Solaris ZFS Administration Guide

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

Part No: 819–5461–11
November 2006

061030@15490

Contents

Preface

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

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

Note – This Solaris release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems appear in the *Solaris 10 Hardware Compatibility List* at http://www.sun.com/bigadmin/hcl. This document cites any implementation differences between the platform types.

In this document these x86 terms mean the following:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" points out specific 64-bit information about AMD64 or EM64T systems.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

For supported systems, see the *Solaris 10 Hardware Compatibility List*.

Who Should Use This Book

This guide is intended for anyone who is interested in setting up and managing Solaris ZFS file systems. Experience using the Solaris Operating System (OS) or another UNIX® version is recommended.

How This Book Is Organized

The following table describes the chapters in this book.

Chapter	Description
Chapter 1	Provides an overview of ZFS and its features and benefits. It also covers some basic concepts and terminology.

Chapter	Description
Chapter 2	Provides step-by-step instructions on setting up simple ZFS configurations with simple pools and file systems. This chapter also provides the hardware and software required to create ZFS file systems.
Chapter 3	Identifies important features that make ZFS significantly different from traditional file systems. Understanding these key differences will help reduce confusion when using traditional tools to interact with ZFS.
Chapter 4	Provides a detailed description of how to create and administer storage pools.
Chapter 5	Provides detailed information about managing ZFS file systems. Included are such concepts as hierarchical file system layout, property inheritance, and automatic mount point management and share interactions.
Chapter 6	Describes how to create and administer ZFS snapshots and clones.
Chapter 7	Describes how to use access control lists (ACLs) to protect your ZFS files by providing more granular permissions then the standard UNIX permissions.
Chapter 8	Provides information on using 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*

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

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

TABLE P–2 Shell Prompts

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

◆ ◆ ◆ CHAPTER 1

Solaris ZFS File System (Introduction)

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

The following sections are provided in this chapter:

For information about ZFS training courses, see the following descriptions:

`http://www.sun.com/training/catalog/courses/SA-229-S10.xml`

`http://www.sun.com/training/catalog/courses/VC-SA-229-S10.xml`

What's New in ZFS?

This section summarizes new features in the ZFS file system that were added after the initial Solaris Express December 2005 release.

- "ZFS Web-Based Management" on page 17

Support for ZFS in the Sun Cluster 3.2 Release

ZFS is supported as a highly available local file system in the Sun Cluster 3.2 release. ZFS with Sun™ Cluster offers a best-class file-system solution combining high availability, data integrity, performance, and scalability, covering the needs of the most demanding environments.

For more information, see *Sun Cluster Data Services Planning and Administration Guide for Solaris OS*.

Recursive ZFS Snapshots

Solaris 10 11/06 Release: When you use the `zfs snapshot` command to create a file system snapshot, you can use the `-r` option to recursively create snapshots for all descendant file systems. In addition, using the `-r` option recursively destroys all descendant snapshots when a snapshot is destroyed.

Recursive ZFS snapshots are created extremely quickly and as one atomic operation, either all at once or none at all. The benefit of atomic snapshot operations is that the snapshot data is always from one consistent point in time, even across many descendant file systems.

For more information, see "Creating and Destroying ZFS Snapshots" on page 92.

Double Parity RAID-Z (`raidz2`)

Solaris 10 11/06 Release: A replicated RAID-Z configuration can now have either single- or double-parity, which means that one or two device failures can be sustained respectively, without any data loss. You can specify the `raidz2` keyword for a double-parity RAID-Z configuration. Or, you can specify the `raidz` or `raidz1` keyword for a single-parity RAID-Z configuration.

For more information, see "Creating a Double-Parity RAID-Z Storage Pool" on page 39 or `zpool`(1M).

Hot Spares for ZFS Storage Pool Devices

Solaris 10 11/06 Release: The ZFS hot spares feature enables you to identify disks that could be used to replace a failed or faulted device in one or more storage pools. Designating a device as a *hot spare* means that if an active device in the pool fails, the hot spare automatically replaces the failed device. Or, you can manually replace a device in a storage pool with a hot spare.

For more information, see "Designating Hot Spares in Your Storage Pool" on page 47 and `zpool`(1M).

Replacing a ZFS File System With a ZFS Clone (`zfs promote`)

Solaris 10 11/06 Release: The `zfs promote` command enables you to replace an existing ZFS file system with a clone of that file system. This feature is helpful when you want to run tests on an alternative version of a file system and then, make that alternative version of the file system the active file system.

For more information, see "Replacing a ZFS File System With a ZFS Clone" on page 96 and `zfs`(1M).

Upgrading ZFS Storage Pools (`zpool upgrade`)

Solaris 10 6/06 Release: You can upgrade your storage pools to a newer version to take advantage of the latest features by using the `zpool upgrade` command. In addition, the `zpool status` command has been modified to notify you when your pools are running older versions.

For more information, see "Upgrading ZFS Storage Pools" on page 63 and `zpool`(1M).

ZFS Backup and Restore Commands are Renamed

Solaris 10 6/06 Release: In this Solaris release, the `zfs backup` and `zfs restore` commands are renamed to `zfs send` and `zfs receive` to more accurately describe their function. The function of these commands is to save and restore ZFS data stream representations.

For more information about these commands, see "Saving and Restoring ZFS Data" on page 97.

Recovering Destroyed Storage Pools

Solaris 10 6/06 Release: This release includes the `zpool import -D` command, which enables you to recover pools that were previously destroyed with the `zpool destroy` command.

For more information, see "Recovering Destroyed ZFS Storage Pools" on page 61.

ZFS is Integrated With Fault Manager

Solaris 10 6/06 Release: This release includes the integration of a ZFS diagnostic engine that is capable of diagnosing and reporting pool failures and device failures. Checksum, I/O, device, and pool errors associated with pool or device failures are also reported.

The diagnostic engine does not include predictive analysis of checksum and I/O errors, nor does it include proactive actions based on fault analysis.

In the event of the ZFS failure, you might see a message similar to the following from `fmd`:

```
SUNW-MSG-ID: ZFS-8000-D3, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Fri Mar 10 11:09:06 MST 2006
PLATFORM: SUNW,Ultra-60, CSN: -, HOSTNAME: neo
SOURCE: zfs-diagnosis, REV: 1.0
EVENT-ID: b55ee13b-cd74-4dff-8aff-ad575c372ef8
DESC: A ZFS device failed.  Refer to http://sun.com/msg/ZFS-8000-D3 for more information.
AUTO-RESPONSE: No automated response will occur.
IMPACT: Fault tolerance of the pool may be compromised.
REC-ACTION: Run 'zpool status -x' and replace the bad device.
```

By reviewing the recommended action, which will be to follow the more specific directions in the `zpool status` command, you will be able to quickly identify and resolve the failure.

For an example of recovering from a reported ZFS problem, see "Repairing a Missing Device" on page 143.

New `zpool clear` Command

Solaris 10 6/06 Release: This release includes the `zpool clear` command for clearing error counts associated with a device or the pool. Previously, error counts were cleared when a device in a pool was brought online with the `zpool online` command. For more information, see `zpool`(1M) and "Clearing Storage Pool Devices" on page 46.

Compact NFSv4 ACL Format

Solaris 10 6/06 Release: In this release, three NFSv4 ACL formats are available: verbose, positional, and compact. The new compact and positional ACL formats are available to set and display ACLs. You can use the `chmod` command to set all 3 ACL formats. You can use the `ls -V` command to display compact and positional ACL formats and the `ls -v` command to display verbose ACL formats.

For more information, see "Setting and Displaying ACLs on ZFS Files in Compact Format" on page 122, `chmod`(1), and `ls`(1).

File System Monitoring Tool (`fsstat`)

Solaris 10 6/06 Release: A new file system monitoring tool, `fsstat`, is available to report file system operations. Activity can be reported by mount point or by file system type. The following example shows general ZFS file system activity.

```
$ fsstat zfs
 new  name   name  attr  attr lookup rddir  read read  write write
 file remov  chng   get   set    ops   ops   ops bytes   ops bytes
7.82M 5.92M 2.76M 1.02G 3.32M  5.60G 87.0M  363M 1.86T 20.9M  251G zfs
```

For more information, see `fsstat`(1M).

ZFS Web-Based Management

Solaris 10 6/06 Release: A web-based ZFS management tool is available to perform many administrative actions. With this tool, you can perform the following tasks:

- Create a new storage pool.
- Add capacity to an existing pool.
- Move (export) a storage pool to another system.
- Import a previously exported storage pool to make it available on another system.
- View information about storage pools.
- Create a file system.
- Create a volume.
- Take a snapshot of a file system or a volume.
- Roll back a file system to a previous snapshot.

You can access the ZFS Administration console through a secure web browser at the following URL:

```
https://system-name:6789/zfs
```

If you type the appropriate URL and are unable to reach the ZFS Administration console, the server might not be started. To start the server, run the following command:

```
# /usr/sbin/smcwebserver start
```

If you want the server to run automatically when the system boots, run the following command:

```
# /usr/sbin/smcwebserver enable
```

Note – You cannot use the Solaris Management Console (`smc`) to manage ZFS storage pools or file systems.

What Is ZFS?

The Solaris ZFS file system is a revolutionary new file system that fundamentally changes the way file systems are administered, with features and benefits not found in any other file system available today. ZFS has been designed to be robust, scalable, and simple to administer.

ZFS Pooled Storage

ZFS uses the concept of *storage pools* to manage physical storage. Historically, file systems were constructed on top of a single physical device. 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 file systems would not have to be modified to take advantage of multiple devices. This design added another layer of complexity and ultimately prevented certain file system advances, because the file system had no control over the physical placement of data on the virtualized volumes.

ZFS eliminates the volume management altogether. Instead of forcing you 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, and so on,) and acts as an arbitrary data store from which file systems can be created. File systems are no longer constrained to individual devices, allowing them to share space with all file systems in the pool. You no longer need to predetermine the size of a file system, as file systems grow automatically within the space allocated to the storage pool. When new storage is added, all file systems within the pool can immediately use 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 you to invoke some commands to configure the memory and assign it to individual processes. All processes on the system automatically use the additional memory.

Transactional Semantics

ZFS is a transactional file system, which means that the file system state is always consistent on disk. Traditional file systems overwrite data in place, which means that if the machine loses power, for example, between the time a data block is allocated and when it is linked into a directory, the file system will be left in an inconsistent state. Historically, this problem was solved through the use of the `fsck` command. This command was responsible for going through and verifying file system state, making an attempt to repair any inconsistencies in the process. This problem caused great pain to administrators and was never guaranteed to fix all possible problems. More recently, file systems have introduced the concept of *journaling*. The journaling process records action in a separate journal, which can then be replayed safely if a system crash occurs. This process introduces unnecessary overhead, because 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 file system, 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 mechanism means that the file system can never be corrupted through accidental loss of power or a system crash. So, no need for a `fsck` equivalent exists. While the most recently written pieces of data might be lost, the file system 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.

Checksums and Self-Healing Data

With ZFS, all data and metadata is checksummed using a user-selectable algorithm. Traditional file systems that do provide checksumming have performed it on a per-block basis, out of necessity due to the volume management layer and traditional file system design. The traditional design 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 file system layer, and is transparent to applications.

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, ZFS fetches the correct data from another replicated copy, and repairs the bad data, replacing it with the good copy.

Unparalleled Scalability

ZFS has been designed from the ground up to be the most scalable file system, ever. The file system itself is 128-bit, allowing for 256 quadrillion zettabytes of storage. All metadata is allocated dynamically, so no need exists to pre-allocate inodes or otherwise limit the scalability of the file system 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 no limit exists on the number of file systems or number of files that can be contained within a file system.

ZFS Snapshots

A *snapshot* is a read-only copy of a file system 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. As a result, the snapshot prevents the data from being freed back to the pool.

Simplified Administration

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

ZFS manages file systems through a hierarchy that allows for this simplified management of properties such as quotas, reservations, compression, and mount points. In this model, file systems become the central point of control. File systems themselves are very cheap (equivalent to a new directory), so you are encouraged to create a file system for each user, project, workspace, and so on. This design allows you to define fine-grained management points.

ZFS Terminology

This section describes the basic terminology used throughout this book:

checksum
: A 256-bit hash of the data in a file system block. The checksum capability can range from the simple and fast fletcher2 (the default) to cryptographically strong hashes such as SHA256.

clone
: A file system whose initial contents are identical to the contents of a snapshot.

  For information about clones, see "ZFS Clones" on page 95.

dataset
: A generic name for the following ZFS entities: clones, file systems, snapshots, or volumes.

  Each dataset is identified by a unique name in 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 path name for the dataset object

  snapshot
  : Is an optional component that identifies a snapshot of a dataset

  For more information about datasets, see Chapter 5.

file system
: A dataset that contains a standard POSIX file system.

  For more information about file systems, 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 other disk in that mirror can provide the same data.

pool
: A logical group of devices describing the layout and physical characteristics of the 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. For more information about RAID-Z, see "RAID-Z Storage Pool Configuration" on page 36.

resilvering	The process of transferring data from one device to another device is known as *resilvering*. For example, if a mirror component is replaced or taken offline, the data from the up-to-date mirror component is copied to the newly restored mirror component. This process is referred to as *mirror resynchronization* in traditional volume management products. For more information about ZFS resilvering, see "Viewing Resilvering Status" on page 148.
snapshot	A read-only image of a file system or volume at a given point in time. For more information about snapshots, see "ZFS Snapshots" on page 91.
virtual device	A logical device in a pool, which can be a physical device, a file, or a collection of devices. For more information about virtual devices, see "Virtual Devices in a Storage Pool" on page 35.
volume	A dataset used to emulate a physical device. For example, you can create an emulated volume as a swap device. For more information about emulated volumes, see "Emulated Volumes" on page 127.

ZFS Component Naming Requirements

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

- Empty components are not allowed.
- Each component can only contain alphanumeric characters in addition to the following four special characters:
  - Underscore (_)
  - Hyphen (-)
  - Colon (:)
  - Period (.)
- Pool names must begin with a letter, except that the beginning sequence c[0-9] is not allowed. In addition, pool names that begin with `mirror`, `raidz`, or `spare` are not allowed as these name are reserved.
- Dataset names must begin with an alphanumeric character.

◆ ◆ ◆ CHAPTER 2

Getting Started With ZFS

This chapter provides step-by-step instructions on 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 file systems. This chapter 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:

ZFS Hardware and Software Requirements and Recommendations

Make sure you review the following hardware and software requirements and recommendations before attempting to use the ZFS software:

- A SPARC® or x86 system that is running the Solaris 10 6/06 release or later release.
- The minimum disk size is 128 Mbytes. The minimum amount of disk space required for a storage pool is approximately 64 Mbytes.
- Currently, the minimum amount of memory recommended to install a Solaris system is 512 Mbytes. However, for good ZFS performance, at least one Gbyte or more of memory is recommended.
- If you create a mirrored disk configuration, multiple controllers are recommended.

Creating a Basic ZFS File System

ZFS administration has been designed with simplicity in mind. Among the goals of the ZFS design is to reduce the number of commands needed to create a usable file system. When you create a new pool, a new ZFS file system is created and mounted automatically.

The following example illustrates how to create a storage pool named `tank` and a ZFS file system name `tank` in one command. Assume that the whole disk `/dev/dsk/c1t0d0` is available for use.

```
# zpool create tank c1t0d0
```

The new ZFS file system, `tank`, can use as much of the disk space on `c1t0d0` as needed, and is automatically mounted at `/tank`.

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

Within a pool, you will probably want to create additional file systems. File systems provide points of administration that allow you to manage different sets of data within the same pool.

The following example illustrates how to create a file system named `fs` in the storage pool `tank`. Assume that the whole disk `/dev/dsk/c1t0d0` is available for use.

```
# zpool create tank c1t0d0
# zfs create tank/fs
```

The new ZFS file system, `tank/fs`, can use as much of the disk space on `c1t0d0` 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
```

In most cases, you will probably want to create and organize a hierarchy of file systems that matches your organizational needs. For more information about creating a hierarchy of ZFS file systems, see "Creating a ZFS File System Hierarchy" on page 26.

Creating a ZFS Storage Pool

The previous example illustrates the simplicity of ZFS. The remainder of this chapter demonstrates a more complete example similar to what you would encounter in your environment. The first tasks are to identify your storage requirements and create a storage pool. The pool describes the physical characteristics of the storage and must be created before any file systems are created.

▼ Identifying Storage Requirements

1 Determine available devices.

Before creating a storage pool, you must determine which devices will store your data. These devices must be disks of at least 128 Mbytes in size, and they must not be in use by other parts of the operating system. The devices can be individual slices on a preformatted disk, or they can be entire disks that ZFS formats as a single large slice.

For the storage example used in "Creating the ZFS Storage Pool" on page 25, assume that the whole disks `/dev/dsk/c1t0d0` and `/dev/dsk/c1t1d0` are available for use.

For more information about disks and how they are used and labeled, see "Using Disks in a ZFS Storage Pool" on page 33.

2 Choose data replication.

ZFS supports multiple types of data replication, which determines what types of hardware failures the pool can withstand. ZFS supports nonredundant (striped) configurations, as well as mirroring and RAID-Z (a variation on RAID-5).

For the storage example used in "Creating the ZFS Storage Pool" on page 25, basic mirroring of two available disks is used.

For more information about ZFS replication features, see "Replication Features of a ZFS Storage Pool" on page 36.

▼ Creating the ZFS Storage Pool

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

For more information about the ZFS rights profiles, see "ZFS Rights Profiles" on page 133.

2 Pick a pool name.

The pool name is used to identify the storage pool when you are using the `zpool` or `zfs` commands. Most systems require only a single pool, so you can pick any name that you prefer, provided it satisfies the naming requirements outlined in "ZFS Component Naming Requirements" on page 21.

3 **Create the pool.**

For example, create a mirrored pool that is named `tank`.

