replicas. On the other hand, if two disks in a four-way RAID-Z device are faulted, then neither can be replaced since there are not enough replicas from which to retrieve data. If the device is damaged but otherwise online, it can be replaced as long as the pool is not in the `FAULTED` state, though any bad data on the device is copied to the new device unless there are sufficient replicas with good data. In the following configuration:

```
mirror            DEGRADED
    c0t0d0             ONLINE
    c0t0d1             FAULTED
```

The disk `c0t0d1` can be replaced, and any data in the pool is copied from the good replica, `c0t0d0`. The disk `c0t0d0` can also be replaced, though no self-healing of data can take place since there is no good replica available. In the following configuration:

```
raidz             FAULTED
    c0t0d0             ONLINE
    c0t0d1             FAULTED
    c0t0d2             FAULTED
    c0t0d3             ONLINE
```

Neither of the faulted disks can be replaced. The `ONLINE` disks cannot be replaced either, since the pool itself is faulted in this case. In the following configuration:

```
c0t0d0          ONLINE
c0t0d1          ONLINE
```

Either top level disk can be replaced, though any bad data present on the disk is copied to the new disk. If either disk were faulted, then no replacement could be done since the pool itself would be faulted.

9.6.3.2 Unreplaceable Devices

If a loss of device causes the pool to become faulted, or the device contains too many data errors in an unreplicated configuration, then it cannot safely be replaced. Without sufficient replicas, there is no good data with which to heal the damaged device. In this case, the only option is to destroy the pool and recreate the configuration, restoring your data in the process.

For more information on restoring an entire pool, see "9.7.3 Repairing Pool Wide Damage" on page 128.

9.6.3.3 Replacing a Device

Once it has been determined that a device can be replaced, simply use the `zpool replace` command. If you are replacing the damaged device with another different device, use the following command:

```
# zpool replace tank c0t0d0 c0t0d1
```

This command begins migrating data to the new device from the damaged device, or other devices in the pool if it is in a replicated configuration. When it is finished, it detaches the damaged device from the configuration, at which point it can be removed from the system. If you have already removed the device and replaced it with a new device in the same location, use the single device form of the command:

```
# zpool replace tank c0t0d0
```

This command takes an unformatted disk, formats it appropriately, and then begins resilvering data from the rest of the configuration.

For more information on the `zpool replace` command, see "4.5.3 Replacing Devices" on page 39.

9.6.3.4 Viewing Resilvering Status

The process of replacing a drive can take an extended period of time, depending on the size of the drive and the amount of data in the pool. The process of moving data from one device to another is known as *resilvering*, and can be monitored via the `zpool status` command. Traditional filesystems resilver data at the block level. Since ZFS eliminates the artificial layering of the volume manager, it is capable of performing resilvering in a much more powerful and controlled manner. The two main advantages are:

- ZFS only resilvers the minimum amount of data necessary. In the case of a short outage (as opposed to a complete device replacement), the entire disk can be resilvered in a matter of minutes (or seconds), rather than having to resilver the whole disk, or complicate matters with "dirty region" logging that some volume managers support. When replacing a whole disk, this means that the resilvering process takes time proportional to the amount of data used on disk — replacing a 500GB disk can take seconds if there is only a few gigabytes of used space in the pool.
- Resilvering is interruptible and safe. If the system loses power or is rebooted, the resilvering process will resume exactly where it left off, without need for administrator intervention.

To view the resilvering process, use the `zpool status` command:

```
# zpool status tank
  pool: tank
 state: DEGRADED
```

```
reason: One or more devices is being resilvered.
action: Wait for the resilvering process to complete.
   see: http://www.sun.com/msg/ZFS-XXXX-08
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  DEGRADED     0     0     0
          mirror              DEGRADED     0     0     0
            replacing         DEGRADED     0     0     0  52% resilvered
              c0t0d0          ONLINE       0     0     0
              c0t0d2          ONLINE       0     0     0  21GB/40GB ETA 0:13
            c0t0d1            ONLINE       0     0     0

 scrub: none requested
```

In the above example, the disk `c0t0d0` is being replaced by `c0t0d2`. This can be seen with the introduction of the *replacing* virtual device in the configuration. This is not a real device, nor is it possible for the user to create a pool using this virtual device type. Its purpose is solely to display the resilvering process, and to identify exactly which device is being replaced.