```
# zpool create tank mirror c1t0d0 c1t1d0
```

If one or more devices contains another file system or is otherwise in use, the command cannot create the pool.

Caution – Do not add a disk that is currently configured as a Sun Cluster quorum device to a ZFS storage pool. After the disk is added to a storage pool, then the disk can be configured as a quorum device.

For more information about creating storage pools, see "Creating a ZFS Storage Pool" on page 38.

For more information about how device usage is determined, see "Detecting in Use Devices" on page 39.

4 **View the results.**

You can determine if 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 about viewing pool status, see "Querying ZFS Storage Pool Status" on page 50.

Creating a ZFS File System Hierarchy

After creating a storage pool to store your data, you can create your file system hierarchy. Hierarchies are simple yet powerful mechanisms for organizing information. They are also very familiar to anyone who has used a file system.

ZFS allows file systems to be organized into arbitrary hierarchies, where each file system has only a single parent. The root of the 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 file systems.

▼ Determining the ZFS File System Hierarchy

1 **Pick the file system granularity.**

ZFS file systems are the central point of administration. They are lightweight and can be created easily. A good model to use is a file system per user or project, as this model allows properties, snapshots, and backups to be controlled on a per-user or per-project basis.

Two ZFS file systems, `bonwick` and `billm`, are created in "Creating ZFS File Systems" on page 27.

For more information on managing file systems, see Chapter 5.

2 Group similar file systems.

ZFS allows file systems to be organized into hierarchies so that similar file systems can be grouped. This model provides a central point of administration for controlling properties and administering file systems. Similar file systems should be created under a common name.

For the example in "Creating ZFS File Systems" on page 27, the two file systems are placed under a file system named `home`.

3 Choose the file system properties.

Most file system characteristics are controlled by using simple properties. These properties control a variety of behavior, including where the file systems are mounted, how they are shared, if they use compression, and if any quotas are in effect.

For the example in "Creating ZFS File Systems" on page 27, all home directories are mounted at `/export/zfs/`*user*, are shared by using NFS, and with compression enabled. In addition, a quota of 10 Gbytes on `bonwick` is enforced.

For more information about properties, see "ZFS Properties" on page 68.

▼ Creating ZFS File Systems

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

For more information about the ZFS rights profiles, see "ZFS Rights Profiles" on page 133.

2 Create the desired hierarchy.

In this example, a file system that acts as a container for individual file systems is created.

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

Next, individual file systems are grouped under the `home` file system in the pool `tank`.

3 Set the inherited properties.

After the file system hierarchy is established, 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
# zfs get compression tank/home
NAME             PROPERTY       VALUE                      SOURCE
tank/home        compression    on                         local
```

For more information about properties and property inheritance, see "ZFS Properties" on page 68.

4 Create the individual file systems.

Note that the file systems could have been created and then the properties could have been changed at the `home` level. All properties can be changed dynamically while file systems are in use.

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

These file systems inherit their property settings from their parent, so they are automatically mounted at `/export/zfs/`*user* and are NFS shared. You do not need to edit the `/etc/vfstab` or `/etc/dfs/dfstab` file.

For more information about creating file systems, see "Creating a ZFS File System" on page 66.

For more information about mounting and sharing file systems, see "Mounting and Sharing ZFS File Systems" on page 81.

5 Set the file system-specific properties.

In this example, user `bonwick` is assigned a quota of 10 Gbytes. This property 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.

View available file system information by using 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 the user `bonwick` only has 10 Gbytes of space available, while the user `billm` can use the full pool (67 Gbytes).

For more information about viewing file system status, see "Querying ZFS File System Information" on page 75.

For more information about how space is used and calculated, see "ZFS Space Accounting" on page 30.

◆ ◆ ◆ CHAPTER 3

ZFS and Traditional File System Differences

This chapter discusses some significant differences between ZFS and traditional file systems. Understanding these key differences can help reduce confusion when using traditional tools to interact with ZFS.

The following sections are provided in this chapter:

ZFS File System Granularity

Historically, file systems have been constrained to one device so that the file systems themselves have been constrained to the size of the device. Creating and re-creating traditional file systems because of size constraints are time-consuming and sometimes difficult. Traditional volume management products helped manage this process.

Because ZFS file systems are not constrained to specific devices, they can be created easily and quickly, similar to the way directories are created. ZFS file systems grow automatically within the space allocated to the storage pool.

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

For an example of creating a file system hierarchy, see "Creating a ZFS File System Hierarchy" on page 26.

ZFS 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 such as `df` might change even when the file system is inactive, as other file systems in the pool consume or release space. Note that the maximum file system size can be limited by using quotas. For information about quotas, see "Setting Quotas on ZFS File Systems" on page 87. Space can be guaranteed to a file system by using reservations. For information about reservations, see "Setting Reservations on ZFS File Systems" on page 88. This model is very similar to the NFS model, where multiple directories are mounted from the same file system (consider `/home`).

All metadata in ZFS is allocated dynamically. Most other file systems pre-allocate much of their metadata. As a result, an immediate space cost at file system creation for this metadata is required. This behavior also means that the total number of files supported by the file systems is predetermined. Because ZFS allocates its metadata as it needs it, no initial space cost is required, 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 that is 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 these modifications are committed to disk, they are termed *pending changes*. The amount of space used, available, and referenced by a file or file system does not consider pending changes. Pending changes are generally accounted for within a few seconds. Even committing a change to disk by using `fsync(3c)` or `O_SYNC` does not necessarily guarantee that the space usage information is updated immediately.

Out of Space Behavior

File system snapshots are inexpensive and easy to create in ZFS. Most likely, snapshots will be common in most ZFS environments. For information about ZFS snapshots, see Chapter 6.

The presence of snapshots can cause some unexpected behavior when you attempt to free space. Typically, given appropriate permissions, you can remove a file from a full file system, and 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 no space is gained from the file deletion. The blocks used by the file continue to be referenced from the snapshot.

As a result, the file deletion can consume more disk space, because a new version of the directory needs to be created to reflect the new state of the namespace. This behavior means that you can get an unexpected `ENOSPC` or `EDQUOT` when attempting to remove a file.

Mounting ZFS File Systems

ZFS is designed to reduce complexity and ease administration. For example, with existing file systems you must edit the `/etc/vfstab` file every time you add a new file system. ZFS has eliminated this requirement by automatically mounting and unmounting file systems according to the properties of the dataset. You do not need to manage ZFS entries in the `/etc/vfstab` file.

For more information about mounting and sharing ZFS file systems, see "Mounting and Sharing ZFS File Systems" on page 81.

Traditional Volume Management

As described in "ZFS Pooled Storage" on page 18, ZFS eliminates the need for a separate volume manager. ZFS operates on raw devices, so it is possible to create a storage pool comprised of logical volumes, either software or hardware. This configuration is not recommended, as ZFS works best when it uses raw physical devices. Using logical volumes might sacrifice performance, reliability, or both, and should be avoided.

New Solaris ACL Model

Previous versions of the Solaris OS 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 Solaris 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 ZFS Storage Pools

This chapter describes how to create and administer ZFS storage pools.

The following sections are provided in this chapter:

Components of a ZFS Storage Pool

This section provides detailed information about the following storage pool components:

- Disks
- Files
- Virtual devices

Using Disks in a ZFS Storage Pool

The most basic element of a storage pool is a piece of physical storage. Physical storage can be any block device of at least 128 Mbytes in size. Typically, this device is a hard drive that is visible to the system in the `/dev/dsk` directory.

A storage device can be a whole disk (`c1t0d0`) 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 way, the partition table that is displayed by the `format` command appears 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
```

To use whole disks, the disks must be named using the standard Solaris convention, such as `/dev/dsk/cXtXdXsX`. Some third-party drivers use a different naming convention or place disks in a location other than the `/dev/dsk` directory. To use these disks, you must manually label the disk and provide a slice to ZFS.

ZFS applies an EFI label when you create a storage pool with whole disks. Disks can be labeled with a traditional Solaris VTOC label when you create a storage pool with a disk slice.

Slices should only be used under the following conditions:

- The device name is nonstandard.
- A single disk is shared between ZFS and another file system, such as UFS.
- A disk is used as a swap or a dump device.

Disks can be specified by using either the full path, such as `/dev/dsk/c1t0d0`, or a shorthand name that consists of the device name within the `/dev/dsk` directory, such as `c1t0d0`. For example, the following are valid disk names:

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

Using whole physical disks is the simplest way to create ZFS storage pools. ZFS configurations become progressively more complex, from management, reliability, and performance perspectives, when you build pools from disk slices, LUNs in hardware RAID arrays, or volumes presented by software-based volume managers. The following considerations might help you determine how to configure ZFS with other hardware or software storage solutions:

- If you construct ZFS configurations on top of LUNs from hardware RAID arrays, you need to understand the relationship between ZFS redundancy features and the redundancy features offered by the array. Certain configurations might provide adequate redundancy and performance, but other configurations might not.

- You can construct logical devices for ZFS using volumes presented by software-based volume managers, such as Solaris™ Volume Manager (SVM) or Veritas Volume Manager (VxVM). However, these configurations are not recommended. While ZFS functions properly on such devices, less-than-optimal performance might be the result.

Disks are identified both by their path and by their device ID, if available. This method 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 situation happens, ZFS can still access the device by path and update the stored device ID automatically. If you inadvertently change both the path and the ID of the device, then export and re-import the pool in order to use it.

Using Files in a ZFS Storage Pool

ZFS also allows you to use UFS files as virtual devices in your storage pool. This feature is aimed primarily at testing and enabling simple experimentation, not for production use. The reason is that **any use of files relies on the underlying file system for consistency**. If you create a ZFS pool backed by files on a UFS file system, then you are implicitly relying on UFS to guarantee correctness and synchronous semantics.

However, files can be quite useful when you are 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 Mbytes in size. If a file is moved or renamed, the pool must be exported and re-imported in order to use it, as no device ID is associated with files by which they can be located.

Virtual Devices in a Storage Pool

Each storage pool is comprised of one or more virtual devices. A *virtual device* is an internal representation of the storage pool that describes the layout of physical storage and its fault characteristics. As such, a virtual device represents the disk devices or files that are used to create the storage pool.

Two top-level virtual devices provide data redundancy: mirror and RAID-Z virtual devices. These virtual devices consist of disks, disk slices, or files.

Disks, disk slices, or files that are used in pools outside of mirrors and RAID-Z virtual devices, function as top-level virtual devices themselves.

Storage pools typically contain multiple top-level virtual devices. ZFS dynamically stripes data among all of the top-level virtual devices in a pool.

Replication Features of a ZFS Storage Pool

ZFS provides two levels of data redundancy in a mirrored and a RAID-Z configuration.

Mirrored Storage Pool Configuration

A mirrored storage pool configuration requires at least two disks, preferably on separate controllers. Many disks can be used in a mirrored configuration. In addition, you can create more than one mirror in each pool. Conceptually, a simple mirrored configuration would look similar to the following:

```
mirror c1t0d0 c2t0d0
```

Conceptually, a more complex mirrored configuration would look similar to the following:

```
mirror c1t0d0 c2t0d0 c3t0d0 mirror c4t0d0 c5t0d0 c6t0d0
```

For information about creating a mirrored storage pool, see "Creating a Mirrored Storage Pool" on page 38.

RAID-Z Storage Pool Configuration

In addition to a mirrored storage pool configuration, ZFS provides a RAID-Z configuration. RAID-Z is 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 the 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 design is only possible because ZFS integrates file system and device management in such a way that the file system'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.

A RAID-Z configuration with N disks of size X with P parity disks can hold approximately (N-P)*X bytes and can withstand one device failing before data integrity is compromised. You need at least two disks for a single-parity RAID-Z configuration and at least three disks for a double-parity RAID-Z configuration. For example, if you have three disks in a single-parity RAID-Z configuration, parity data occupies space equal to one of the three disks. Otherwise, no special hardware is required to create a RAID-Z configuration.

Conceptually, a RAID-Z configuration with three disks would look similar to the following:

```
raidz c1t0d0 c2t0d0 c3t0d0
```

A more complex conceptual RAID-Z configuration would look similar to the following:

```
raidz c1t0d0 c2t0d0 c3t0d0 c4t0d0 c5t0d0 c6t0d0 c7t0d0 raidz c8t0d0 c9t0d0 c10t0d0 c11t0d0
c12t0d0 c13t0d0 c14t0d0
```

If you are creating a RAID-Z configuration with many disks, as in this example, a RAID-Z configuration with 14 disks is better split into a two 7-disk groupings. RAID-Z configurations with single-digit groupings of disks should perform better.

For information about creating a RAID-Z storage pool, see "Creating a Single-Parity RAID-Z Storage Pool" on page 38 or "Creating a Double-Parity RAID-Z Storage Pool" on page 39

For more information about choosing between a mirrored configuration or a RAID-Z configuration based on performance and space considerations, see the following blog:

`http://blogs.sun.com/roller/page/roch?entry=when_to_and_not_to`

Self-Healing Data in a Replicated Configuration

ZFS provides for self-healing data in a mirrored or RAID-Z configuration.

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

Dynamic Striping in a Storage Pool

For each virtual device that is added to the pool, ZFS dynamically stripes data across all available devices. The decision about where to place data is done at write time, so no fixed width stripes are created at allocation time.

When virtual devices are added to a pool, ZFS gradually allocates data to the new device in order to maintain performance and space allocation policies. Each virtual device can also be a mirror or a RAID-Z device that contains other disk devices or files. This configuration allows for flexibility in controlling the fault characteristics of your pool. For example, you could create the following configurations out of 4 disks:

- 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, this practice is not recommended. For example, you can create a pool with a two-way mirror and a three-way RAID-Z configuration. However, 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 of the same type with the same replication level in each device.

Creating and Destroying ZFS Storage Pools

By design, creating and destroying pools is fast and easy. However, be cautious when doing these operations. Although checks are performed to prevent using devices known to be in use in a new pool, ZFS cannot always know when a device is already in use. Destroying a pool is even easier. Use `zpool destroy` with caution. This is a simple command with significant consequences. For information about destroy pools, see "Destroying ZFS Storage Pools" on page 42.

Creating a ZFS Storage Pool

To create a storage pool, use the `zpool create` command. This command takes a pool name and any number of virtual devices as arguments. The pool name must satisfy the naming conventions outlined in "ZFS Component Naming Requirements" on page 21.

Caution – Do not add a disk that is currently configured as a Sun Cluster quorum device to a ZFS storage pool. After the disk is added to a storage pool, then the disk can be configured as a quorum device.

Creating a Basic Storage Pool

The following command creates a new pool named `tank` that consists of the disks `c1t0d0` and `c1t1d0`:

```
# zpool create tank c1t0d0 c1t1d0
```

These whole disks are found in the `/dev/dsk` directory and are labelled appropriately by ZFS to contain a single, large slice. Data is dynamically striped across both disks.

Creating a Mirrored Storage Pool

To create a mirrored pool, use the `mirror` keyword, followed by any number of storage devices that will comprise 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 c1d0 c2d0 mirror c3d0 c4d0
```

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.

Creating a Single-Parity RAID-Z Storage Pool

Creating a single-parity RAID-Z pool is identical to creating a mirrored pool, except that the `raidz` keyword is used instead of `mirror`. The following example shows how to create a pool with a single RAID-Z device that consists of five disks:

```
# zpool create tank raidz c1t0d0 c2t0d0 c3t0d0 c4t0d0 /dev/dsk/c5t0d0
```

This example demonstrates that disks can be specified by using their full paths. The `/dev/dsk/c5t0d0` device is identical to the `c5t0d0` device.

A similar configuration could be created with disk slices. For example:

```
# zpool create tank raidz c1t0d0s0 c2t0d0s0 c3t0d0s0 c4t0d0s0 c5t0d0s0
```

However, the disks must be preformatted to have an appropriately sized slice zero.

For more information about a RAID-Z configuration, see "RAID-Z Storage Pool Configuration" on page 36.

Creating a Double-Parity RAID-Z Storage Pool

You can create a double-parity RAID-Z configuration by using the `raidz2` keyword when the pool is created. For example:

```
# zpool create -f tank raidz2 c1t0d0 c2t0d0 c3t0d0
# zpool status -v tank
  pool: tank
 state: ONLINE
 scrub: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        tank          ONLINE       0     0     0
          raidz2      ONLINE       0     0     0
            c1t0d0    ONLINE       0     0     0
            c2t0d0    ONLINE       0     0     0
            c3t0d0    ONLINE       0     0     0

errors: No known data errors
```

Handling ZFS Storage Pool Creation Errors

Pool creation errors can occur for many reasons. Some of these reasons are obvious, such as when a specified device doesn't exist, while other reasons are more subtle.

Detecting in Use Devices

Before formatting a device, ZFS first determines if the disk is in use by ZFS or some other part of the operating system. If the disk is in use, you might see errors such as the following:

```
# zpool create tank c1t0d0 c1t1d0
invalid vdev specification
use '-f' to override the following errors:
```

```
/dev/dsk/c1t0d0s0 is currently mounted on /. Please see umount(1M).
/dev/dsk/c1t0d0s1 is currently mounted on swap. Please see swap(1M).
/dev/dsk/c1t1d0s0 is part of active ZFS pool zeepool. Please see zpool(1M).
```

Some of these errors can be overridden by using the `-f` option, but most errors cannot. The following uses cannot be overridden by using the `-f` option, and you must manually correct them:

Mounted file system — The disk or one of its slices contains a file system that is currently mounted. To correct this error, use the `umount` command.

File system in /etc/vfstab — The disk contains a file system that is listed in the `/etc/vfstab` file, but the file system 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` command.

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

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

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

Part of volume — The disk is part of an SVM volume.

Live upgrade — The disk is in use as an alternate boot environment for Solaris 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 might or might not be a network-attached drive in use by another system. Be cautious when overriding a potentially active pool.

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

```
# zpool create tank c1t0d0
invalid vdev specification
use '-f' to override the following errors:
/dev/dsk/c1t0d0s0 contains a ufs filesystem.
# zpool create -f tank c1t0d0
```

Ideally, correct the errors rather than use the `-f` option.

Mismatched Replication Levels

Creating pools with virtual devices of different replication levels is not recommended. The `zpool` 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 see errors similar to the following:

```
# zpool create tank c1t0d0 mirror c2t0d0 c3t0d0
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: both disk and mirror vdevs are present
# zpool create tank mirror c1t0d0 c2t0d0 mirror c3t0d0 c4t0d0 c5t0d0
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: 2-way mirror and 3-way mirror vdevs are present
```

You can override these errors with the `-f` option, though this practice is not recommended. The command also warns you about creating a mirrored or RAID-Z pool using devices of different sizes. While this configuration is allowed, mismatched replication levels result in unused space on the larger device, and requires the `-f` option to override the warning.

Doing a Dry Run of Storage Pool Creation

Because creating a pool can fail unexpectedly in different ways, and because formatting disks is such a potentially harmful action, the `zpool create` command has an additional option, `-n`, which simulates creating the pool without actually writing data to disk. This option performs 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 c1t1d0
would create 'tank' with the following layout:

        tank
          mirror
            c1t0d0
            c1t1d0
```

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

Default Mount Point for Storage Pools

When a pool is created, the default mount point for the root dataset is */pool-name*. This directory must either not exist or be empty. If the directory does not exist, it is automatically created. If the directory is empty, the root dataset is mounted on top of the existing directory. To create a pool with a different default mount point, use the `-m` option of the `zpool create` command:

```
# zpool create home c1t0d0
default mountpoint '/home' exists and is not empty
use '-m' option to specify a different default
# zpool create -m /export/zfs home c1t0d0

# zpool create home c1t0d0
default mountpoint '/home' exists and is not empty
use '-m' option to provide a different default
# zpool create -m /export/zfs home c1t0d0
```

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

For more information about mount points, see "Managing ZFS Mount Points" on page 81.

Destroying ZFS Storage 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 – Be very careful when you destroy a pool. Make sure you are destroying the right pool and you always have copies of your data. If you accidentally destroy the wrong pool, you can attempt to recover the pool. For more information, see "Recovering Destroyed ZFS Storage Pools" on page 61.

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 state information prevents the devices from showing up as a potential pool when you perform an import. If one or more devices are unavailable, the pool can still be destroyed. However, the necessary state information won't be written to these damaged devices.

These devices, when suitably repaired, are reported as *potentially active* when you create a new pool, and appear as valid devices when you search for pools to import. If a pool has enough faulted devices such that the pool itself is faulted (meaning that a top-level virtual device is faulted), then the command prints a warning and cannot complete without the `-f` option. This option is necessary because the pool cannot be opened, so whether data is stored there or not is unknown. For example:

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

For more information about pool and device health, see "Health Status of ZFS Storage Pools" on page 54.

For more information about importing pools, see "Importing ZFS Storage Pools" on page 60.

Managing Devices in ZFS Storage Pools

Most of the basic information regarding devices is covered in "Components of a ZFS Storage Pool" on page 33. Once a pool has been created, you can perform several tasks to manage the physical devices within the pool.

Adding Devices to a Storage Pool

You can dynamically add space 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. For example:

```
# zpool add zeepool mirror c2t1d0 c2t2d0
```

The format of the virtual devices is the same as for the `zpool create` command, and the same rules apply. Devices are checked to determine if they are in use, and the command cannot change the replication level without the `-f` option. The command also supports the `-n` option so that you can perform a dry run. For example:

```
# zpool add -n zeepool mirror c3t1d0 c3t2d0
would update 'zeepool' to the following configuration:
      zeepool
        mirror
            c1t0d0
            c1t1d0
        mirror
            c2t1d0
            c2t2d0
        mirror
            c3t1d0
            c3t2d0
```

This command syntax would add mirrored devices `c3t1d0` and `c3t2d0` to `zeepool`'s existing configuration.

For more information about how virtual device validation is done, see "Detecting in Use Devices" on page 39.

Attaching and Detaching Devices in a Storage Pool

In addition to the `zpool add` command, you can use the `zpool attach` command to add a new device to an existing mirrored or non-mirrored device. For example:

```
# zpool attach zeepool c1t1d0 c2t1d0
```

If the existing device is part of a two-way mirror, attaching the new device, creates a three-way mirror, and so on. In either case, the new device begins to resilver immediately.

In this example, `zeepool` is an existing two-way mirror that is transformed to a three-way mirror by attaching `c2t1d0`, the new device, to the existing device, `c1t1d0`.

You can use the `zpool detach` command to detach a device from a pool. For example:

```
# zpool detach zeepool c2t1d0
```

However, this operation is refused if there are no other valid replicas of the data. For example:

```
# zpool detach newpool c1t2d0
cannot detach c1t2d0: only applicable to mirror and replacing vdevs
```

Onlining and Offlining Devices in a Storage Pool

ZFS allows individual devices to be taken offline or brought online. When hardware is unreliable or not functioning properly, ZFS continues to read or write data to the device, assuming the condition is only temporary. If the condition is not temporary, it is possible to instruct ZFS to ignore the device by bringing it offline. ZFS does not send any requests to an offlined device.

Note – Devices do not need to be taken offline in order to replace them.

You can use the `offline` command when you need to temporarily disconnect storage. For example, if you need to physically disconnect an array from one set of Fibre Channel switches and connect the array to a different set, you could take the LUNs offline from the array that was used in ZFS storage pools. After the array was reconnected and operational on the new set of switches, you could then bring the same LUNs online. Data that had been added to the storage pools while the LUNs were offline would resilver to the LUNs after they were brought back online.

This scenario is possible assuming that the systems in question see the storage once it is attached to the new switches, possibly through different controllers than before, and your pools are set up as RAID-Z or mirrored configurations.

Taking a Device Offline

You can take a device offline by using the `zpool offline` command. The device can be specified by path or by short name, if the device is a disk. For example:

```
# zpool offline tank c1t0d0
bringing device c1t0d0 offline
```

You cannot take a pool offline to the point where it becomes faulted. For example, you cannot take offline two devices out of a RAID-Z configuration, nor can you take offline a top-level virtual device.

```
# zpool offline tank c1t0d0
cannot offline c1t0d0: no valid replicas
```

Offlined devices show up in the `OFFLINE` state when you query pool status. For information about querying pool status, see "Querying ZFS Storage Pool Status" on page 50.

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

To temporarily take a device offline, use the `zpool offline -t` option. For example:

```
# zpool offline -t tank c1t0d0
 bringing device 'c1t0d0' offline
```

When the system is rebooted, this device is automatically returned to the `ONLINE` state.

For more information on device health, see "Health Status of ZFS Storage Pools" on page 54.

Bringing a Device Online

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

```
# zpool online tank c1t0d0
bringing device c1t0d0 online
```

When a device is brought online, any data that has been written to the pool is resynchronized 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.

If you attempt to online a faulted device, a message similar to the following is displayed from `fmd`:

```
# zpool online tank c1t0d0
Bringing device c1t0d0 online
#
SUNW-MSG-ID: ZFS-8000-D3, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Thu Aug 31 11:13:59 MDT 2006
PLATFORM: SUNW,Ultra-60, CSN: -, HOSTNAME: neo
SOURCE: zfs-diagnosis, REV: 1.0
```

```
EVENT-ID: e11d8245-d76a-e152-80c6-e63763ed7e4f
DESC: A ZFS device failed.  Refer to http://sun.com/msg/ZFS-8000-D3 for more information.
AUTO-RESPONSE: No automated response will occur.
IMPACT: Fault tolerance of the pool may be compromised.
REC-ACTION: Run 'zpool status -x' and replace the bad device.
```

For more information on replacing a faulted device, see "Repairing a Missing Device" on page 143.

Clearing Storage Pool Devices

If a device is taken offline due to a failure that causes errors to be listed in the `zpool status` output, you can clear the error counts with the `zpool clear` command.

If specified with no arguments, this command clears all device errors within the pool. For example:

```
# zpool clear tank
```

If one or more devices are specified, this command only clear errors associated with the specified devices. For example:

```
# zpool clear tank c1t0d0
```

For more information on clearing `zpool` errors, see "Clearing Transient Errors" on page 146.

Replacing Devices in a Storage Pool

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

```
# zpool replace tank c1t1d0 c1t2d0
```

In this example, the previous device, `c1t1d0`, is replaced by `c1t2d0`.

The replacement device must be greater than or equal to the minimum size of all the devices in a mirror or RAID-Z configuration. If the replacement device is larger, the pool size in an unmirrored or non RAID-Z configuration is increased when the replacement is complete.

For more information about replacing devices, see "Repairing a Missing Device" on page 143 and "Repairing a Damaged Device" on page 145.

Designating Hot Spares in Your Storage Pool

The hot spares feature enables you to identify disks that could be used to replace a failed or faulted device in one or more storage pools. Designating a device as a *hot spare* means that the device is not an active device in a pool, but if an active device in the pool fails, the hot spare automatically replaces the failed device.

Devices can be designated as hot spares in the following ways:

- When the pool is created with the `zpool create` command
- After the pool is created with the `zpool add` command
- Hot spare devices can be shared between multiple pools

Designate devices as hot spares when the pool is created. For example:

```
# zpool create zeepool mirror c1t1d0 c2t1d0 spare c1t2d0 c2t2d0
# zpool status zeepool
pool: zeepool
 state: ONLINE
 scrub: none requested
config:

        NAME         STATE     READ WRITE CKSUM
        zeepool      ONLINE       0     0     0
          mirror     ONLINE       0     0     0
            c1t1d0   ONLINE       0     0     0
            c2t1d0   ONLINE       0     0     0
        spares
          c1t2d0     AVAIL
          c2t2d0     AVAIL
```

Designate hot spares by adding them to a pool after the pool is created. For example:

```
# zpool add -f zeepool spare c1t3d0 c2t3d0
# zpool status zeepool
pool: zeepool
 state: ONLINE
 scrub: none requested
config:

        NAME         STATE     READ WRITE CKSUM
        zeepool      ONLINE       0     0     0
          mirror     ONLINE       0     0     0
            c1t1d0   ONLINE       0     0     0
            c2t1d0   ONLINE       0     0     0
        spares
          c1t3d0     AVAIL
          c2t3d0     AVAIL
```

Multiple pools can share devices that are designated as hot spares. For example:

```
# zpool create zeepool mirror c1t1d0 c2t1d0 spare c1t2d0 c2t2d0
# zpool create tank raidz c3t1d0 c4t1d0 spare c1t2d0 c2t2d0
```

Hot spares can be removed from a storage pool by using the `zpool remove` command. For example:

```
# zpool remove zeepool c1t2d0
# zpool status zeepool
pool: zeepool
 state: ONLINE
 scrub: none requested
config:

        NAME         STATE     READ WRITE CKSUM
        zeepool      ONLINE       0     0     0
          mirror     ONLINE       0     0     0
            c1t1d0   ONLINE       0     0     0
            c2t1d0   ONLINE       0     0     0
        spares
          c1t3d0     AVAIL
```

A hot spare cannot be removed if it is currently used by the storage pool.

Currently, the `zpool remove` command can only be used to remove hot spares.

Activating and Deactivating Hot Spares in Your Storage Pool

Hot spares are activated in the following ways:

- Manually replacement – Replace a failed device in a storage pool with a hot spare by using the `zpool replace` command.
- Automatic replacement – When a fault is received, an FMA agent examines the pool to see if it has any available hot spares. If so, it replaces the faulted device with an available spare.

  If a hot spare that is currently in use fails, the agent detachs the spare and thereby cancels the replacement. The agent then attempts to replace the device with another hot spare, if one is available. This feature is currently limited by the fact that the ZFS diagnosis engine only emits faults when a device disappears from the system.

  Currently, no automated response is available to bring the original device back online. You must explicitly take one of the actions described in the example below. A future enhancement will allow ZFS to subscribe to hotplug events and automatically replace the affected device when it is replaced on the system.

Manually replace a device with a hot spare by using the `zpool replace` command. For example:

```
# zpool replace zeepool c2t1d0 c2t3d0
# zpool status zeepool
  pool: zeepool
 state: ONLINE
 scrub: resilver completed with 0 errors on Fri Jun  2 13:44:40 2006
config:

        NAME            STATE     READ WRITE CKSUM
        zeepool         ONLINE       0     0     0
          mirror        ONLINE       0     0     0
            c1t2d0      ONLINE       0     0     0
            spare       ONLINE       0     0     0
              c2t1d0    ONLINE       0     0     0
              c2t3d0    ONLINE       0     0     0
        spares
          c1t3d0        AVAIL
          c2t3d0        INUSE     currently in use

errors: No known data errors
```

A faulted device is automatically replaced if a hot spare is available. For example:

```
# zpool status -x
  pool: zeepool
 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-D3
 scrub: resilver completed with 0 errors on Fri Jun  2 13:56:49 2006
config:

        NAME                STATE     READ WRITE CKSUM
        zeepool             DEGRADED     0     0     0
          mirror            DEGRADED     0     0     0
            c1t2d0          ONLINE       0     0     0
            spare           DEGRADED     0     0     0
              c2t1d0        UNAVAIL      0     0     0  cannot open
              c2t3d0        ONLINE       0     0     0
        spares
          c1t3d0            AVAIL
          c2t3d0            INUSE     currently in use

errors: No known data errors
```

Currently, three ways to deactivate hot spares are available:

- Canceling the hot spare by removing it from the storage pool

- Replacing the original device with a hot spare
- Permanently swapping in the hot spare

After the faulted device is replaced, use the `zpool detach` command to return the hot spare back to the spare set. For example:

```
# zpool detach zeepool c2t3d0
# zpool status zeepool
  pool: zeepool
 state: ONLINE
 scrub: resilver completed with 0 errors on Fri Jun  2 13:58:35 2006
config:

        NAME               STATE     READ WRITE CKSUM
        zeepool            ONLINE       0     0     0
          mirror           ONLINE       0     0     0
            c1t2e0         ONLINE       0     0     0
            c2t1d0         ONLINE       0     0     0
        spares
          c1t3d0           AVAIL
          c2t3d0           AVAIL

errors: No known data errors
```

Querying ZFS Storage Pool Status

The `zpool list` command provides a number of ways to request information regarding pool status. The information available generally falls into three categories: basic usage information, I/O statistics, and health status. All three types of storage pool information are covered in this section.

Basic ZFS Storage Pool Information

You can use the `zpool list` command to display basic information about pools.

Listing Information About All Storage Pools

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

```
# 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     -
```

This output displays 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 amount is different from the amount of space as reported at the file system level. For more information about determining available file system space, see "ZFS Space Accounting" on page 30.
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 about pool health, see "Health Status of ZFS Storage Pools" on page 54.
ALTROOT	The alternate root of the pool, if any. For more information about alternate root pools, see "ZFS Alternate Root Pools" on page 132.

You can also gather statistics for a specific pool by specifying the pool name. For example:

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

Listing Specific Storage Pool Statistics

Specific statistics can be requested by using the `-o` option. This option allows for custom reports or a quick way to list pertinent information. For example, to list only the name and size of each pool, you use the following syntax:

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

The column names correspond to the properties that are listed in "Listing Information About All Storage Pools" on page 50.

Scripting ZFS Storage Pool Output

The default output for the `zpool list` command is designed for readability, and is not easy to use as part of a shell script. 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 by spaces. For example, to request a simple list of all pool names on the system:

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

Here is another example:

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

ZFS Storage Pool I/O Statistics

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

`USED CAPACITY`	The amount of data currently stored in the pool or device. This figure differs from the amount of space available to actual file systems by a small amount due to internal implementation details. For more information about the difference between pool space and dataset space, see "ZFS Space Accounting" on page 30.
`AVAILABLE CAPACITY`	The amount of space available in the pool or device. As with the `used` statistic, this amount 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.

Listing Pool-Wide Statistics

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

```
# 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
```

Because these statistics are cumulative since boot, bandwidth might appear low if the pool is relatively idle. You can request a more accurate view of current bandwidth usage by specifying an interval. For example:

```
# 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
```

In this example, the command displays usage statistics only for the pool `tank` every two seconds until you type 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 a summary every two seconds for three iterations, for a total of six seconds. If there is a single pool, then the statistics are displayed on consecutive lines. If more than one pool exists, then an additional dashed line delineates each iteration to provide visual separation.

Listing Virtual Device Statistics

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

```
# zpool iostat -v
               capacity     operations    bandwidth
tank         used  avail   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
mirror      20.4G  59.6G      0     22      0  6.00K
  c1t0d0        -      -      1    295  11.2K   148K
  c1t1d0        -      -      1    299  11.2K   148K
----------  -----  -----  -----  -----  -----  -----
total       24.5K   149M      0     22      0  6.00K
```

Note two important things when viewing I/O statistics on a virtual device basis.

- First, 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.

- Second, the numbers might 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 difference 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.

You can use the same set of options (interval and count) when examining virtual device statistics.

Health Status of ZFS Storage Pools

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 state information is displayed by using the `zpool status` command. In addition, potential pool and device failures are reported by `fmd` and are displayed on the system console and the `/var/adm/messages` file. This section describes how to determine pool and device health. This chapter 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 might still occur, the device is otherwise in working order.
`DEGRADED`	The virtual device has experienced failure but is still able to function. This state is most common when a mirror or RAID-Z device has lost one or more constituent devices. The fault tolerance of the pool might be compromised, as a subsequent fault in another device might be unrecoverable.
`FAULTED`	The virtual device is completely inaccessible. This status 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 taken offline by the administrator.
`UNAVAILABLE`	The device or virtual device cannot be opened. In some cases, pools with `UNAVAILABLE` devices appear in `DEGRADED` mode. If a top-level virtual device is unavailable, then nothing in the pool can be accessed.

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 the virtual devices is `DEGRADED` or `UNAVAILABLE`, 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 might not achieve the same level of data replication or data throughput than if the pool were online.

Basic Storage Pool Health Status

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

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

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

Detailed Health Status

You can request a more detailed health summary by using the `-v` option. For example:

```
# 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
            c1t0d0          FAULTED      0     0     0  cannot open
            c1t1d0          ONLINE       0     0     0
errors: No known data errors
```

This output displays a complete description of why the pool is in its current state, including a 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 problem. Using the detailed configuration information, you should be able to determine which device is damaged and how to repair the pool.

In the above example, the faulted device should be replaced. After the device is replaced, use the `zpool online` command to bring the device back online. For example:

```
# zpool online tank c1t0d0
Bringing device c1t0d0 online
# zpool status -x
all pools are healthy
```

If a pool has an offlined device, the command output 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
            c1t0d0   ONLINE       0     0     0
            c1t1d0   OFFLINE      0     0     0

errors: No known data errors
```

The `READ` and `WRITE` columns provides a count of I/O errors seen on the device, while the `CKSUM` column provides a count of uncorrectable checksum errors that occurred on the device. Both of these error counts likely indicate potential device failure, and some corrective action is needed. If non-zero errors are reported for a top-level virtual device, portions of your data might have become inaccessible. The errors count identifies any known data errors.

In the example output above, the offlined device is not causing data errors.

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

Migrating ZFS Storage Pools

Occasionally, you might need 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 task can be accomplished by physically recabling the devices, or by using multiported devices such as the devices on a SAN. ZFS enables you to export the pool from one machine and import it on the destination machine, even if the machines are of different endianness. For information about replicating or migrating file systems between different storage pools, which might reside on different machines, see "Saving and Restoring ZFS Data" on page 97.

Preparing for ZFS Storage Pool Migration

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

If you do not explicitly export the pool, but instead remove the disks manually, you can still import the resulting pool on another system. However, you might lose the last few seconds of data transactions, and the pool will appear faulted on the original machine because the devices are no longer present. By default, the destination machine refuses to import a pool that has not been explicitly exported. This condition is necessary to prevent accidentally importing an active pool that consists of network attached storage that is still in use on another system.

Exporting a ZFS Storage Pool

To export a pool, use the `zpool export` command. For example:

```
# 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 file systems within the pool before continuing. If any of the file systems fail to unmount, you can forcefully unmount them by using the `-f` option. For example:

```
# 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 appears as "potentially active." If emulated volumes are in use in the pool, the pool cannot be exported, even with the `-f` option. To export a pool with an emulated volume, first make sure that all consumers of the volume are no longer active.

For more information about emulated volumes, see "Emulated Volumes" on page 127.

Determining Available Storage Pools to Import

Once the pool has been removed from the system (either through export or by forcefully removing the devices), attach the devices to the target system. Although ZFS can handle some situations in which only a portion of the devices is available, 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. For example:

```
# zpool import
  pool: tank
    id: 3778921145927357706
 state: ONLINE
action: The pool can be imported using its name or numeric identifier.
config:
```

```
        tank        ONLINE
          mirror    ONLINE
            c1t0d0  ONLINE
            c1t1d0  ONLINE
```

In this 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. If multiple pools available to import have the same name, you can use the numeric identifier to distinguish between them.