Note that any pool currently undergoing resilvering is placed in the `DEGRADED` state, because the pool is not capable of providing the desired replication level until the resilvering process is complete. Resilvering always proceeds as fast as possible, though the I/O is always be scheduled with lower priority than user-requested I/O, to minimize impact on the system. Once the resilvering is complete, the configuration reverts to the new, complete, config:

```
# zpool status tank
  pool: tank
 state: ONLINE
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  ONLINE       0     0     0
          mirror              ONLINE       0     0     0
            c0t0d2            ONLINE       0     0     0
            c0t0d1            ONLINE       0     0     0

 scrub: scrub completed with 0 errors on Tue Nov 15 14:31:51 2005
errors: No data errors detected.
```

The pool is once again `ONLINE`, and the original bad disk (`c0t0d0`) has been removed from the configuration.

9.7 Repairing Damaged Data

ZFS uses checksumming, replication, and self-healing data to minimize the chances of data corruption. Even still data corruption can occur if the pool isn't replicated, corruption occurred while the pool was degraded, or an unlikely series of events conspired to corrupt multiple copies of a piece of data. Regardless of the source, the result is the same: the data is corrupted and therefore no longer accessible. The action taken depends on the type of data being corrupted, and its relative value. There are two basic types of data that can be corrupted:

- Pool metadata. ZFS requires a certain amount of data to be parsed in order to open a pool and access datasets. If this data is corrupted, it will result in the entire pool becoming unavailable, or complete portions of the dataset hierarchy being unavailable.
- Object data. In this case, the corruption is within a specific file or directory. This may result in a portion of the file or directory being inaccessible, or it may cause the object to be broken altogether.

Data is verified during normal operation as well as through scrubbing. For more information on how to verify the integrity of pool data, see "9.2 Checking Data Integrity" on page 113.

9.7.1 Identifying Type of Data Corruption

By default, the `zpool status` command shows only the fact that corruption has occurred, without specifics on where this corruption was seen:

```
# zpool status tank
  pool: tank
 state: ONLINE
reason: Data corruption detected.
action: Remove corrupted data or restore from backup.
   see: http://www.sun.com/msg/ZFS-XXXX-09
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  ONLINE       0     0     0
          mirror              ONLINE       0     0     0
            c0t0d2            ONLINE       0     0     0
            c0t0d1            ONLINE       0     0     0

 scrub: ...
 errors: 4 uncorrectable errors seen.  Use 'zpool status -v' for
        a complete list.
```

With the `-v` option, a complete list of errors is given:

```
# zpool status -v tank
  pool: tank
 state: ONLINE
reason: Data corruption detected.
action: Remove corrupted data or restore from backup.
   see: http://www.sun.com/msg/ZFS-XXXX-09
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  ONLINE       0     0     0
          mirror              ONLINE       0     0     0
            c0t0d2            ONLINE       0     0     0
            c0t0d1            ONLINE       0     0     0

 scrub: ...
errors: TYPE   OBJECT                         DATE
        file   /home/eschrock/.vimrc          12:03 Oct 2, 2005
        file   10$10cde24756492342            12:04 Oct 2, 2005
        dir    /export/ws/bonwick/current     3:05 Oct 3, 2005
        meta   12$010ceefde12a5856            13:45 Oct 17, 2005
```

The command attempts to determine the pathname for a given object, though this may not always succeed. For the case where the path cannot be determined, or if the error is within metadata not corresponding to a particular file, the numeric object ID is displayed. This does not help in determining the exact location of failure, though it may help support engineers diagnose the failure pathology to determine if it is a software bug. Each error is also displayed with a date identifying the last time the error was seen. This may have been part of a scrubbing operation, or when a user last tried to access the file.

Each error indicates only that an error was seen at the given point in time. It does not necessarily mean that the error is still present on the system. Under normal circumstances, this will always be true. Certain temporary outages may result in data corruption which is automatically repaired once the outage ends. A complete scrub of the pool (either explicitly or scheduled) is guaranteed to examine every active block in the pool, so the error log is reset whenever a scrub finishes. If the administrator determines that the errors are no longer present, and doesn't want to wait for a scrub to complete, all errors in the pool can be reset using the `zpool online` command.

If the data corruption is in pool-wide metadata, the output is slightly different:

```
# zpool status -v tank
  pool: tank
 state: FAULTED
reason: Data corruption detected.
action: Restore pool from backup
   see: http://www.sun.com/msg/ZFS-XXXX-09
config:
        NAME                  STATE     READ WRITE CKSUM
        test                  OFFLINE      0     0     0
          mirror              ONLINE       0     0     0
            c0t0d2            ONLINE       0     0     0
            c0t0d1            ONLINE       0     0     0
```

```
 scrub: none requested
errors: pool meata corrupted.  This pool cannot be accessed.
```

In the case of pool-wide corruption, the pool is placed into the `FAULTED` state, since it cannot possibly provide the needed replication level.

9.7.2 Repairing a Corrupted File or Directory

If a file or directory is corrupted, the system may still be able to function depending on the type of corruption. First, any damage is effectively unrecoverable — there are no good copies of the data anywhere on the system. If the data is valuable, there is no choice except to restore the affected data from backup. Even so, there may be ways to recover from this corruption without requiring the whole pool to need restoration.

If the damage is within a file data block, then the file can safely be removed, thereby clearing the error from the system. The first step is to try removing the file with rm(1). If this doesn't work, it means the corruption is within the file's metadata, and ZFS cannot determine which blocks belong to the file in order to remove it.

If the corruption is within a directory or a file's metadata, the only choice is to move the file out of the way. The administrator can safely move any file or directory to a less convenient location, allowing the original object to be restored in place. Once this is done, these 'damaged' files will forever appear in `zpool status` output, though they will be in a non-critical location where they should never be accessed. Future enhancements to ZFS will allow these damaged files to be flagged in such a way as to remove them from the namespace and not show up as permanent errors in the system.

9.7.3 Repairing Pool Wide Damage

If the damage is in pool metadata that prevents the pool from being openable, then you have no choice except to restore the pool and all its data from backup. The mechanism used to do this varies widely by pool configuration and backup strategy used. First, you should save the configuration as displayed by `zpool status` so that you can recreate it once the pool is destroyed. Then, use `zpool destroy -f` to destroy the pool. You should also keep a file describing the layout of datasets and the various locally set properties somewhere safe, as this information will become inaccessible if the pool is ever rendered inaccessible. Between the pool configuration and dataset layout, you can reconstruct your complete configuration after destroying the pool. The data can then be populated using whatever backup/restoration strategy you have employed.

9.8 Repairing an Unbootable System

ZFS is designed to be robust and stable in the face of errors. Even so, software bugs or certain unexpected pathologies may cause the system to panic when a pool is accessed. As part of the boot process, each pool must be opened, which means that such failures will cause a system to enter into a panic-reboot loop. In order to recover from this situation, ZFS must be informed not to look for any pools on startup.

ZFS keeps an internal cache of available pools and their configurations in `/etc/zfs/zpool.cache`. The location and contents of this file are private, and are subject to change at a future date. If the system becomes unbootable, boot to the `none` milestone using the `-m milestone=none` boot option. Once the system is up, remount your root filesystem writable and then remove `/etc/zfs/zpool.cache`. These actions cause ZFS to forget than any pools exist on the system, preventing it from trying to access the bad pool causing the problem. You can then proceed to normal system state by issuing the `svcadm milestone all` command. A similar process can be used when booting from an alternate root in order to perform repairs.

Once the system is up, you can attempt to import the pool using the `zpool import` command, although doing so will likely cause the same error as seen during boot, since it uses the same mechanism to access pools. If more than pool is on the system and you want to import a specific pool without accessing any others, you will have to re-initialize the devices in the damaged pool, at which point you can safely import the good pool.