Similar to the `zpool status` command, the `zpool import` command refers to a knowledge article available on the web with the most up-to-date information regarding repair procedures for a problem that is preventing a pool from being imported. In this case, the user can force the pool to be imported. However, 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 devices in the pool are not available but enough redundancy is available to have a usable pool, the pool appears in the `DEGRADED` state. For example:

```
# zpool import
  pool: tank
    id: 3778921145927357706
 state: DEGRADED
status: One or more devices are missing from the system.
action: The pool can be imported despite missing or damaged devices.  The
        fault tolerance of the pool may be compromised if imported.
   see: http://www.sun.com/msg/ZFS-8000-2Q
config:

        tank         DEGRADED
          mirror     DEGRADED
            c1t0d0   UNAVAIL   cannot open
            c1t1d0   ONLINE
```

In this example, the first disk is damaged or missing, though you can still import the pool because the mirrored data is still accessible. If too many faulted or missing devices are present, the pool cannot be imported. For example:

```
# zpool import
  pool: dozer
    id: 12090808386336829175
 state: FAULTED
action: The pool cannot be imported. Attach the missing
        devices and try again.
   see: http://www.sun.com/msg/ZFS-8000-6X
config:
        raidz               FAULTED
          c1t0d0    ONLINE
```

```
          c1t1d0    FAULTED
          c1t2d0    ONLINE
          c1t3d0    FAULTED
```

In this example, two disks are missing from a RAID-Z virtual device, which means that sufficient replicated data is not available to reconstruct the pool. In some cases, 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 ZFS does report as much information as possible about the situation. For example:

```
# zpool import
pool: dozer
    id: 12090808386336829175
 state: FAULTED
status: One or more devices are missing from the system.
action: The pool cannot be imported. Attach the missing
        devices and try again.
   see: http://www.sun.com/msg/ZFS-8000-6X
config:
        dozer          FAULTED   missing device
          raidz       ONLINE
            c1t0d0    ONLINE
            c1t1d0    ONLINE
            c1t2d0    ONLINE
            c1t3d0    ONLINE
        Additional devices are known to be part of this pool, though their
        exact configuration cannot be determined.
```

Finding ZFS Storage Pools From Alternate Directories

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

```
# 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 devices exist in multiple directories, you can specify multiple `-d` options.

Importing ZFS Storage Pools

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

```
# zpool import tank
```

If multiple available pools have the same name, you can specify which pool to import using the numeric identifier. For example:

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

        dozer       ONLINE
          c1t9d0    ONLINE

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

        dozer       ONLINE
          c1t8d0    ONLINE
# zpool import dozer
cannot import 'dozer': more than one matching pool
import by numeric ID instead
# zpool import 6223921996155991199
```

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

```
# zpool import dozer zeepool
```

This command imports the exported pool `dozer` using the new name `zeepool`. 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. For example:

```
# 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 by using the `-R` option. For more information on alternate root pools, see "ZFS Alternate Root Pools" on page 132.

Recovering Destroyed ZFS Storage Pools

You can use the `zpool import -D` command to recover a storage pool that has been destroyed. For example:

```
# zpool destroy tank
# zpool import -D
pool: tank
    id: 3778921145927357706
 state: ONLINE (DESTROYED)
action: The pool can be imported using its name or numeric identifier.  The
        pool was destroyed, but can be imported using the '-Df' flags.
config:

        tank        ONLINE
          mirror    ONLINE
            c1t0d0  ONLINE
            c1t1d0  ONLINE
```

In the above `zpool import` output, you can identify this pool as the destroyed pool because of the following state information:

```
state: ONLINE (DESTROYED)
```

To recover the destroyed pool, issue the `zpool import -D` command again with the pool to be recovered and the `-f` option. For example:

```
# zpool import -Df tank
# zpool status tank
  pool: tank
 state: ONLINE
 scrub: none requested
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

errors: No known data errors
```

If one of the devices in the destroyed pool is faulted or unavailable, you might be able to recover the destroyed pool anyway. In this scenario, import the degraded pool and then attempt to fix the device failure. For example:

```
# zpool destroy dozer
# zpool import -D
pool: dozer
    id:
 state: DEGRADED (DESTROYED)
status: One or more devices are missing from the system.
action: The pool can be imported despite missing or damaged devices.  The
        fault tolerance of the pool may be compromised if imported.  The
        pool was destroyed, but can be imported using the '-Df' flags.
   see: http://www.sun.com/msg/ZFS-8000-2Q
config:

        dozer        DEGRADED
           raidz      ONLINE
            c1t0d0    ONLINE
            c1t1d0    ONLINE
            c1t2d0    UNAVAIL  cannot open
            c1t3d0    ONLINE
# zpool import -Df dozer
# zpool status -x
  pool: dozer
 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-D3
 scrub: resilver completed with 0 errors on Fri Mar 17 16:11:35 2006
config:

        NAME                    STATE     READ WRITE CKSUM
        dozer                   DEGRADED     0     0     0
          raidz                 ONLINE       0     0     0
            c1t0d0              ONLINE       0     0     0
            c1t1d0              ONLINE       0     0     0
            c1t2d0              UNAVAIL      0     0     0  cannot open
```

```
            c1t3d0              ONLINE       0     0     0

errors: No known data errors
# zpool online dozer c1t2d0
Bringing device c1t2d0 online
# zpool status -x
all pools are healthy
```

Upgrading ZFS Storage Pools

If you have ZFS storage pools from a previous Solaris release, such as the Solaris 10 6/06 release, you can upgrade your pools with the `zpool upgrade` command to take advantage of the pool features in the Solaris 10 11/06 release. In addition, the `zpool status` command has been modified to notify you when your pools are running older versions. For example:

```
# zpool status
  pool: test
 state: ONLINE
status: The pool is formatted using an older on-disk format.  The pool can
        still be used, but some features are unavailable.
action: Upgrade the pool using 'zpool upgrade'.  Once this is done, the
        pool will no longer be accessible on older software versions.
 scrub: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        test        ONLINE       0     0     0
          c1t27d0   ONLINE       0     0     0

errors: No known data errors
```

You can use the following syntax to identify additional information about a particular version and supported releases.

```
# zpool upgrade -v
This system is currently running ZFS version 3.

The following versions are supported:

VER  DESCRIPTION
---  --------------------------------------------------------
 1   Initial ZFS version
 2   Ditto blocks (replicated metadata)
 3   Hot spares and double parity RAID-Z

For more information on a particular version, including supported releases, see:
```

```
http://www.opensolaris.org/os/community/zfs/version/N

Where 'N' is the version number.
```

Then, you can run the `zpool upgrade` command to upgrade your pools. For example:

```
# zpool upgrade -a
```

Note – If you upgrade your pools to the latest version, they will not be accessible on systems that run older ZFS versions.

◆ ◆ ◆ CHAPTER 5

Managing ZFS File Systems

This chapter provides detailed information about managing Solaris™ ZFS file systems. Concepts such as hierarchical file system layout, property inheritance, and automatic mount point management and share interactions are included in this chapter.

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

ZFS file systems are administered by using the `zfs` command. The `zfs` command provides a set of subcommands that perform specific operations on file systems. This chapter describes these subcommands in detail. Snapshots, volumes, and clones are also managed by using this command, but these features are only covered briefly in this chapter. For detailed information about snapshots and clones, see Chapter 6. For detailed information about emulated volumes, see "Emulated Volumes" on page 127.

Note – The term *dataset* is used in this chapter as a generic term to refer to a file system, snapshot, clone, or volume.

The following sections are provided in this chapter:

Creating and Destroying ZFS File Systems

ZFS file systems can be created and destroyed by using the `zfs create` and `zfs destroy` commands.

Creating a ZFS File System

ZFS file systems are created by using the `zfs create` command. The `create` subcommand takes a single argument: the name of the file system to create. The file system 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 file system names in the path identify the location in the hierarchy where the new file system will be created. All the intermediate file system names must already exist in the pool. The last name in the path identifies the name of the file system to be created. The file system name must satisfy the naming conventions defined in "ZFS Component Naming Requirements" on page 21.

In the following example, a file system named `bonwick` is created in the `tank/home` file system.

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

ZFS automatically mounts the newly created file system if it is created successfully. By default, file systems are mounted as */dataset*, using the path provided for the file system name in the `create` subcommand. In this example, the newly created `bonwick` file system is at `/tank/home/bonwick`. For more information about automanaged mount points, see "Managing ZFS Mount Points" on page 81.

For more information about the `zfs create` command, see `zfs`(1M).

Destroying a ZFS File System

To destroy a ZFS file system, use the `zfs destroy` command. The destroyed file system is automatically unmounted and unshared. For more information about automanaged mounts or automanaged shares, see "Automatic Mount Points" on page 82.

In the following example, the `tabriz` file system is destroyed.

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

Caution – No confirmation prompt appears with the `destroy` subcommand. Use it with extreme caution.

If the file system to be destroyed is busy and so cannot be unmounted, the `zfs destroy` command fails. To destroy an active file system, use the `-f` option. Use this option with caution as it can unmount, unshare, and destroy active file systems, 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 file system has children. To recursively destroy a file system and all its descendants, use the `-r` option. Note that a recursive destroy also destroys snapshots so use this option with caution.

```
# 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 file system to be destroyed has indirect dependents, even the recursive destroy command described above fails. To force the destruction of *all* dependents, including cloned file systems outside the target hierarchy, the `-R` option must be used. Use extreme caution with this option.

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

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

Caution – No confirmation prompt appears with the `-f`, `-r`, or `-R` options so use these options carefully.

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

Renaming a ZFS File System

File systems can be renamed by using the `zfs rename` command. Using the `rename` subcommand can perform the following operations:

- Change the name of a file system
- Relocate the file system to a new location within the ZFS hierarchy
- Change the name of a file system and relocate it with the ZFS hierarchy

The following example uses the `rename` subcommand to do a simple rename of a file system:

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

This example renames the `kustarz` file system to `kustarz_old`.

The following example shows how to use `zfs rename` to relocate a file system.

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

In this example, the `maybee` file system is relocated from `tank/home` to `tank/ws`. When you relocate a file system through rename, the new location must be within the same pool and it must have enough space to hold this new file system. If the new location does not have enough space, possibly because it has reached its quota, the rename will fail.

For more information about quotas, see "ZFS Quotas and Reservations" on page 87.

The rename operation attempts an unmount/remount sequence for the file system and any descendant file systems. The rename fails if the operation is unable to unmount an active file system. If this problem occurs, you will need to force unmount the file system.

For information about renaming snapshots, see "Renaming ZFS Snapshots" on page 93.

ZFS Properties

Properties are the main mechanism that you use to control the behavior of file systems, volumes, snapshots, and clones. Unless stated otherwise, the properties defined in the section apply to all the dataset types.

Properties are either read-only statistics or settable properties. Most settable properties are also inheritable. An inheritable property is a property that, when set on a parent, is propagated down to all of its descendants.

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

`local`	A `local` source indicates that the property was explicitly set on the dataset by using the `zfs set` command as described in "Setting ZFS Properties" on page 77.
`inherited from` *dataset-name*	A value of `inherited from` *dataset-name* means that the property was inherited from the named ancestor.
`default`	A value of `default` means that the property setting was not inherited or set locally. This source is a result of no ancestor having the property as source `local`.

The following table identifies both read-only and settable ZFS file system properties. Read-only properties are identified as such. All other properties are settable.

TABLE 5–1 ZFS Property Descriptions

Property 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 "ACL Property Modes" on page 106.
`aclmode`	String	`groupmask`	Controls how an ACL entry is modified during a `chmod` operation. The values are `discard`, `groupmask`, and `passthrough`. For a description of these values, see "ACL Property Modes" on page 106.
`atime`	Boolean	`on`	Controls whether the access time for files is updated when they are read. Turning this property off avoids producing write traffic when reading files and can result in significant performance gains, though it might confuse mailers and other similar utilities.
`available`	Number	N/A	Read-only property that identifies 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, available space can be limited by various factors including physical pool size, quotas, reservations, or other datasets within the pool. This property can also be referenced by its shortened column name, `avail`. For more information about space accounting, see "ZFS Space Accounting" on page 30.
`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 `on, off, fletcher2`, `fletcher4`, and `sha256`. A value of `off` disables integrity checking on user data. A value of `off` is not recommended.
`compression`	String	`off`	Controls the compression algorithm used for this dataset. Currently, only one algorithm, `lzjb`, exists. This property can also be referred to by its shortened column name, `compress`.

TABLE 5–1 ZFS Property Descriptions *(Continued)*

Property Name	Type	Default Value	Description
`compressratio`	Number	N/A	Read-only property that identifies the compression ratio achieved for this dataset, expressed as a multiplier. Compression can be turned on by running `zfs set compression=on` *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.
`creation`	Number	N/A	Read-only property that identifies the date and time that this dataset was created.
`devices`	Boolean	`on`	Controls whether device nodes found within this file system can be opened.
`exec`	Boolean	`on`	Controls whether programs within this file system are allowed to be executed. Also, when set to `off`, `mmap(2)` calls with `PROT_EXEC` are disallowed.
`mounted`	boolean	N/A	Read-only property that indicates whether this file system, clone, or snapshot is currently mounted. This property does not apply to volumes. Value can be either yes or no.
`mountpoint`	String	N/A	Controls the mount point used for this file system. When the `mountpoint` property is changed for a file system, the file system 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 about using this property, see "Managing ZFS Mount Points" on page 81.
`origin`	String	N/A	Read-only property for cloned file systems or volumes that identifies the snapshot from which the clone was created. The origin cannot be destroyed (even with the `-r` or `-f` options) as long as a clone exists. Non-cloned file systems have an origin of none.

TABLE 5–1 ZFS Property Descriptions *(Continued)*

Property Name	Type	Default Value	Description
`quota`	Number (or `none`)	`none`	Limits the amount of space a dataset and its descendents can consume. This property enforces a hard limit on the amount of space used, including 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 "Setting Quotas on ZFS File Systems" on page 87.
`readonly`	Boolean	`off`	Controls whether this dataset can be modified. When set to `on`, no modifications can be made to the dataset. 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. This property can also be referred to by its shortened column name, `recsize`. For a detailed description, see "The `recordsize` Property" on page 74.
`referenced`	Number	N/A	Read-only property that identifies the amount of data accessible by this dataset, which might or might 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, because its contents are identical. This property can also be referred to by its shortened column name, `refer`.
`reservation`	Number (or `none`)	`none`	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 using 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 "Setting Reservations on ZFS File Systems" on page 88.

TABLE 5–1 ZFS Property Descriptions *(Continued)*

Property Name	Type	Default Value	Description
`sharenfs`	String	`off`	Controls whether the file system is available over NFS, and what options are used. If set to `on`, the `zfs share` command is invoked with no options. Otherwise, the `zfs share` command is invoked with options equivalent to the contents of this property. If set to `off`, the file system is managed by using the legacy `share` and `unshare` commands and the `dfstab` file. For more information on sharing ZFS file systems, see "Sharing ZFS File Systems" on page 85.
`setuid`	Boolean	`on`	Controls whether the `setuid` bit is honored in the file system.
`snapdir`	String	`hidden`	Controls whether the `.zfs` directory is hidden or visible in the root of the file system. For more information on using snapshots, see "ZFS Snapshots" on page 91.
`type`	String	N/A	Read-only property that identifies the dataset type as `filesystem` (file system or clone), `volume`, or `snapshot`.
`used`	Number	N/A	Read-only property that identifies the amount of space consumed by the dataset and all its descendants. For a detailed description, see "The `used` Property" on page 73.
`volsize`	Number	N/A	For volumes, specifies the logical size of the volume. For a detailed description, see "The `volsize` Property" on page 75.
`volblocksize`	Number	8 Kbytes	For volumes, specifies the block size of the volume. The block size cannot be changed once the volume has been written, so set the block size at volume creation time. The default block size 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`	Boolean	N/A	Indicates whether this dataset has been added to a non-global zone. If this property is set, then the mount point is not honored in the global zone, and ZFS cannot mount such a file system when requested. When a zone is first installed, this property is set for any added file systems. For more information about using ZFS with zones installed, see "Using ZFS on a Solaris System With Zones Installed" on page 128.

Read-Only ZFS 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 mentioned in the description in Table 5–1.

The read-only properties are listed here and are described in Table 5–1.

- `available`
- `creation`
- `mounted`
- `origin`
- `compressratio`
- `referenced`
- `type`
- `used`

  For detailed information, see "The `used` Property" on page 73.

For more information on space accounting, including the `used`, `referenced`, and `available` properties, see "ZFS Space Accounting" on page 30.

The `used` Property

The amount of space consumed by this dataset and all its descendants. This value is checked against the dataset's quota and reservation. The space used does not include the dataset's reservation, but does consider the reservation 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 the 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 "Out of Space Behavior" on page 30.

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.

Settable ZFS Properties

Settable properties are properties whose values can be both retrieved and set. Settable properties are set by using the `zfs set` command, as described in "Setting ZFS Properties" on page 77. With the

exceptions of quotas and reservations, settable properties are inherited. For more information about quotas and reservations, see "ZFS Quotas and Reservations" on page 87.

Some settable properties are specific to a particular type of dataset. In such cases, the particular dataset type is mentioned in the description in Table 5–1. If not specifically mentioned, a property applies to all dataset types: file systems, volumes, clones, and snapshots.

The settable properties are listed here and are described in Table 5–1.

- `aclinherit`

  For a detailed description, see "ACL Property Modes" on page 106.
- `aclmode`

  For a detailed description, see "ACL Property Modes" on page 106.
- `atime`
- `checksum`
- `compression`
- `devices`
- `exec`
- `mountpoint`
- `quota`
- `readonly`
- `recordsize`

  For a detailed description, see "The `recordsize` Property" on page 74.
- `reservation`
- `sharenfs`
- `setuid`
- `snapdir`
- `volsize`

  For a detailed description, see "The `volsize` Property" on page 75.
- `volblocksize`
- `zoned`

The `recordsize` Property

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 adjust block sizes according to internal algorithms optimized for typical access patterns. For databases that create very large files but access the files 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 file system's `recordsize` only affects files created afterward. Existing files are unaffected.

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

The `volsize` Property

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 users. A volume that 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 you shrink 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 "Emulated Volumes" on page 127.

Querying ZFS File System 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.

Listing Basic ZFS Information

You can list basic dataset information by using the `zfs list` command with no options. This command displays the names of all datasets on the system including their `used`, `available`, `referenced`, and `mountpoint` properties. For more information about these properties, see "ZFS Properties" on page 68.

For example:

```
# 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/home/marks@snap
```

You can also use this command to display specific datasets by providing the dataset name on the command line. Additionally, use the `-r` option to recursively display all descendants of that dataset.

The following example shows how 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
```

For additional information about the `zfs list` command, see `zfs`(1M).

Creating Complex ZFS Queries

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

You can customize property value output by using the `-o` option and a comma-separated list of desired properties. Supply any dataset property as a valid value. For a list of all supported dataset properties, see "ZFS Properties" on page 68. In addition to the properties defined there, the `-o` option list can also contain the literal `name` to indicate that the output should include the name of the dataset.

The following example 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                  off              /tank
tank/home             on               /tank/home
tank/home/ahrens      on               /tank/home/ahrens
tank/home/bonwick     on               /tank/home/bonwick
tank/home/chua        on               /tank/home/chua
tank/home/eschrock    on               legacy
tank/home/moore       on               /tank/home/moore
tank/home/tabriz      ro               /tank/home/tabriz
```

You can use the `-t` option to specify the types of datasets to display. The valid types are described in the following table.

TABLE 5–2 Types of ZFS Datasets

Type	Description
`filesystem`	File systems and clones

TABLE 5–2 Types of ZFS Datasets *(Continued)*

Type	Description
`volume`	Volumes
`snapshot`	Snapshots

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

```
# 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
```

You can use the `-H` option to omit the `zfs list` header from the generated output. With the `-H` option, all white space is output as tabs. This option can be useful when you need parseable output, for example, when scripting. The following example shows the output generated from using the `zfs list` command with 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
```

Managing ZFS Properties

Dataset properties are managed through the `zfs` command's `set`, `inherit`, and `get` subcommands.

Setting ZFS Properties

You can use the `zfs set` command to modify any settable dataset property. For a list of settable dataset properties, see "Settable ZFS Properties" on page 73. The `zfs set` command takes a property/value sequence in the format of *property=value* and a dataset name.

The following example sets the `atime` property to `off` for `tank/home`. Only one property can be set or modified during each `zfs set` invocation.

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

You can specify numeric properties by using the following easy to understand 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 be set to the value of 50 Gbytes on the `tank/home/marks` file system:

```
# 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
```

Values of non-numeric properties are case-sensitive and must be lowercase, with the exception of `mountpoint` and `sharenfs`. The values of these properties can have mixed upper and lower case letters.

For more information about the `zfs set` command, see `zfs`(1M).

Inheriting ZFS Properties

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

The following example uses the `zfs set` command to turn on compression for the `tank/home/bonwick` file system. Then, `zfs inherit` is used to unset the `compression` property, thus causing the property to inherit the default setting of `off`. Because neither `home` nor `tank` have the `compression` property set locally, the default value is used. If both had compression on, 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. In the following example, the command causes the value for the `compression` property to be inherited by `tank/home` and any descendants it might have.

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

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

For more information about the `zfs` command, see `zfs`(1M).

Querying ZFS Properties

The simplest way to query property values is by using the `zfs list` command. For more information, see "Listing Basic ZFS Information" on page 75. However, for complicated queries and for scripting, use the `zfs get` command to provide more detailed information in a customized format.

You can use the `zfs get` command to retrieve any dataset property. The following example 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 following table defines the meaning of the possible source values.

TABLE 5–3 Possible `SOURCE` Values (`zfs get`)

Source 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 as specified by *dataset-name*.
`local`	This property value was explicitly set for this dataset by using `zfs set`.
`temporary`	This property value was set by using the `zfs mount -o` option and is only valid for the lifetime of the mount. For more information about temporary mount point properties, see "Temporary Mount Properties" on page 84.
- (none)	This property is a read-only property. Its value is generated by ZFS.

You can use the special keyword `all` to retrieve all dataset properties. The following example uses the `all` keyword to retrieve all existing dataset properties:

```
# zfs get all pool
NAME             PROPERTY       VALUE                      SOURCE
pool             type           filesystem                 -
pool             creation       Mon Mar 13 11:41 2006      -
pool             used           2.62M                      -
pool             available      33.5G                      -
pool             referenced     10.5K                      -
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    off                        default
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        hidden                     default
pool             aclmode        groupmask                  default
pool             aclinherit     secure                     default
```

The `-s` option to `zfs get` enables you to specify, by source value, the type of properties to display. This option takes a comma-separated list indicating the desired source types. Only properties with the specified source type are displayed. The valid source types are `local`, `default`, `inherited`, `temporary`, and `none`. The following example 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 display the specified properties on all children of the specified dataset. In the following example, all temporary properties on all datasets within `tank` are recursively displayed:

```
# 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
```

For more information about the `zfs get` command, see `zfs`(1M).

Querying ZFS Properties for Scripting

The `zfs get` command supports the `-H` and `-o` options, which are designed for scripting. The `-H` option indicates that any header information should be omitted and that all white space should come in the form of tab. Uniform white space allows for easily parseable data. You can use the `-o` option to customize the output. This option takes a comma-separated list of values to be output. All properties defined in "ZFS Properties" on page 68, 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 by using the `-H` and `-o` options of `zfs get`.

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

The `-p` option 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
```

You can use the `-r` option along with any of the above options to recursively retrieve the requested values for all descendants. The following example uses the `-r`, `-o`, and `-H` options to retrieve 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
```

Mounting and Sharing ZFS File Systems

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

Managing ZFS Mount Points

By default, all ZFS file systems are mounted by ZFS at boot by using SMF's `svc://system/filesystem/local` service. File systems are mounted under */path*, where *path* is the name of the file system.

You can override the default mount point by setting the `mountpoint` property to a specific path by using the `zfs set` command. ZFS automatically creates this mount point, if needed, and automatically mounts this file system when the `zfs mount -a` command is invoked, without requiring you 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 its `mountpoint` property.

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

If desired, file systems can also be explicitly managed through legacy mount interfaces by setting the `mountpoint` property to `legacy` by using `zfs set`. Doing so prevents ZFS from automatically mounting and managing this file system. Legacy tools including the `mount` and `umount` commands, and the `/etc/vfstab` file must be used instead. For more information about legacy mounts, see "Legacy Mount Points" on page 83.

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

- Automatic mount point behavior
- Legacy mount point behavior

Automatic Mount Points

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

You can also set the default mount point for the root dataset at creation time by using `zpool create`'s `-m` option. For more information about creating pools, see "Creating a ZFS Storage Pool" on page 38.

Any dataset whose `mountpoint` property is not `legacy` is managed by ZFS. In the following example, a dataset is created whose mount point is automatically 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                       -
```

You can also explicitly set the `mountpoint` property as shown in the following example:

```
# 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 file system 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 file system due to it being active, an error is reported and a forced manual unmount is necessary.

Legacy Mount Points

You can manage ZFS file systems with legacy tools by setting the mountpoint property to legacy. Legacy file systems must be managed through the mount and umount commands and the /etc/vfstab file. ZFS does not automatically mount legacy file systems on boot, and the ZFS mount and umount command do not operate on datasets of this type. The following examples 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 must indicate that they are legacy file systems. In addition, 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 file system on boot, you must add an entry to the /etc/vfstab file. 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 syntax is because the fsck command is not applicable to ZFS file systems. For more information regarding data integrity and the lack of need for fsck in ZFS, see "Transactional Semantics" on page 18.

Mounting ZFS File Systems

ZFS automatically mounts file systems when file systems are created or when the system boots. Use of the zfs mount command is necessary only when changing mount options or explicitly mounting or unmounting file systems.

The zfs mount command with no arguments shows all currently mounted file systems that are managed by ZFS. Legacy managed mount points are not displayed. For example:

```
# zfs mount
tank                            /tank
tank/home                       /tank/home
tank/home/bonwick               /tank/home/bonwick
tank/ws                         /tank/ws
```

You can use the `-a` option to mount all ZFS managed file systems. Legacy managed file systems are not mounted. For example:

```
# zfs mount -a
```

By default, ZFS does not allow mounting on top of a nonempty directory. To force a mount on top of a nonempty directory, you must use the `-O` option. For example:

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

Legacy mount points must be managed through legacy tools. An attempt to use ZFS tools results in an error. For example:

```
# 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 file system 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`.

Temporary Mount Properties

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

In the following example, the read-only mount option is temporarily set on the `tank/home/perrin` file system:

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

In this example, the file system is assumed to be unmounted. To temporarily change a property on a file system that is currently mounted, you must use the special `remount` option. In the following example, the `atime` property is temporarily changed to `off` for a file system 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
```

For more information about the `zfs mount` command, see `zfs`(1M).

Unmounting ZFS File Systems

You can unmount file systems by using the `zfs unmount` subcommand. The `unmount` command can take either the mount point or the file system name as arguments.

In the following example, a file system is unmounted by file system name:

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

In the following example, the file system is unmounted by mount point:

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

The `unmount` command fails if the file system is active or busy. To forceably unmount a file system, you can use the `-f` option. Be cautious when forceably unmounting a file system, 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` command can be used to unmount ZFS file systems. For example:

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

For more information about the `zfs umount` command, see `zfs`(1M).

Sharing ZFS File Systems

Similar to mount points, ZFS can automatically share file systems by using the `sharenfs` property. Using this method, you do not have to modify the `/etc/dfs/dfstab` file when a new file system is

added. The sharenfs property is a comma-separated list of options to pass to the share 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 file system is not managed by ZFS and can be shared through traditional means, such as the /etc/dfs/dfstab file. All file systems whose sharenfs property is not off are shared during boot.

Controlling Share Semantics

By default, all file systems are unshared. To share a new file system, use zfs set syntax similar to the following:

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

The property is inherited, and file systems 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/bricker and tank/home/tabriz are initially shared writable because they inherit the sharenfs property from tank/home. Once the property is set to ro (readonly), tank/home/tabriz is shared read-only regardless of the sharenfs property that is set for tank/home.

Unsharing ZFS File Systems

While most file systems are automatically shared and unshared during boot, creation, and destruction, file systems sometimes need to be explicitly unshared. To do so, use the zfs unshare command. For example:

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

This command unshares the tank/home/tabriz file system. To unshare all ZFS file systems on the system, you need to use the -a option.

```
# zfs unshare -a
```

Sharing ZFS File Systems

Most of the time the automatic behavior of ZFS, sharing on boot and creation, is sufficient for normal operation. If, for some reason, you unshare a file system, you can share it again by using the zfs share command. For example:

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

You can also share all ZFS file systems on the system by using the `-a` option.

```
# zfs share -a
```

Legacy Share Behavior

If the `sharenfs` property is `off`, then ZFS does not attempt to share or unshare the file system at any time. This setting enables you to administer 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 file systems. As a result, you can manually share a file system with options that are different from the settings of the `sharenfs` property. This administrative model is discouraged. 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. However, in some cases, you might still want to control file system sharing behavior through the familiar model.

ZFS Quotas and Reservations

ZFS supports quotas and reservations at the file system level. You can use the `quota` property to set a limit on the amount of space a file system can use. In addition, you can use the `reservation` property to guarantee that some amount of space is available to a file system. Both properties apply to the dataset they are set on and all descendants of that dataset.

That is, if a quota is set on the `tank/home` dataset, the total amount of 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.

Setting Quotas on ZFS File Systems

ZFS quotas can be set and displayed by using the `zfs set` and `zfs get` commands. In the following example, a quota of 10 Gbytes is set 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. For example:

```
# 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 `tank/home/bonwick/ws` only have 10 Gbytes of space available, due to the quota on `tank/home/bonwick`.

You cannot set a quota to an amount less than is currently being used by a dataset. For example:

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

Setting Reservations on ZFS File Systems

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

```
# 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 affect the output of the `zfs list` command. For example:

```
# 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` is using 5 Gbytes of space, although the total amount of 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 considered in the used space of the parent dataset and do count against its quota, reservation, or both.

```
# zfs set quota=5G pool/filesystem
# zfs set reservation=10G pool/filesystem/user1
```

```
cannot set reservation for 'pool/filesystem/user1': size is greater than
available space
```

A dataset can use more space than its reservation, as long as space is available in the pool that is unreserved and the dataset's current usage 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 reservation. Rather, the second reservation replaces the first reservation.

```
# 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
```

◆ ◆ ◆ CHAPTER 6

Working With ZFS Snapshots and Clones

This chapter describes how to create and manage ZFS snapshots and clones. Information about saving snapshots is also provided in this chapter.

The following sections are provided in this chapter:

ZFS Snapshots

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

ZFS snapshots include the following features:

- Provides persistence across system reboots.
- The theoretical maximum number of snapshots is 2^{64}.
- Uses no separate backing store. Snapshots consume disk 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 about backing up a ZFS snapshot, see "Saving and Restoring ZFS Data" on page 97.

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 "ZFS Component Naming Requirements" on page 21.

In the following example, a snapshot of `tank/home/ahrens` that is named `friday` is created.

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

You can create snapshots for all descendant file systems by using the `-r` option. For example:

```
# zfs snapshot -r tank/home@now
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
tank/home@now             0      -  29.5K  -
tank/home/ahrens@now      0      -  2.15M  -
tank/home/anne@now        0      -  1.89M  -
tank/home/bob@now         0      -  1.89M  -
tank/home/cindys@now      0      -  2.15M  -
```

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

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

Snapshots are destroyed by using the `zfs destroy` command. For example:

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

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

```
# zfs destroy tank/home/ahrens
cannot destroy 'tank/home/ahrens': filesystem has children
use '-r' to destroy the following datasets:
tank/home/ahrens@tuesday
tank/home/ahrens@wednesday
tank/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 about the `destroy` subcommand, see "Destroying a ZFS File System" on page 66.

Renaming ZFS Snapshots

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

```
# zfs rename tank/home/cindys@083006 tank/home/cindys@today
```

The following snapshot rename operation is not supported because the target pool and file system name are different from the pool and file system where the snapshot was created.

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

Displaying and Accessing ZFS Snapshots

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

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

You can list snapshots as follows:

```
# zfs list -t snapshot
NAME                         USED  AVAIL  REFER  MOUNTPOINT
pool/home/anne@monday          0      -   780K  -
pool/home/bob@monday           0      -  1.01M  -
tank/home/ahrens@tuesday    8.50K      -   780K  -
tank/home/ahrens@wednesday  8.50K      -  1.01M  -
tank/home/ahrens@thursday      0      -  1.77M  -
tank/home/cindys@today      8.50K      -   524K  -
```

You can list snapshots that were created for a particular file system as follows:

```
# zfs list -r -t snapshot -o name,creation tank/home
NAME                        CREATION
tank/home@now                Wed Aug 30 10:53 2006
tank/home/ahrens@tuesday     Wed Aug 30 10:53 2006
tank/home/ahrens@wednesday   Wed Aug 30 10:54 2006
tank/home/ahrens@thursday    Wed Aug 30 10:53 2006
tank/home/cindys@now         Wed Aug 30 10:57 2006
```

Snapshot Space Accounting

When a snapshot is created, its 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 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 file system's was when the snapshot was created.

Rolling Back to a ZFS Snapshot

The `zfs rollback` command can be used to discard all changes made since a specific snapshot. The file system reverts to its state at the time the snapshot was taken. By default, the command cannot roll back to a snapshot other than the most recent snapshot.

To roll back to an earlier snapshot, all intermediate snapshots must be destroyed. You can destroy earlier snapshots by specifying the `-r` option.

If clones of any intermediate snapshots exist, the `-R` option must be specified to destroy the clones as well.

Note – The file system that you want to roll back must be unmounted and remounted, if it is currently mounted. If the file system cannot be unmounted, the rollback fails. The `-f` option forces the file system to be unmounted, if necessary.

In the following example, the `tank/home/ahrens` file system is rolled back to the `tuesday` snapshot:

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

In the above example, the `wednesday` and `thursday` snapshots are removed because you rolled back to the previous `tuesday` snapshot.

```
# zfs list -r -t snapshot -o name,creation tank/home/ahrens
NAME                     CREATION
tank/home/ahrens@tuesday  Wed Aug 30 10:53 2006
```

ZFS Clones

A *clone* is a writable volume or file system whose initial contents are the same as the dataset from which it was created. As with snapshots, creating a clone is nearly instantaneous, and initially consumes no additional disk space.

Clones can only be created from a snapshot. When a snapshot is cloned, an implicit dependency is created 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 it was created. Rather, clones inherit their properties based on where the clones 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 "Setting ZFS Properties" on page 77.

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

Creating a ZFS Clone

To create a clone, use the `zfs clone` command, specifying the snapshot from which to create the clone, and the name of the new file system or volume. The new file system or volume can be located anywhere in the ZFS hierarchy. The type of the new dataset (for example, file system or volume) is the same type as the snapshot from which the clone was created. You cannot create clone of a file system in a pool that is different from where the original file system snapshot resides.

In the following example, a new clone named `tank/home/ahrens/bug123` with the same initial contents as the snapshot `tank/ws/gate@yesterday` is created.

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

In the following example, a cloned workspace is created from the `projects/newproject@today` snapshot for a temporary user as `projects/teamA/tempuser`. Then, properties are set on the cloned workspace.

```
# 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
```

Destroying a ZFS Clone

ZFS clones are destroyed by using the `zfs destroy` command. For example:

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

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

Replacing a ZFS File System With a ZFS Clone

You can use the `zfs promote` command to replace an active ZFS file system with a clone of that file system. This feature facilitates the ability to clone and replace file systems so that the "origin" file system become the clone of the specified file system. In addition, this feature makes it possible to destroy the file system from which the clone was originally created. Without clone promotion, you cannot destroy a "origin" file system of active clones. For more information about destroying clones, see "Destroying a ZFS Clone" on page 96.

In the following example, the `tank/test/productA` file system is cloned and then the clone file system, `tank/test/productAbeta` becomes the `tank/test/productA` file system.

```
# zfs create tank/test
# zfs create tank/test/productA
# zfs snapshot tank/test/productA@today
# zfs clone tank/test/productA@today tank/test/productAbeta
# zfs list -r tank/test
NAME                   USED  AVAIL  REFER  MOUNTPOINT
tank/test              314K  8.24G  25.5K  /tank/test
tank/test/productA     288K  8.24G   288K  /tank/test/productA
tank/test/productA@today      0      -   288K  -
tank/test/productAbeta      0  8.24G   288K  /tank/test/productAbeta
# zfs promote tank/test/productAbeta
# zfs list -r tank/test
NAME                   USED  AVAIL  REFER  MOUNTPOINT
tank/test              316K  8.24G  27.5K  /tank/test
tank/test/productA        0  8.24G   288K  /tank/test/productA
tank/test/productAbeta   288K  8.24G   288K  /tank/test/productAbeta
tank/test/productAbeta@today      0      -   288K  -
```

In the above `zfs -list` output, you can see that the space accounting of the original `productA` file system has been replaced with the `productAbeta` file system.

Complete the clone replacement process by renaming the file systems. For example:

```
# zfs rename tank/test/productA tank/test/productAlegacy
# zfs rename tank/test/productAbeta tank/test/productA
# zfs list -r tank/test
NAME                   USED  AVAIL  REFER  MOUNTPOINT
```

```
tank/test              316K  8.24G  27.5K  /tank/test
tank/test/productA     288K  8.24G   288K  /tank/test/productA
tank/test/productA@today      0      -   288K  -
tank/test/productAlegacy      0  8.24G   288K  /tank/test/productAlegacy
```

Optionally, you can remove the legacy file system. For example:

```
# zfs destroy tank/test/productAlegacy
```

Saving and Restoring ZFS Data

The `zfs send` command creates a stream representation of a snapshot that is written to standard output. By default, a full stream is generated. You can redirect the output to a file or to a different system. The `zfs receive` command creates a snapshot whose contents are specified in the stream that is provided on standard input. If a full stream is received, a new file system is created as well. You can save ZFS snapshot data and restore ZFS snapshot data and file systems with these commands. See the examples in the next section.

The following solutions for saving ZFS data are provided:

- Saving ZFS snapshots and rolling back snapshots, if necessary.
- Saving full and incremental copies of ZFS snapshots and restoring the snapshots and file systems, if necessary.
- Remotely replicating ZFS file systems by saving and restoring ZFS snapshots and file systems.
- Saving ZFS data with archive utilities such as `tar` and `cpio` or third-party backup products.

Consider the following when choosing a solution for saving ZFS data:

- 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, you could use this solution.

  For more information about creating and rolling back to a snapshot, see "ZFS Snapshots" on page 91.
- Saving snapshots – Use the `zfs send` and `zfs receive` commands to save and restore a ZFS snapshot. You can save incremental changes between snapshots, but you cannot restore files individually. You must restore the entire file system snapshot.
- Remote replication – Use the `zfs send` and `zfs receive` 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. No special configuration or hardware is required. The advantage of replicating a ZFS file system is that you can re-create 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.

Saving ZFS Data With Other Backup Products

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

For up-to-date information about issues with ZFS and third-party backup products, please see the Solaris 10 6/06 release notes.

Saving a ZFS Snapshot

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

```
# zfs send tank/dana@0830 > /bkups/dana.083006
```

You can save incremental data by using the `zfs send -i` option. For example:

```
# zfs send -i tank/dana@0829 tank/dana@0830 > /bkups/dana.today
```

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

If you need to store many copies, you might consider compressing a ZFS snapshot stream representation with the `gzip` command. For example:

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

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 a conflicting file system name exists, `zfs rename` can be used to rename the file system. For example:

```
# zfs send tank/gozer@0830 > /bkups/gozer.083006
# zfs receive tank/gozer2@today < /bkups/gozer.083006
# zfs rename tank/gozer tank/gozer.old
# zfs rename tank/gozer2 tank/gozer
```

You can use `zfs recv` as an alias for the `zfs receive` command.

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. In the following example, the previous incremental saved copy of `tank/dana` is restored.

```
# zfs rollback tank/dana@0829
cannot rollback to 'tank/dana@0829': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
```

```
tank/dana@0830
# zfs rollback -r tank/dana@0829
# zfs recv tank/dana < /bkups/dana.today
```

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

Remote Replication of ZFS Data

You can use the `zfs send` and `zfs recv` commands to remotely copy a snapshot stream representation from one system to another system. For example:

```
# zfs send tank/cindy@today | ssh newsys zfs recv sandbox/restfs@today
```

This command saves the `tank/cindy@today` snapshot data and restores it into the `sandbox/restfs` file system and also creates a `restfs@today` snapshot on the `newsys` system. In this example, the user has been configured to use `ssh` on the remote system.

◆ ◆ ◆ 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 than the standard UNIX permissions.

The following sections are provided in this chapter:

New Solaris ACL Model

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

With the introduction of NFSv4, a new ACL model fully supports the interoperability that NFSv4 offers between UNIX and non-UNIX clients. The new ACL implementation, as defined in the NFSv4 specification, provides 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 similar to NT-style ACLs.
- Provide 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.
- Provide richer inheritance semantics for designating how access privileges are applied from directory to subdirectories, and so on. For more information, see "ACL Inheritance" on page 105.

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

POSIX-draft style ACLs use a single entry to define what permissions are allowed and what permissions are denied. The new ACL model has two types of ACEs that affect access checking: `ALLOW` and `DENY`. As such, you cannot infer from any single ACE that defines a 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.
- Some NFSv4-style ACLs are translated to POSIX-draft ACLs. You see a message similar to the following if an NFSv4–style ACL isn't translated to a POSIX-draft ACL:

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

- If you create a UFS `tar` or `cpio` archive with the preserve ACL option (`tar -p` or `cpio -P`) on a system that runs a current Solaris release, you will lose the ACLs when the archive is extracted on a system that runs a previous Solaris release.

  All of the files are extracted with the correct file modes, but the ACL entries are ignored.
- You can use the `ufsrestore` command to restore data into a ZFS file system, but the ACLs will be lost.
- If you attempt to set an NFSv4-style ACL on a UFS file, you 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.
  ```

For information about other limitations with ACLs and backup products, see "Saving ZFS Data With Other Backup Products" on page 98.

Syntax Descriptions for Setting ACLs

Two basic ACL formats are provided as follows:

Syntax for Setting Trivial ACLs

`chmod [options] A[index]{+|=}owner@ |group@ |everyone@:`*access-permissions*`/...[:`*inheritance-flags*`]:deny | allow` *file*

`chmod [options] A-owner@, group@, everyone@:`*access-permissions*`/...[:`*inheritance-flags*`]:deny | allow` *file ...*

`chmod [options] A[index]-` *file*

Syntax for Setting Non-Trivial ACLs

`chmod [options] A[index]{+|=}user|group:name:`*access-permissions*/...[:*inheritance-flags*]`:deny | allow` *file*

`chmod [options] A-user|group:name:`*access-permissions*/...[:*inheritance-flags*]`:deny | allow` *file* ...

`chmod [options] A[index]-` *file*

owner@, group@, everyone@
: Identifies the *ACL-entry-type* for trivial ACL syntax. For a description of *ACL-entry-types*, see Table 7–1.

user or group:*ACL-entry-ID=username or groupname*
: Identifies the *ACL-entry-type* for explicit ACL syntax. The user and group *ACL-entry-type* must also contain 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 privileges, see Table 7–2.

inheritance-flags
: Identifies an optional list of ACL inheritance flags. For a description of the ACL inheritance flags, see Table 7–3.

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

In the following example, the *ACL-entry-ID* value is not relevant.

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

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 this example identifies the ACL entry in the larger ACL, which might have multiple entries for owner, specific UIDs, group, and everyone. You can specify 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 `A3` 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 Type	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.
`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 value is not a valid numeric UID or *username*, the ACL entry type is invalid.
`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 value is not a valid numeric GID or *groupname*, the ACL entry type is invalid.

ACL access privileges are described in the following table.

TABLE 7–2 ACL Access Privileges

Access Privilege	Compact Access Privilege	Description
add_file	w	Permission to add a new file to a directory.
add_subdirectory	p	On a directory, permission to create a subdirectory.
append_data	p	Placeholder. Not currently implemented.
delete	d	Permission to delete a file.
delete_child	D	Permission to delete a file or directory within a directory.
execute	x	Permission to execute a file or search the contents of a directory.
list_directory	r	Permission to list the contents of a directory.
read_acl	c	Permission to read the ACL (`ls`).
read_attributes	a	Permission to read basic attributes (non-ACLs) of a file. Think of basic attributes as the stat level attributes. Allowing this access mask bit means the entity can execute `ls`(1) and `stat`(2).
read_data	r	Permission to read the contents of the file.
read_xattr	R	Permission to read the extended attributes of a file or perform a lookup in the file's extended attributes directory.

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

Access Privilege	Compact Access Privilege	Description
synchronize	s	Placeholder. Not currently implemented.
write_xattr	A	Permission to create extended attributes or write to the extended attributes directory.
		Granting this permission to a user means that the user can create an extended attribute directory for a file. The attribute file's permissions control the user's access to the attribute.
write_data	w	Permission to modify or replace the contents of a file.
write_attributes	W	Permission to change the times associated with a file or directory to an arbitrary value.
write_acl	C	Permission to write the ACL or the ability to modify the ACL by using the `chmod` command.
write_owner	o	Permission to change the file's owner or group. Or, the ability to execute the `chown` or `chgrp` commands on the file.
		Permission to take ownership of a file or permission to change the group ownership of the file to a group of which the user is a member. If you want to change the file or group ownership to an arbitrary user or group, then the `PRIV_FILE_CHOWN` privilege is required.

ACL Inheritance

The purpose of using ACL inheritance is so that a newly created file or directory can 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 non-trivial ACL on a directory, it is not inherited to any subsequent directory. You must 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	Compact Inheritance Flag	Description
file_inherit	f	Only inherit the ACL from the parent directory to the directory's files.
dir_inherit	d	Only inherit the ACL from the parent directory to the directory's subdirectories.

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

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

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

ACL Property Modes

The ZFS file system includes two property modes 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 is 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 them. This mode, in effect, disables `secure` mode.

  The default mode for the `aclinherit` 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 the entries 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 for the `aclmode` property is `groupmask`.

Setting 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 these values are more of a cache of what the ACL represents. As such, if you change the permissions of the file, the file's ACL is updated accordingly. In addition, if you remove an non-trivial 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 that grant 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:

- ZFS processes ACL entries in the order they are listed in the ACL, from the top down.
- Only ACL entries that have a "who" that matches the requester of the access are processed.
- 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 the permission 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 non-trivial ACL on a directory, the ACL is not automatically inherited by the directory's children. If you set an non-trivial 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 "Setting ACL Inheritance on ZFS Files in Verbose Format" on page 115.

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

```
$ ls -v file.1
-r--r--r--   1 root     root      206663 May  4 11:52 file.1
     0:owner@:write_data/append_data/execute:deny
     1:owner@:read_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 each user category (owner@, group@, everyone@) in this example has two ACL entries. One entry for deny permissions, and one entry is for allow permissions.

A description of this file ACL is as follows:

0:owner@	The owner is denied execute permissions to the file (execute:deny).
1:owner@	The owner can read and modify the contents of the file (read_data/write_data/append_data). The owner can also modify the file's attributes such as timestamps, extended attributes, and ACLs (write_xattr/write_attributes /write_acl). In addition, the owner can modify the ownership of the file (write_owner:allow)
2:group@	The group is denied modify and execute permissions to the file (write_data/append_data/execute:deny).
3:group@	The 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 Feb 23 10:37 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@	The owner deny list is empty for the directory (::deny).
1:owner@	The owner can read and modify the directory contents (list_directory/read_data/add_file/write_data/ add_subdirectory/append_data), search the contents (execute), and modify the

file's attributes such as timestamps, extended attributes, and ACLs (`write_xattr/write_attributes/write_acl`). In addition, the owner can modify the ownership of the directory (`write_owner:allow`).

`2:group@` The group cannot add to or modify the directory contents (`add_file/write_data/add_subdirectory/append_data :deny`).

`3:group@` The group can list and read the directory contents. In addition, the group has execute permission to search 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 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.

Setting and Displaying ACLs on ZFS Files in Verbose Format

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 "Syntax Descriptions for Setting ACLs" on page 102.

- Adding ACL entries
  - Adding an ACL entry for a user

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

    ```
    % chmod Aindex-ID+acl-specification  filename
    ```

    This syntax inserts the new ACL entry at the specified *index-ID* location.
- Replacing an ACL entry

  ```
  % chmod Aindex-ID=acl-specification  filename
  ```

  ```
  % chmod A=acl-specification  filename
  ```
- Removing ACL entries

- Removing an ACL entry by *index-ID*

  ```
  % chmod Aindex-ID- filename
  ```

- Removing an ACL entry by user

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

- Removing all non-trivial ACEs from a file

  ```
  % chmod A- filename
  ```

Verbose ACL information is displayed by using the `ls -v` command. For example:

```
# ls -v file.1
-rw-r--r--   1 root     root      206663 Feb 16 11:00 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
```

For information about using the compact ACL format, see "Setting and Displaying ACLs on ZFS Files in Compact Format" on page 122.

EXAMPLE 7–1 Modifying Trivial ACLs on ZFS Files

This section provides examples of setting and displaying trivial ACLs.

In the following example, a trivial ACL exists on `file.1`:

```
# ls -v file.1
-rw-r--r--   1 root     root      206663 Feb 16 11:00 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
```

In the following example, `write_data` permissions are granted for `group@`.

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

```
# chmod A2=group@:append_data/execute:deny file.1
# chmod A3=group@:read_data/write_data:allow file.1
# ls -v file.1
-rw-rw-r--   1 root     root           206663 May  3 16:36 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@: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
```

In the following example, permissions on `file.1` are set back to 644.

```
# chmod 644 file.1
# ls -v file.1
-rw-r--r--   1 root     root           206663 May  3 16:36 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
```

EXAMPLE 7–2 Setting Non-Trivial ACLs on ZFS Files

This section provides examples of setting and displaying non-trivial ACLs.

In the following example, `read_data/execute` permissions are added for the user `gozer` on the `test.dir` directory.

```
# chmod A+user:gozer:read_data/execute:allow test.dir
# ls -dv test.dir
drwxr-xr-x+  2 root     root           2 Feb 16 11:12 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
```

EXAMPLE 7–2 Setting Non-Trivial ACLs on ZFS Files *(Continued)*

```
     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
```

In the following example, `read_data/execute` permissions are removed for user `gozer`.

```
# chmod A0- test.dir
# ls -dv test.dir
drwxr-xr-x   2 root     root           2 Feb 16 11:12 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 7–3 ACL Interaction With Permissions on ZFS Files

These ACL examples illustrate the interaction between setting ACLs and then changing the file or directory's permission bits.

In the following example, a trivial ACL exists on `file.2`:

```
# ls -v file.2
-rw-r--r--   1 root     root        2703 Feb 16 11:16 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
```

In the following example, ACL allow permissions are removed from `everyone@`.

```
# chmod A5- file.2
# ls -v file.2
```

EXAMPLE 7–3 ACL Interaction With Permissions on ZFS Files *(Continued)*

```
-rw-r-----   1 root     root        2703 Feb 16 11:16 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 this output, the file's permission bits are reset from 655 to 650. Read permissions for `everyone@` have been effectively removed from the file's permissions bits when the ACL allow permissions are removed for `everyone@`.

In the following example, the existing ACL is replaced with `read_data/write_data` permissions for `everyone@`.

```
# chmod A=everyone@:read_data/write_data:allow file.3
# ls -v file.3
-rw-rw-rw-+  1 root     root        1532 Feb 16 11:18 file.3
     0:everyone@:read_data/write_data:allow
```

In this output, the `chmod` syntax effectively replaces the existing ACL with `read_data/write_data:allow` permissions to read/write permissions for owner, group, and `everyone@`. In this model, `everyone@` specifies access to any user or group. Since no `owner@` or `group@` ACL entry exists to override the permissions for owner and group, the permission bits are set to 666.

In the following example, the existing ACL is replaced with read permissions for user `gozer`.

```
# chmod A=user:gozer:read_data:allow file.3
# ls -v file.3
----------+  1 root     root        1532 Feb 16 11:18 file.3
     0:user:gozer:read_data:allow
```

In this output, 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. The owner of the file can resolve this problem by resetting the permissions (and the ACL) as follows:

```
# chmod 655 file.3
# ls -v file.3
-rw-r-xr-x+  1 root     root           0 Mar  8 13:24 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
```

EXAMPLE 7–3 ACL Interaction With Permissions on ZFS Files *(Continued)*

```
        /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 7–4 Restoring Trivial ACLs on ZFS Files

You can use the `chmod` command to remove all non-trivial ACLs on a file or directory.

In the following example, 2 non-trivial ACEs exist on `test5.dir`.

```
# ls -dv test5.dir
drwxr-xr-x+  2 root     root           2 Feb 16 11:23 test5.dir
     0:user:gozer:read_data:file_inherit:deny
     1:user:lp:read_data:file_inherit:deny
     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
```

In the following example, the non-trivial ACLs for users `gozer` and `lp` are removed. The remaining ACL contains the six default values for `owner@`, `group@`, and `everyone@`.

```
# chmod A- test5.dir
# ls -dv test5.dir
drwxr-xr-x   2 root     root           2 Feb 16 11:23 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: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
```

Setting ACL Inheritance on ZFS Files in Verbose Format

You can determine how ACLs are inherited or not inherited on files and directories. By default, ACLs are not propagated. If you set an non-trivial ACL on a directory, the ACL is not inherited by any subsequent directory. You must specify the inheritance of an ACL on a file or directory.

In addition, two ACL properties are provided that can be set globally on file systems: `aclinherit` and `aclmode`. By default, `aclinherit` is set to `secure` and `aclmode` is set to `groupmask`.

For more information, see "ACL Inheritance" on page 105.

EXAMPLE 7–5 Default ACL Inheritance

By default, ACLs are not propagated through a directory structure.

In the following example, an non-trivial ACE 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 Feb 17 14:45 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 `everyone@`.

```
# mkdir test.dir/sub.dir
# ls -dv test.dir/sub.dir
drwxr-xr-x   2 root     root           2 Feb 17 14:46 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
```

EXAMPLE 7–5 Default ACL Inheritance *(Continued)*

```
     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 7–6 Granting ACL Inheritance on Files and Directories

This series of examples identify the file and directory ACEs that are applied when the `file_inherit` flag is set.

In the following example, `read_data/write_data` permissions are added for files in the `test.dir` directory for user `gozer` so that he has read access on any newly created files.

```
# chmod A+user:gozer:read_data/write_data:file_inherit:allow test2.dir
# ls -dv test2.dir
drwxr-xr-x+  2 root     root           2 Feb 17 14:47 test2.dir
     0:user:gozer:read_data/write_data:file_inherit: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
```

In the following example, user gozer's permissions are applied on the newly created `test2.dir/file.2` file. The ACL inheritance granted, `read_data:file_inherit:allow`, 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 Feb 17 14:49 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
```

EXAMPLE 7–6 Granting ACL Inheritance on Files and Directories *(Continued)*

```
     7:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

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, is used to propagate the ACL through the directory structure. As such, user `gozer` is only granted or denied permission from `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 Feb 17 14:50 test2.dir/subdir.2
     0:user:gozer:list_directory/read_data/add_file/write_data:file_inherit
         /inherit_only: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
```

The following series of examples identify the file and directory ACLs that are 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:file_inherit/dir_inherit:allow test3.dir
# ls -dv test3.dir
drwxr-xr-x+  2 root     root           2 Feb 17 14:51 test3.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute
         :file_inherit/dir_inherit: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
```

EXAMPLE 7–6 Granting ACL Inheritance on Files and Directories *(Continued)*

```
    /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 Feb 17 14:53 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

          # mkdir test3.dir/subdir.1
          # ls -dv test3.dir/subdir.1
          drwxr-xr-x+  2 root     root           2 May  4 15:00 test3.dir/subdir.1
               0:user:gozer:list_directory/read_data/add_file/write_data/execute
                   :file_inherit/dir_inherit/inherit_only:allow
               1:user:gozer:add_file/write_data:deny
               2:user:gozer:list_directory/read_data/add_file/write_data/execute:allow
               3:owner@::deny
               4:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
                   /append_data/write_xattr/execute/write_attributes/write_acl
                   /write_owner:allow
               5:group@:add_file/write_data/add_subdirectory/append_data:deny
               6:group@:list_directory/read_data/execute:allow
               7:everyone@:add_file/write_data/add_subdirectory/append_data/write_xattr
                   /write_attributes/write_acl/write_owner:deny
               8:everyone@:list_directory/read_data/read_xattr/execute/read_attributes
                   /read_acl/synchronize:allow
```

In these examples, because the permission bits of the parent directory for `group@` and `everyone@` deny write and execute permissions, user `gozer` is denied write and execute permissions. The default `aclmode` property is `secure`, which means that `write_data` and `execute` 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.

EXAMPLE 7–6 Granting ACL Inheritance on Files and Directories *(Continued)*

```
# chmod A+user:gozer:read_data/write_data/execute:file_inherit/no_propagate:allow test4.dir
# ls -dv test4.dir
drwxr-xr-x+  2 root     root           2 Feb 17 14:54 test4.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute
         :file_inherit/no_propagate: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
```

As the following example illustrates, 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 Feb 17 14:57 test4.dir/sub4.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
```

As the following example illustrates, `gozer`'s `read_data/write_data/execute` permission for files is propagated to the newly created file.

```
# touch test4.dir/file.4
# ls -v test4.dir/file.4
-rw-r--r--+  1 root     root           0 May  4 15:02 test4.dir/file.4
     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
```

EXAMPLE 7–6 Granting ACL Inheritance on Files and Directories *(Continued)*

```
     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
```

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

If the `aclmode` property on the `tank/cindy` file system is set to `passthrough`, then user `gozer` would inherit the ACL applied on `test4.dir` 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 Feb 17 15:15 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
         /write_acl/write_owner:deny
     6:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

This 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 7–8 ACL Inheritance With ACL Mode Set to Discard

If the `aclmode` property on a file system is set to `discard`, then ACLs can potentially be 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:dir_inherit:allow test5.dir
# ls -dv test5.dir
drwxr-xr-x+  2 root     root           2 Feb 16 11:23 test5.dir
     0:user:gozer:list_directory/read_data/add_file/write_data/execute
         :dir_inherit:allow
     1:owner@::deny
     2:owner@:list_directory/read_data/add_file/write_data/add_subdirectory
```

EXAMPLE 7–8 ACL Inheritance With ACL Mode Set to Discard *(Continued)*

```
         /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 non-trivial ACL is discarded. For example:

```
# chmod 744 test5.dir
# ls -dv test5.dir
drwxr--r--   2 root     root           2 Feb 16 11:23 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 7–9 ACL Inheritance With ACL Inherit Mode Set to Noallow

In the following example, two non-trivial ACLs with file inheritance are set. One ACL allows `read_data` permission, and one ACL denies `read_data` permission. This example also illustrates how you can specify two ACEs in the same `chmod` command.

```
# zfs set aclinherit=nonallow tank/cindy
# chmod A+user:gozer:read_data:file_inherit:deny,user:lp:read_data:file_inherit:allow test6.dir
# ls -dv test6.dir
drwxr-xr-x+  2 root     root           2 May  4 14:23 test6.dir
     0:user:gozer:read_data:file_inherit:deny
     1:user:lp:read_data:file_inherit: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
```

EXAMPLE 7–9 ACL Inheritance With ACL Inherit Mode Set to Noallow *(Continued)*

```
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
```

As the following example shows, 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 May  4 13:44 test6.dir/file.6
     0:user:gozer:read_data:deny
     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
         /write_acl/write_owner:deny
     6:everyone@:read_data/read_xattr/read_attributes/read_acl/synchronize
         :allow
```

Setting and Displaying ACLs on ZFS Files in Compact Format

You can set and display permissions on ZFS files in a compact format that uses 14 unique letters to represent the permissions. The letters that represent the compact permissions are listed in Table 7–2 and Table 7–3.

You can display compact ACL listings for files and directories by using the `ls -V` command. For example:

```
# ls -V file.1
-rw-r--r--   1 root     root      206663 Feb 16 11:00 file.1
           owner@:--x-----------:------:deny
           owner@:rw-p---A-W-Co-:------:allow
           group@:-wxp----------:------:deny
           group@:r-------------:------:allow
         everyone@:-wxp---A-W-Co-:------:deny
         everyone@:r-----a-R-c--s:------:allow
```

The compact ACL output is described as follows:

`owner@` The owner is denied execute permissions to the file (`x=execute`).

`owner@`	The owner can read and modify the contents of the file (`rw=read_data/write_data`), (`p=append_data`). The owner can also modify the file's attributes such as timestamps, extended attributes, and ACLs (`A=write_xattr`, `W=write_attributes`, `C=write_acl`). In addition, the owner can modify the ownership of the file (`O=write_owner`).
`group@`	The group is denied modify and execute permissions to the file (`rw=read_data/write_data`, `p=append_data`, and `x=execute`).
`group@`	The group is granted read permissions to the file (`r=read_data`).
`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 (`w=write_data`, `x=execute`, `p=append_data`, `A=write_xattr`, `W=write_attributes`, `C=write_acl`, and `o=write_owner`).
`everyone@`	Everyone who is not user or group is granted read permissions to the file and the file's attributes (`r=read_data`, `a=append_data`, `R=read_xattr`, `c=read_acl`, and `s=synchronize`). The `synchronize` access permission is not currently implemented.

Compact ACL format provides the following advantages over verbose ACL format:

- Permissions can be specified as positional arguments to the `chmod` command.
- The hyphen (-) characters, which identify no permissions, can be removed and only the required letters need to be specified.
- Both permissions and inheritance flags are set in the same fashion.

For information about using the verbose ACL format, see "Setting and Displaying ACLs on ZFS Files in Verbose Format" on page 109.

EXAMPLE 7–10 Setting and Displaying ACLs in Compact Format

In the following example, a trivial ACL exists on `file.1`:

```
# ls -V file.1
-rw-r-xr-x   1 root     root      206663 Feb 16 11:00 file.1
           owner@:--x-----------:------:deny
           owner@:rw-p---A-W-Co-:------:allow
           group@:-w-p----------:------:deny
           group@:r-x-----------:------:allow
        everyone@:-w-p---A-W-Co-:------:deny
        everyone@:r-x---a-R-c--s:------:allow
```

In this example, read_data/execute permissions are added for the user `gozer` on `file.1`.

```
# chmod A+user:gozer:rx:allow file.1
# ls -V file.1
-rw-r-xr-x+  1 root     root      206663 Feb 16 11:00 file.1
```

EXAMPLE 7–10 Setting and Displaying ACLs in Compact Format *(Continued)*

```
        user:gozer:r-x-----------:------:allow
            owner@:--x-----------:------:deny
            owner@:rw-p---A-W-Co-:------:allow
            group@:-w-p----------:------:deny
            group@:r-x-----------:------:allow
         everyone@:-w-p---A-W-Co-:------:deny
         everyone@:r-x---a-R-c--s:------:allow
```

Another way to add the same permissions for user `gozer` is to insert a new ACL at a specific position, 4, for example. As such, the existing ACLs at positions 4–6 are pushed down. For example:

```
# chmod A4+user:gozer:rx:allow file.1
# ls -V file.1
-rw-r-xr-x+  1 root     root      206663 Feb 16 11:00 file.1
            owner@:--x-----------:------:deny
            owner@:rw-p---A-W-Co-:------:allow
            group@:-w-p----------:------:deny
            group@:r-x-----------:------:allow
        user:gozer:r-x-----------:------:allow
         everyone@:-w-p---A-W-Co-:------:deny
         everyone@:r-x---a-R-c--s:------:allow
```

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

```
# chmod A+user:gozer:rwx:fd:allow dir.2
# ls -dV dir.2
drwxr-xr-x+  2 root     root           2 Aug 28 13:21 dir.2
        user:gozer:rwx-----------:fd----:allow
            owner@:--------------:------:deny
            owner@:rwxp---A-W-Co-:------:allow
            group@:-w-p----------:------:deny
            group@:r-x-----------:------:allow
         everyone@:-w-p---A-W-Co-:------:deny
         everyone@:r-x---a-R-c--s:------:allow
```

You can also cut and paste permissions and inheritance flags from the `ls -V` output into the compact `chmod` format. For example, to duplicate the permissions and inheritance flags on `dir.1` for user `gozer` to user `cindys`, copy and paste the permission and inheritance flags (`rwx-----------:f-----:allow`) into your `chmod` command. For example:

```
# chmod A+user:cindys:rwx-----------:fd----:allow dir.2
# ls -dv dir.2
drwxr-xr-x+  2 root     root           2 Aug 28 14:12 dir.2
       user:cindys:rwx-----------:fd----:allow
```

EXAMPLE 7–10 Setting and Displaying ACLs in Compact Format *(Continued)*

```
        user:gozer:rwx-----------:fd----:allow
            owner@:--------------:------:deny
            owner@:rwxp---A-W-Co-:------:allow
            group@:-w-p----------:------:deny
            group@:r-x-----------:------:allow
         everyone@:-w-p---A-W-Co-:------:deny
         everyone@:r-x---a-R-c--s:------:allow
```

◆ ◆ ◆ CHAPTER 8

ZFS Advanced Topics

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

The following sections are provided in this chapter:

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.

In the following example, 5-Gbyte ZFS volume, `tank/vol`, is created:

```
# 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 the volume shrinks, data corruption might occur. You must be careful when changing the size of the volume.

In addition, if you create a snapshot of a volume that changes in size, you might introduce file system inconsistencies if you attempt to rollback the snapshot or create a clone from the snapshot.

For information about file system properties that can be applied to volumes, see Table 5–1.

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 will fail. For information about using ZFS volumes in a global zone, see "Adding ZFS Volumes to a Non-Global Zone" on page 130.

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.

In the following example, the 5-Gbyte `tank/vol` volume is added 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 not supported. Use the `dumpadm` command to set up a dump device.

Using ZFS on a Solaris System With Zones Installed

ZFS datasets can be added to a zone either as a generic file system or as a delegated dataset.

Adding a file system allows the non-global zone to share space with the global zone, though the zone administrator cannot control properties or create new file systems in the underlying file system hierarchy. This is identical to adding any other type of file system 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 file systems within that dataset, and modify properties of the datasets. The zone administrator cannot affect datasets that have not been added to the zone, and cannot exceed any top-level quotas set on the exported dataset.

Consider the following interactions when working with ZFS on a system with Solaris zones installed:

- A ZFS file system that is added to a non-global zone must have its `mountpoint` property set to legacy.
- A ZFS file system cannot serve as zone root because of issues with the Solaris upgrade process. Do not include any system-related software that is accessed by the patch or upgrade process in a ZFS file system that is delegated to a non-global zone.

Adding ZFS File Systems to a Non-Global Zone

You can add a ZFS file system as a generic file system when the goal is solely to share space with the global zone. A ZFS file system that is added to a non-global zone must have its `mountpoint` property set to legacy.

You can add a ZFS file system to a non-global zone by using the `zonecfg` command's `add fs` subcommand. For example:

In the following example, a ZFS file system is added to a non-global zone by a 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 file system, `tank/zone/zion`, to the zone `zion`, mounted at `/export/shared`. The `mountpoint` property of the file system must be set to `legacy`, and the file system cannot already be mounted in another location. The zone administrator can create and destroy files within the file system. The file system cannot be remounted in a different location, nor can the zone administrator change properties on the file system such as atime, readonly, compression, and so on. The global zone administrator is responsible for setting and controlling properties of the file system.

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

Delegating Datasets to a Non-Global Zone

If the primary goal is to delegate the administration of storage to a zone, then ZFS supports adding datasets to a non-global zone through use of the `zonecfg` command's `add dataset` subcommand.

In the following example, a ZFS file system is delegated to a non-global zone by a 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 file system, this syntax causes the ZFS file system `tank/zone/zion` to be visible within the zone `zion`. The zone administrator can set file system properties, as well as create children. In addition, the zone administrator can take snapshots, create clones, and otherwise control the entire file system hierarchy.

For more information about what actions are allowed within zones, see "Property Management Within a Zone" on page 130.

Adding ZFS Volumes to a Non-Global Zone

Emulated volumes cannot be added to a non-global zone by using the `zonecfg` command's `add dataset` subcommand. If an attempt to add an emulated volume is detected, the zone cannot boot. However, volumes can be added to a zone by using the `zonecfg` command's `add device` subcommand.

In the following example, a ZFS emulated volume is added to a non-global zone by a 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 file systems that would panic the system when a mount is attempted. For more information about adding devices to zones and the related security risks, see "Understanding the zoned Property" on page 131.

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

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 creating, adding, or removing devices, is not allowed from within a zone. Even if physical devices are added to a zone by using the `zonecfg` command's `add device` subcommand, or if files are used, the `zpool` command does not allow the creation of any new pools within the zone.

Property Management Within a Zone

Once a dataset is added to a zone, the zone administrator can control specific dataset properties. 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, consider the following configuration:

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

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

If `tank/data/zion` is added 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 datasets that are 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 zone administrator change the `zoned` property, because doing so 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 behavior 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.

Understanding the `zoned` Property

When a dataset is added to a non-global zone, the dataset 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 file system, there might be setuid binaries, symbolic links, or otherwise questionable contents that might adversely affect the security of the global zone. In addition, the `mountpoint` property cannot be interpreted in the context of the global zone. Otherwise, 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. A zone administrator will not need to manually turn on this property. 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 the dataset 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 behavior is due to the inherent security risks associated with these tasks. Because an untrusted user has had complete access to the dataset and its children, the mountpoint property might be set to bad values, or setuid binaries might exist on the file systems.

To prevent accidental security risks, the zoned property must be manually cleared by the global administrator if you want to reuse the dataset in any way. Before setting the zoned property to off, 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 off the setuid property.

Once you have verified that no security vulnerabilities are left, the zoned property can be turned off by using 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.

ZFS Alternate Root Pools

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

Creating ZFS Alternate Root Pools

The most common use for creating an alternate root pool is for use with removable media. In these circumstances, users typically want a single file system, and they want it to be mounted wherever they

choose on the target system. When an alternate root pool is created by using the `-R` option, the mount point of the root file system is automatically set to `/`, which is the equivalent of the alternate root itself.

In the following example, a pool called `morpheus` is created with `/mnt` as the alternate root path:

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

Note the single file system, `morpheus`, whose mount point is the alternate root of the pool, `/mnt`. The mount point that is stored on disk is `/` and the full path to `/mnt` is interpreted only in the context of the alternate root pool. This file system can then be exported and imported under an arbitrary alternate root pool on a different system.

Importing Alternate Root Pools

Pools can also be imported using an alternate root. This feature 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 feature also can be used when mounting removable media as described above.

In the following example, a pool called `morpheus` is imported with `/mnt` as the alternate root path. This example assumes that `morpheus` was previously exported.

```
# 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
```

ZFS Rights Profiles

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

- ZFS Storage Management – Provides the ability to create, destroy, and manipulate devices within a ZFS storage pool
- ZFS File system Management – Provides the ability to create, destroy, and modify ZFS file systems

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

◆ ◆ ◆ CHAPTER 9

ZFS Troubleshooting and Data Recovery

This chapter describes how to identify and recover from ZFS failure modes. Information for preventing failures is provided as well.

The following sections are provided in this chapter:

ZFS Failure Modes

As a combined file system and volume manager, ZFS can exhibit many different failure modes. This chapter begins by outlining the various failure modes, then discusses how to identify them on a running system. This chapter concludes by discussing how to repair the problems. ZFS can encounter three basic types of errors:

- Missing devices
- Damaged devices
- Corrupted data

Note that a single pool can experience all three errors, so a complete repair procedure involves finding and correcting one error, proceeding to the next error, and so on.

Missing Devices in a ZFS Storage Pool

If a device is completely removed from the system, ZFS detects that the device cannot be opened and places it in the FAULTED state. Depending on the data replication level of the pool, this might or might not result in the entire pool becoming unavailable. If one disk in a mirrored or RAID-Z device is removed, the pool continues to be accessible. If all components of a mirror are removed, if more than one device in a RAID-Z device is removed, or if a single-disk, top-level device is removed, the pool becomes FAULTED. No data is accessible until the device is reattached.

Damaged Devices in a ZFS Storage Pool

The term "damaged" covers a wide variety of possible errors. Examples include the following errors:

- 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 or from the wrong location
- 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 is having problems. In other cases, the damage is permanent, such as on-disk corruption. Even still, whether the damage is permanent does not necessarily indicate that the error is likely to occur again. For example, if an administrator accidentally overwrites part of a disk, no type of hardware failure has occurred, and the device need 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.

Corrupted ZFS Data

Data corruption occurs when one or more device errors (indicating 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, corrupted data will be the result.

Data corruption is always permanent and requires special consideration during repair. Even if the underlying devices are repaired or replaced, the original data is lost forever. Most often this scenario requires restoring data from backups. Data errors are recorded as they are encountered, and can be controlled through regular disk scrubbing as explained in the following section. When a corrupted block is removed, the next scrubbing pass recognizes that the corruption is no longer present and removes any trace of the error from the system.

Checking ZFS Data Integrity

No `fsck` utility equivalent exists for ZFS. This utility has traditionally served two purposes, data repair and data validation.

Data Repair

With traditional file systems, the way in which data is written is inherently vulnerable to unexpected failure causing data inconsistencies. Because a traditional file system is not transactional, unreferenced blocks, bad link counts, or other inconsistent data structures are possible. The addition of journaling does solve some of these problems, but can introduce additional problems when the log cannot be rolled back. With ZFS, none of these problems exist. The only way for inconsistent data to exist on disk is through hardware failure (in which case the pool should have been replicated) or a bug in the ZFS software exists.

Given that the `fsck` utility is designed to repair known pathologies specific to individual file systems, writing such a utility for a file system with no known pathologies is impossible. Future experience might 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, the chance that data corruption can render some or all of your data inaccessible is always present.

Data Validation

In addition to data repair, the `fsck` utility validates that the data on disk has no problems. Traditionally, this task is done by unmounting the file system and running the `fsck` utility, possibly taking the system to single-user mode in the process. This scenario results in downtime that is proportional to the size of the file system being checked. Instead of requiring an explicit utility to perform the necessary checking, ZFS provides a mechanism to perform regular checking of all data. This functionality, known as *scrubbing*, is commonly used in memory and other systems as a method of detecting and preventing errors before they result in hardware or software failure.

Controlling ZFS Data Scrubbing

Whenever ZFS encounters an error, either through scrubbing or when accessing a file on demand, the error is logged internally so that you can get a quick overview of all known errors within the pool.

Explicit ZFS Data Scrubbing

The simplest way to check your data integrity is to initiate an explicit scrubbing of all data within the pool. This operation traverses all the data in the pool once and verifies that all blocks can be read. Scrubbing proceeds as fast as the devices allow, though the priority of any I/O remains below that of

normal operations. This operation might negatively impact performance, though the file system should remain usable and nearly as responsive while the scrubbing occurs. To initiate an explicit scrub, use the `zpool scrub` command. For example:

```
# zpool scrub tank
```

The status of the current scrub can be displayed in the `zpool status` output. For example:

```
# zpool status -v tank
  pool: tank
 state: ONLINE
 scrub: scrub completed with 0 errors on Wed Aug 30 14:02:24 2006
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

errors: No known data errors
```

Note that only one active scrubbing operation per pool can occur at one time.

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 issue can safely be ignored.

For more information about interpreting `zpool status` output, see "Querying ZFS Storage Pool Status" on page 50.

ZFS Data Scrubbing and Resilvering

When a device is replaced, a resilvering operation is initiated to move data from the good copies to the new device. This action is a form of disk scrubbing. Therefore, only one such action can happen at a given time in the pool. If a scrubbing operation is in progress, a resilvering operation suspends the current scrubbing, and restarts it after the resilvering is complete.

For more information about resilvering, see "Viewing Resilvering Status" on page 148.

Identifying Problems in ZFS

All ZFS troubleshooting is centered around the `zpool status` command. This command analyzes the various failures in the system and identifies the most severe problem, presenting you with a suggested action and a link to a knowledge article for more information. 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 does not fix the data corruption problems.

In addition, a ZFS diagnostic engine is provided to diagnose and report pool failures and device failures. Checksum, I/O, device, and pool errors associated with pool or device failures are also reported. ZFS failures as reported by `fmd` are displayed on the console as well as the system messages file. In most cases, the `fmd` message directs you to the `zpool status` command for further recovery instructions.

The basic recovery process is as follows:

- Identify the errors through the `fmd` messages that are displayed on the system console or in the `/var/adm/messages` files.
- Find further repair instructions in the `zpool status -x` command.
- Repair the failures, such as:
  - Replace the faulted or missing device and bring it online.
  - Restore the faulted configuration or corrupted data from a backup.
  - Verify the recovery by using the `zpool status -x` command.
  - Back up your restored configuration, if applicable.

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

Determining if Problems Exist in a ZFS Storage Pool

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

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

Without the `-x` flag, the command displays the 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 about command-line options to the `zpool status` command, see "Querying ZFS Storage Pool Status" on page 50.

Understanding `zpool status` Output

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

```
# zpool status tank
  pool: tank
 state: DEGRADED
status: One or more devices has been taken offline by the administrator.
        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
            c1t0d0   ONLINE       0     0     0
            c1t1d0   OFFLINE      0     0     0

errors: No known data errors
```

This output is divided into several sections:

Overall Pool Status Information

This header section in the `zpool status` output 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 information refers only to the ability of the pool to provide the necessary replication level. Pools that are `ONLINE` might still have failing devices or data corruption.

`status` A description of what is wrong with the pool. This field is omitted if no problems are found.

`action` A recommended action for repairing the errors. This field 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. Online articles are updated more often than this guide can be updated, and should always be referenced for the most up-to-date repair procedures. This field is omitted if no problems are found.

`scrub` Identifies the current status of a scrub operation, which might include the date and time that the last scrub was completed, a scrub in progress, or if no scrubbing was requested.

`errors` Identifies known data errors or the absence of known data errors.

Configuration Information

The `config` field in the `zpool status` output describes the configuration layout of the devices comprising the pool, as well as their state and any errors generated from the devices. The state can be one of the following: `ONLINE`, `FAULTED`, `DEGRADED`, `UNAVAILABLE`, or `OFFLINE`. If the state is anything but `ONLINE`, 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 occurred while issuing a read request.
- `WRITE` – I/O error occurred 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 might indicate a temporary outage, while a large number might indicate a permanent problem with the device. These errors do not necessarily correspond to data corruption as interpreted by applications. If the device is in a redundant configuration, the disk devices might show uncorrectable errors, while no errors appear at the mirror or RAID-Z device level. If this scenario is the case, then ZFS successfully retrieved the good data and attempted to heal the damaged data from existing replicas.

For more information about interpreting these errors to determine device failure, see "Determining the Type of Device Failure" on page 145.

Finally, additional auxiliary information is displayed in the last column of the `zpool status` output. This information expands on the `state` field, aiding in diagnosis of failure modes. If a device is `FAULTED`, this field indicates whether the device is inaccessible or whether the data on the device is corrupted. If the device is undergoing resilvering, this field displays the current progress.

For more information about monitoring resilvering progress, see "Viewing Resilvering Status" on page 148.

Scrubbing Status

The third section of the `zpool status` output describes the current status of any explicit scrubs. This information is distinct from whether any errors are detected on the system, though this information can be used to determine the accuracy of the data corruption error reporting. If the last scrub ended recently, most likely, any known data corruption has been discovered.

For more information about data scrubbing and how to interpret this information, see "Checking ZFS Data Integrity" on page 137.

Data Corruption Errors

The `zpool status` command also shows whether any known errors are associated with the pool. These errors might have been found during disk scrubbing or during normal operation. ZFS maintains a persistent log of all data errors associated with the pool. This log is rotated whenever a complete scrub of the system finishes.

Data corruption 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 by using the `zpool status -v` option. For example:

```
# zpool status -v
  pool: tank
 state: DEGRADED
status: One or more devices has experienced an error resulting in data
        corruption.  Applications may be affected.
action: Restore the file in question if possible.  Otherwise restore the
        entire pool from backup.
   see: http://www.sun.com/msg/ZFS-8000-8A
 scrub: resilver completed with 1 errors on Fri Mar 17 15:42:18 2006
config:

        NAME         STATE     READ WRITE CKSUM
        tank         DEGRADED     0     0     1
          mirror     DEGRADED     0     0     1
            c1t0d0   ONLINE       0     0     2
            c1t1d0   UNAVAIL      0     0     0  corrupted data

errors: The following persistent errors have been detected:

          DATASET  OBJECT  RANGE
          5        0       lvl=4294967295 blkid=0
```

A similar message is also displayed by `fmd` on the system console and the `/var/adm/messages` file. These messages can also be tracked by using the `fmdump` command.

For more information about interpreting data corruption errors, see "Identifying the Type of Data Corruption" on page 150.

System Reporting of ZFS Error Messages

In addition to persistently keeping track of errors within 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 might 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 detected, ZFS logs a message describing when and where the corruption was detected. This message is only logged the first time it is detected. Subsequent accesses do not generate a message.
- **Pool failures and device failures** – If a pool failure or device failure occurs, the fault manager daemon reports these errors through syslog messages as well as the `fmdump` command.

If ZFS detects a device error and automatically recovers from it, no notification occurs. Such errors do not constitute a failure in the pool redundancy or data integrity. Moreover, such errors are typically the result of a driver problem accompanied by its own set of error messages.

Repairing a Damaged ZFS Configuration

ZFS maintains a cache of active pools and their configuration on the root file system. 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 tries to avoid this situation, though arbitrary corruption is always possible given the qualities of the underlying file system and storage. This situation typically results in a pool disappearing from the system when it should otherwise be available. This situation can also manifest itself as a partial configuration that is missing an unknown number of top-level virtual devices. In either case, the configuration can be recovered by exporting the pool (if it is visible at all), and re-importing it.

For more information about importing and exporting pools, see "Migrating ZFS Storage Pools" on page 56.

Repairing a Missing Device

If a device cannot be opened, it displays as `UNAVAILABLE` in the `zpool status` output. This status 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 might be compromised. In either case, the device simply needs to be reattached to the system to restore normal operation.

For example, you might see a message similar to the following from `fmd` after a device failure:

```
SUNW-MSG-ID: ZFS-8000-D3, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Thu Aug 31 11:40:59 MDT 2006
PLATFORM: SUNW,Sun-Blade-1000, CSN: -, HOSTNAME: tank
SOURCE: zfs-diagnosis, REV: 1.0
```

```
EVENT-ID: e11d8245-d76a-e152-80c6-e63763ed7e4e
DESC: A ZFS device failed.  Refer to http://sun.com/msg/ZFS-8000-D3 for more information.
AUTO-RESPONSE: No automated response will occur.
IMPACT: Fault tolerance of the pool may be compromised.
REC-ACTION: Run 'zpool status -x' and replace the bad device.
```

The next step is to use the `zpool status -x` command to view more detailed information about the device problem and the resolution. For example:

```
# zpool status -x
  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-D3
 scrub: resilver completed with 0 errors on Thu Aug 31 11:45:59 MDT 2006
config:

        NAME         STATE     READ WRITE CKSUM
        tank         DEGRADED     0     0     0
          mirror     DEGRADED     0     0     0
            c0t1d0   UNAVAIL      0     0     0  cannot open
            c1t1d0   ONLINE       0     0     0
```

You can see from this output that the missing device `c0t1d0` is not functioning. If you determine that the drive is faulty, replace the device.

Then, use the `zpool online` command to online the replaced device. For example:

```
# zpool online tank c0t1d0
```

Confirm that the pool with the replaced device is healthy.

```
# zpool status -x tank
pool 'tank' is healthy
```

Physically Reattaching the Device

Exactly how a missing device is reattached depends 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 might have failed such that the device is no longer visible to the system. In this case, the controller should be replaced at which point the disks will again be available. Other pathologies can exist and depend on the type of

hardware and its configuration. If a drive fails and it is no longer visible to the system (an unlikely event), the device should be treated as a damaged device. Follow the procedures outlined in "Repairing a Damaged Device" on page 145.

Notifying ZFS of Device Availability

Once a device is reattached to the system, ZFS might or might not automatically detect its availability. If the pool was previously faulted, or the system was rebooted as part of the attach procedure, then ZFS 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, you must notify ZFS that the device is now available and ready to be reopened by using the `zpool online` command. For example:

```
# zpool online tank c0t1d0
```

For more information about bringing devices online, see "Bringing a Device Online" on page 45.

Repairing a Damaged Device

This section describes how to determine device failure types, clear transient errors, and replace a device.

Determining the Type of Device 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 might indicate a faulty drive.
- **Administrator error** – Administrators can unknowingly overwrite portions of the disk with bad data (such as copying `/dev/zero` over portions of the disk) that cause permanent corruption on disk. These errors are always transient.
- **Temporary outage**– A disk might become unavailable for a period time, causing I/Os to fail. This situation is typically associated with network-attached devices, though local disks can experience temporary outages as well. These errors might or might not be transient.
- **Bad or flaky hardware** – This situation 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 a device is offline, it is assumed that the administrator placed the device in this state because it is presumed faulty. The administrator who placed the device in this state can determine is this assumption is accurate.

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

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

The errors are divided into I/O errors and checksum errors, both of which might 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 this situation probably indicates impending or complete device failure. However, the pathology for administrator error can result in large error counts. The other source of information is the system log. If the log shows a large number of SCSI or fibre channel driver messages, then this situation probably indicates serious hardware problems. If no syslog messages are generated, then the damage is likely transient.

The goal is to answer the following question:

Is another error likely to occur on this device?

Errors that happen only once are considered *transient*, and do not indicate potential failure. Errors that 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 must be done manually by you, 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.

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 "Repairing Damaged Data" on page 150.

Clearing Transient Errors

If the device errors 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 no fatal error occurred. To clear error counters for RAID-Z or mirrored devices, use the `zpool clear` command. For example:

```
# zpool clear tank c1t0d0
```

This syntax clears any errors associated with the device and clears any data error counts associated with the device.

To clear all errors associated with the virtual devices in the pool, and clear any data error counts associated with the pool, use the following syntax:

```
# zpool clear tank
```

For more information about clearing pool errors, see "Clearing Storage Pool Devices" on page 46.

Replacing a Device in a ZFS Storage Pool

If device damage is permanent or future permanent damage is likely, the device must be replaced. Whether the device can be replaced depends on the configuration.

Determining if a Device Can Be Replaced

For a device to be replaced, the pool must be in the `ONLINE` state. 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, sufficient replicas from which to retrieve good data must exist. If two disks in a four-way mirror are faulted, then either disk can be replaced because healthy replicas are available. However, if two disks in a four-way RAID-Z device are faulted, then neither disk can be replaced because not enough replicas from which to retrieve data exist. If the device is damaged but otherwise online, it can be replaced as long as the pool is not in the `FAULTED` state. However, any bad data on the device is copied to the new device unless there are sufficient replicas with good data.

In the following configuration, the disk `c1t1d0` can be replaced, and any data in the pool is copied from the good replica, `c1t0d0`.

```
mirror            DEGRADED
    c1t0d0             ONLINE
    c1t1d0             FAULTED
```

The disk `c1t0d0` can also be replaced, though no self-healing of data can take place because no good replica is available.

In the following configuration, neither of the faulted disks can be replaced. The `ONLINE` disks cannot be replaced either, because the pool itself is faulted.

```
raidz             FAULTED
    c1t0d0             ONLINE
    c2t0d0             FAULTED
    c3t0d0             FAULTED
    c3t0d0             ONLINE
```

In the following configuration, either top-level disk can be replaced, though any bad data present on the disk is copied to the new disk.

```
c1t0d0          ONLINE
c1t1d0          ONLINE
```

If either disk were faulted, then no replacement could be performed because the pool itself would be faulted.

Unreplaceable Devices

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

For more information about restoring an entire pool, see "Repairing ZFS Storage Pool-Wide Damage" on page 152.

Replacing a Device

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

```
# zpool replace tank c1t0d0 c2t0d0
```

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 the command is finished, it detaches the damaged device from the configuration, at which point the device 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. For example:

```
# zpool replace tank c1t0d0
```

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

For more information about the `zpool replace` command, see "Replacing Devices in a Storage Pool" on page 46.

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 device is known as *resilvering*, and can be monitored by using the `zpool status` command.

Traditional file systems resilver data at the block level. Because ZFS eliminates the artificial layering of the volume manager, it can perform resilvering in a much more powerful and controlled manner. The two main advantages of this feature are as follows:

- ZFS only resilvers the minimum amount of necessary data. 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 resilvering the entire disk, or complicating matters with "dirty region" logging that some volume managers support. When an entire disk is replaced, the resilvering process takes time proportional to the amount of data used on disk. Replacing a 500-Gbyte disk can take seconds if only a few gigabytes of used space is in the pool.
- Resilvering is interruptible and safe. If the system loses power or is rebooted, the resilvering process resumes exactly where it left off, without any need for manual intervention.

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

```
# 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
 scrub: none requested
config:
        NAME                  STATE     READ WRITE CKSUM
        tank                  DEGRADED     0     0     0
          mirror              DEGRADED     0     0     0
            replacing         DEGRADED     0     0     0  52% resilvered
              c1t0d0          ONLINE       0     0     0
              c2t0d0          ONLINE       0     0     0
            c1t1d0            ONLINE       0     0     0
```

In this example, the disk `c1t0d0` is being replaced by `c2t0d0`. This event is observed in the status output by presence of the *replacing* virtual device in the configuration. This device is not real, nor is it possible for you to create a pool by using this virtual device type. The purpose of this device 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 cannot provide the desired replication level until the resilvering process is complete. Resilvering proceeds as fast as possible, though the I/O is always scheduled with a 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, configuration. For example:

```
# zpool status tank
  pool: tank
 state: ONLINE
 scrub: scrub completed with 0 errors on Thu Aug 31 11:20:18 2006
config:

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

```
            c2t0d0  ONLINE       0     0     0
            c1t1d0  ONLINE       0     0     0

errors: No known data errors
```

The pool is once again `ONLINE`, and the original bad disk (`c1t0d0`) has been removed from the configuration.

Repairing Damaged Data

ZFS uses checksumming, replication, and self-healing data to minimize the chances of data corruption. Nonetheless, data corruption can occur if the pool isn't replicated, if 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. Two basic types of data can be corrupted:

- Pool metadata – ZFS requires a certain amount of data to be parsed to open a pool and access datasets. If this data is corrupted, the entire pool or complete portions of the dataset hierarchy will become unavailable.
- Object data – In this case, the corruption is within a specific file or directory. This problem might result in a portion of the file or directory being inaccessible, or this problem might cause the object to be broken altogether.

Data is verified during normal operation as well as through scrubbing. For more information about how to verify the integrity of pool data, see "Checking ZFS Data Integrity" on page 137.

Identifying the Type of Data Corruption

By default, the `zpool status` command shows only that corruption has occurred, but not where this corruption occurred. For example:

```
# zpool status tank -v
   pool: tank
     state: ONLINE
    status: One or more devices has experienced an error resulting in data
             corruption.  Applications may be affected.
    action: Restore the file in question if possible.  Otherwise restore the
             entire pool from backup.
       see: http://www.sun.com/msg/ZFS-8000-8A
     scrub: none requested
    config:

        NAME         STATE     READ WRITE CKSUM
```

```
        tank         ONLINE       1     0     0
          mirror     ONLINE       1     0     0
            c2t0d0   ONLINE       2     0     0
            c1t1d0   ONLINE       2     0     0

    errors: The following persistent errors have been detected:

          DATASET  OBJECT  RANGE
          tank     6       0-512
```

Each error indicates only that an error occurred at the given point in time. Each error is not necessarily still present on the system. Under normal circumstances, this situation is true. Certain temporary outages might result in data corruption that is automatically repaired once the outage ends. A complete scrub of the pool is guaranteed to examine every active block in the pool, so the error log is reset whenever a scrub finishes. If you determine that the errors are no longer present, and you don't want to wait for a scrub to complete, reset all errors in the pool by using the `zpool online` command.

If the data corruption is in pool-wide metadata, the output is slightly different. For example:

```
# zpool status -v morpheus
  pool: morpheus
    id: 1422736890544688191
 state: FAULTED
status: The pool metadata is corrupted.
action: The pool cannot be imported due to damaged devices or data.
   see: http://www.sun.com/msg/ZFS-8000-72
config:

        morpheus    FAULTED   corrupted data
          c1t10d0   ONLINE
```

In the case of pool-wide corruption, the pool is placed into the `FAULTED` state, because the pool cannot possibly provide the needed replication level.

Repairing a Corrupted File or Directory

If a file or directory is corrupted, the system might still be able to function depending on the type of corruption. Any damage is effectively unrecoverable. No good copies of the data exist anywhere on the system. If the data is valuable, you have no choice but to restore the affected data from backup. Even so, you might be able to recover from this corruption without restoring the entire pool.

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 to locate the file by using the `find` command and specify the object number that is identified in the `zpool status` output under `DATASET/OBJECT/RANGE` output as the inode number to find. For example:

```
# find -inum 6
```

Then, try removing the file with the `rm` command. If this command doesn't work, the corruption is within the file's metadata, and ZFS cannot determine which blocks belong to the file in order to remove the corruption.

If the corruption is within a directory or a file's metadata, the only choice is to move the file elsewhere. You can safely move any file or directory to a less convenient location, allowing the original object to be restored in place.

Repairing ZFS Storage Pool-Wide Damage

If the damage is in pool metadata that damage prevents the pool from being opened, then you must restore the pool and all its data from backup. The mechanism you use varies widely by the pool configuration and backup strategy. First, 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. Also, keep a file describing the layout of the datasets and the various locally set properties somewhere safe, as this information will become inaccessible if the pool is ever rendered inaccessible. With the pool configuration and dataset layout, you can reconstruct your complete configuration after destroying the pool. The data can then be populated by using whatever backup or restoration strategy you use.

Repairing an Unbootable System

ZFS is designed to be robust and stable despite errors. Even so, software bugs or certain unexpected pathologies might 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 maintains 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. If the system becomes unbootable, boot to the `none` milestone by using the `-m milestone=none` boot option. Once the system is up, remount your root file system as writable and then remove `/etc/zfs/zpool.cache`. These actions cause ZFS to forget that any pools exist on the system, preventing it from trying to access the bad pool causing the problem. You can then proceed to a normal system state by issuing the `svcadm milestone all` command. You can use a similar process when booting from an alternate root to perform repairs.

Once the system is up, you can attempt to import the pool by using the `zpool import` command. However, doing so will likely cause the same error that occurred during boot, because the command uses the same mechanism to access pools. If more than one pool is on the system and you want to import a specific pool without accessing any other pools, you must re-initialize the devices in the damaged pool, at which point you can safely import the good pool.

Index

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z